# WSCAD 2018

## XIX Simpósio em Sistemas Computacionais de Alto Desempenho

# Conteúdo

## Palestras



## WSCAD Artigos Completos

### WSCAD S1 - Escalonamento, Algoritmos e Arquiteturas

**WSCAD S4 e S5 - Avaliação de Desempenho**

**WSCAD S6 - Ferramentas**



**WSCAD S7 - Aplicações de Computação de Alto Desempenho**

## WSCAD Artigos Curtos



## Lista de Autores 490

**Palestra 01**
**Até onde podemos confiar nas especificações de processadores?**
Prof. Rodolfo Azevedo
Instituto de Computação (IC), Universidade de Campinas (UNICAMP)
*rodolfo@ic.unicamp.br*

**Resumo**: No último ano, uma nova categoria de falhas de processadores ganhou grande importância, as falhas que exploram efeitos colaterais da execução especulativa. A principal característica delas é que não é possível detectá-las através de um efeito incorreto na execução de programas mas somente através de efeitos colaterais, especialmente relacionados com temporização. As três falhas mais conhecidas são Spectre, Meldtown e Foreshadow. Em comum, elas tiram proveito da execução fora de ordem dos processadores superescalares modernos e também da precisão das medidas de tempo dos mesmos processadores para descobrirem informações que o sistema de segurança do hardware não deveria permitir. Na sequência será falado sobre outras formas de investigar processadores e como isto pode permitir melhor detecção de falhas no futuro.

**Palestra 02**

**Analisando o comportamento de usuários e sistemas de HPC: O caso do supercomputador Santos Dumont**

Antônio Tadeu Azevedo Gomes,
Pesquisador no Laboratório Nacional de Computação Científica (LNCC), Brasil
*atagomes@lncc.br*

**Resumo**: O supercomputador SDumont, hospedado no LNCC, na cidade de Petrópolis-RJ, atende atualmente mais de 500 usuários e 100 projetos de pesquisa, em 16 áreas de pesquisa distintas, coordenados por instituições de ensino e pesquisa distribuídas em 11 estados brasileiros. Nesta palestra serão apresentadas brevemente as características do supercomputador e as áreas e projetos de pesquisa atualmente em desenvolvimento no mesmo. Em seguida, será apresentada uma análise do perfil de jobs executados no SDumont desde sua inauguração. Essa análise, baseada nos logs do gerenciador de recursos do SDumont, abre possibilidades de investigação e emprego de técnicas de ajuste automático de configurações neste e outros ambientes de HPC.

**Palestra 03**

## Microarchitecture, Computing Systems, High Performance Computing and End-to-End Deep Neural Networks

Prof. Mauricio Breternitz (ISCTE-IUL - ISTAR – DLS - Portugal)
*Mauricio.Breternitz.Jr@iscte-iul.pt*

**Abstract**: In this multi-part presentation I intend to discuss highlights from my experience as a researcher in industrial labs in the USA (at IBM, Motorola, Intel Labs and IBM Research), discussing the microarchitecture computing systems, and related compilation issues. Then I intend to discuss current and future experience in high performance computing systems leading to Exascale, from system organization to performance analysis and big data applications. Finally I describe current investigation topics which focus on novel applications of end-to-end deep neural networks, focusing on system organization and research opportunities. I am also available to share experiences related to industry and university interaction and collaboration.

**Palestra 04**
**Improved Static Analysis to Generate More Efficient Code for Execution of Loop Nests in GPUs**

Prof. Jose Nelson Amaral (University of Alberta - Canada)
*jamaral@ualberta.ca*

**Resumo**: Em 2018 a IBM e a Nvidia estão entregando dois supercomputadores para o Departamento de Energia dos Estados Unidos. Estes supercomputadores utilizam o POWER 9, a GPU Volta, e a rede de interconexão NVLink. Esta palestra vai descrever como o trabalho de dois alunos de mestrado na Universidade de Alberta foi fundamental para melhorar a qualidade do código gerado para programas escritos em OpenMP 4.x pelos compiladores entregues pela IBM para estes supercomputadores. Este trabalho começou com o desenvolvimento de uma nova análise chamada Arithmetic Control Form (ACF) que aplica os princípios de análise de mancha (taint analysis) para detectar divergência de controle e de acesso de dados em programas escritos em CUDA. Subsequentemente uma outra nova análise, chamada Interaction Point Algebraic Difference (IPAD) foi capaz de provar a não existência de dependências em vários ninhos de laços em OpenMP e possibilitou transformações de laços que levaram a ganhos dramáticos de performance nos programas gerados para execução na GPU Volta. Além de descrever os avanços em análise de programas e geração de código desenvolvidos na University of Alberta, esta palestra também vai descrever o processo que levou estes dois alunos de mestrado a influenciar significativamente os compiladores para estes supercomputadores.

**Palestra 05**
**Intel OpenVINO - Video Inference on the Edge**
Jomar Silva (Intel)

**Resumo**: OpenVINO™ é um software gratuito que ajuda desenvolvedores e cientistas de dados a acelerar workloads de visão computacional, agilizar implementação de soluções de inferência de redes neurais, permitindo a execução simplificada e em hardware heterogêneo em plataformas Intel®, abrangendo dispositivos da ponta até a nuvem. Uma mesma base de código pode ser utilizada em CPU, GPU, FPGA e VPU, flexibilizando e simplificando a distribuição de aplicações avançadas de vídeo.

# Analysis of Potential Online Scheduling Improvements by Real-Time Strategy Selection

**Luis Felipe Sant'Ana**[1], **Danilo Carastan-Santos**[1,2]
**Daniel Cordeiro**[3], **Raphael Y. de Camargo**[1]

[1]Center for Mathematics, Computation and Cognition – Federal University of ABC
Santo André – SP – Brazil

{luis.ana, danilo.santos, raphael.camargo}@ufabc.edu.br

[2]Univ. Grenoble Alpes, CNRS, Inria, LIG
Grenoble – France

[3]Schoool of Arts, Sciences and Humanities – University of São Paulo
São Paulo – SP – Brazil

daniel.cordeiro@usp.br

***Abstract.*** *Task Scheduling in large-scale HPC platforms is normally accomplished with simple heuristics combined with a backfilling algorithm. Some strategies, such as the First-Come-First-Served (FCFS) with backfilling, provide reasonable results in a variety of scenarios, including different HPC platforms and task set characteristics. But for each scenario, a different strategy might be the most appropriate for minimizing some metric, such the as the average task waiting time or turnaround time. In this work, we evaluate the effects of choosing different scheduling strategies over sub-sequences of workload logs of 6 real HPC platforms, for periods from 1 to 24 hours. For each platform and workload period, we show that the performance of each scheduling strategy have large variations for different workload sub-sequences. Similarly, the best scheduling strategy for each sub-sequence also varied. Finally, we show that, if one could select the best strategy for each workload sub-sequence, it would significantly reduce the scheduling performance variations and improve the mean queue waiting time by more than 50% for most cases. These results indicate that the development of heuristics or machine learning algorithms for selecting the best scheduling strategy every 6 or 24 hours, can result in significant improvements in the mean queue waiting time of tasks in HPC platforms.*

## 1. Introduction

The use of High Performance Computing (HPC) is becoming more common in several domains of research, resulting in an ever increasing acquisition of HPC platforms by companies and research centers. Typically, an HPC platform can be described as a large set of interconnected computers whose purpose is to process computational (and potentially parallel) tasks that are submitted by HPC platform users. To operate such HPC platform environments, Resource and Job Management Systems (RJMS) are often deployed. In a standard scenario, tasks arrive unpredictably (i.e. in an on-line manner) and are placed into a waiting queue. At this point, the RJMS must define a task execution order that increases the execution efficiency, according to one or multiple performance metrics.

Establishing an efficient execution order for this dynamic waiting queue is a non-trivial problem and in the literature this problem is called on-line parallel job scheduling problem [Brucker 2007]. Though a considerable amount of research is being applied to this problem [Bougeret et al. 2011, Ye et al. 2011, Hurink and Paulus 2007, Xhafa and Abraham 2010, Chakaravarthy et al. 2013], arguably most of existing RJMSs employ simple scheduling heuristics — represented as a scheduling policy — combined with a backfilling mechanism [Mu'alem and Feitelson 2001], which increases the overall utilization of the HPC platform [Skovira et al. 1996].

Scheduling policies normally consist of functions that receive as parameters information about the tasks that is commonly available by the RJMS upon the tasks arrival, such as processing time estimate, requested number of resources and waiting time. The output of these functions is a priority value for each task in the waiting queue. Such priority values can therefore be used by the RJMS to set task execution order. Though most of RJMSs use a fixed scheduling policy, recent studies [Lelong et al. 2017, Deng et al. 2013] indicate that, for a certain performance metric, switching the scheduling policy in certain periods of time can result in performance increases.

In this work we evaluate the conditions on which performance improvements can be obtained by switching scheduling policies. In this regard, this paper presents the following contributions:

- We introduce an experimental framework for evaluating the effects of switching scheduling policies for on-line schedulers in HPC platforms. The framework exploits the rich information that can be obtained from HPC workload logs and accurate HPC simulation software. It provides valuable insights on how the real HPC platform and workload would behave under a well-controlled set of parameters.
- We show how the proposed framework can be used to analyze the effects of the on-line switching of scheduling policies over six real HPC workload logs and using four workload sub-sequence lengths. Our experimental results show that: (i) periodically switching to the best scheduling policy generates potential performance improvement of up to 50% and (ii) changing the periodicity of the switching process has no clear effect on the potential performance gains.

The remainder of this paper is organized as follows. In Section 2 we present the closely related works and in Section 3 we present the proposed experimental framework. We present the main results in Section 4, and the conclusions in Section 5.

## 2. Related Work

In the scheduling literature, several approaches were proposed to solve the many classes of scheduling problems, covering from integer linear programming [Floudas and Lin 2005, Al-Daoud et al. 2010] to genetic algorithms [Pecero et al. 2009, Hou et al. 1994] and neural networks [Agarwal et al. 2006]. Xhafa and Abraham [Xhafa and Abraham 2010] present a review of recent computational models and heuristics for scheduling on HPC platforms.

More specifically for the parallel job scheduling problem, many tackle it under the view of the directly related multiple-strip packing problem [Bougeret et al. 2011, Ye et al. 2011, Hurink and Paulus 2007, Zhuk 2006], with most of them focused on proposing

approximation algorithms and computational bounds to the problem. However, as noticed in [Feitelson et al. 1997], most of the HPC management software (notably RJMSs) rely on low complexity heuristics, with the First-Come-First-Served with aggressive backfilling [Mu'alem and Feitelson 2001] (also known as EASY-backfilling) arguably being the most popular heuristic.

In an attempt to close the gap between theory and practice, some works approach the parallel job scheduling problem with low complexity heuristics [Tang et al. 2009, Gaussier et al. 2015, Carastan-Santos and de Camargo 2017]. Lelong *et al.* [Lelong et al. 2017] propose an interestingly simple idea to tune the EASY-backfilling heuristic by defining two queues, a primary and a backfilling queue. By just changing the scheduling policy of the primary queue in real-time, they achieved good performance improvements with regard to the average waiting time. This simple and innovative idea encouraged us to discover empirical conditions and limits on what can be gained by just switching the scheduling strategy in real-time, which is the main focus of this work.

## 3. Methodology

Our goal is to assess what would be the behavior of the on-line scheduling performance when multiple scheduling policies can be used and switched in real-time, considering several scenarios and HPC workload logs. To achieve such goal, we developed a simulation framework that enables the observation of this on-line scheduling behavior.

### 3.1. Preliminary Definitions

In this work we consider an HPC platform as constituted by a set of $m$ homogeneous processing resources connected by a interconnection topology. In the on-line setting, parallel and rigid (i.e. fixed and known in advance number of required resources) tasks arrive in a centralized waiting queue at any moment in time. For each task $t$, we consider the following characteristics:

- The actual processing time $p_t$ of the task (only known after the task has been executed);
- The resource requirement of the task, measured as the number of cores $q_t$;
- The estimated processing time $\tilde{p}_t$ of the task informed by the user (frequently considered an upper bound of $p_t$);
- The arrival time $r_t$ of the task (also called release date)

A common practice taken by HPC platform administrators is to register these characteristics in workload logs, shared using the Standard Workload Format (SWF) [Feitelson et al. 2014]. This is the main source of data that we use to drive the simulations that are explained in Section 3.3.

There are a large number of cost metrics [Feitelson and Rudolph 1998] — which focus on different performance aspects of the scheduling — that can be used by HPC platform administrators. In this work, we focus on two task-oriented metrics. The fist metric is the *waiting time* (Equation 1) which, as its name suggests, measures the time that the task waited for execution, and it can be defined for a task $t$ as:

$$wait_t = start_t - r_t \tag{1}$$

where $start_t$ is the time in which $t$ starts processing. The second metric is the *turnaround time* (also found in the literature as *flow time*, Equation 2), which measures the time that the tasks spent in the HPC platform and it can be defined as follows for a task $t$:

$$turnaround_t = wait_t + p_t \tag{2}$$

For a queue of tasks $Q$, we consider the average waiting time and average turnaround time of $Q$ as being the average of these two metrics respectively, over all tasks $t \in Q$.

### 3.2. On-line Scheduling Strategies

The on-line scheduling strategy works as follows: the scheduler sorts — in increasing order according to a scheduling policy $f(t)$ — its waiting queue in two distinct events: (i) when a task arrives in the queue or (ii) when a resource (set of processors) is released and becomes available. When a task $t$ is selected for execution and if the requested number of processors $q_t$ is lower than the total number of processors available, then $q_t$ processors are reserved for this task and they become unavailable. These processors will become available again only when $p_t$ units of time have passed since the start of the execution of $t$.

If there are not enough processors to process $t$, then we follow the aggressive backfilling [Mu'alem and Feitelson 2001] strategy. In this strategy, it is estimated at which time there will be enough resources to process $t$. Next, the scheduler looks for tasks $t_b$ in the waiting queue — following the order of tasks already established by the scheduling policy $f(t)$ — for which there are enough processing resources and that do not delay the execution of $t$. If these conditions match for a task $t_b$, then $t_b$ "jumps ahead" and is scheduled for execution.

A key component of this scheduling strategy is the scheduling policy $f(t)$. In this regard, we make use of seven classical policies, shown in Table 1. These policies are simple functions that take as input one or more characteristics of the tasks. The challenge is on how to frame a experimental procedure that simulates the aforementioned on-line scheduler, and also captures the scheduling behavior when $f(t)$ is changed on real-time. We address this challenge in the next Section.

### 3.3. Simulation Framework

Let $T$ be a sub-sequence of tasks that were obtained from a HPC workload log, sorted by arrival time. We start each simulation using the tasks from $T$, which arrive into the waiting queue $Q$ at their arrival times $r_t$ and are scheduled using a fixed scheduling policy $f_0$. The simulation uses the same number of processors $m$ from the original HPC platform.

At any point in the simulation, we can compute $AvgPerf(Q)$, which is the average of a performance metric, such as task waiting time, over the tasks present in $Q$. To determine $AvgPerf(Q)$, we spawn a new simulation, where we freeze the arrival of new tasks and continue until all tasks in $Q$ finish their execution.

In order to establish a baseline state to consistently compare the scheduling performance of different policies, we start with a pre-simulation phase which we call as a *warm-up* period. For every $\rho$ units of simulation time, we calculate the value of $AvgPerf(Q)$.

The *steady point* (sp) is reached when (i) $Q$ is not empty and (ii) the difference between twelve subsequent calculations of $AvgPerf(Q)$ is less than 5%. The warm-up process is shown in Figure 1. We define $Q_{sp}$ as the set of tasks present in the waiting queue $Q$ at the steady point. We also define $P_0 = AvgPerf(Q_{sp})$.

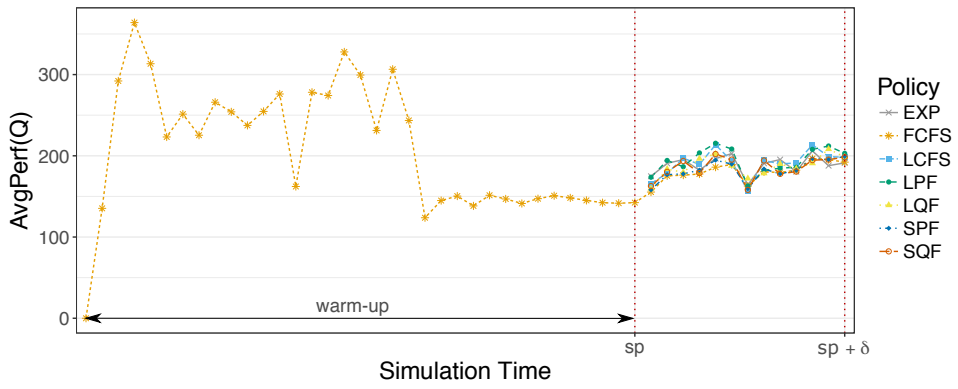



**Figure 1. Example of a simulated execution, with the warm-up period finishing when the average of the performance metric $AvgPerf(Q)$ over the queue $Q$ reaches a steady value. The seven policies are then executed and their performance $AvgPerf(Q_{sp+\delta})$ can be compared.**

As well illustrated in Figure 1, after reaching the steady point, we continue the simulation, switching $f_0$ with a scheduling policy $f$ for $\delta$ units of simulation time, until reaching time $sp + \delta$. We then evaluate $P_1^f = AvgPerf_f(Q_{sp+\delta})$ and $\Delta_f = P_1^f - P_0$ for the policy $f$. We repeat this procedure for each scheduling policy $f$ listed in Table 1.

**Table 1. Scheduling policies used for comparison.**

| Name | Description | Function |
|---|---|---|
| FCFS | First-Come-First-Served [Skovira et al. 1996] | $f(t) = r_t$ |
| LCFS | Last-Come-First-Served | $f(t) = -r_t$ |
| SPF | Smallest Estimated Processing Time [Srinivasan et al. 2002] | $f(t) = \tilde{p}_t$ |
| LPF | Longest Estimated Processing Time | $f(t) = -\tilde{p}_t$ |
| SQF | Smallest Resource Requirement First | $f(t) = q_t$ |
| LQF | Largest Resource Requirement First | $f(t) = -q_t$ |
| EXP | Largest Expansion Factor First [Srinivasan et al. 2002] | $f(t) = -(wait_t + \tilde{p}_t)/\tilde{p}_t$ |

$\Delta_f$ is the main result obtained by the simulations. This value represents how much a scheduling policy $f$ was capable to handle a workload represented by the waiting queue evolution from $Q_{sp}$ to $Q_{sp+\delta}$. The lower the value of $\Delta_f$, the better $f$ was able to handle this workload. Another important result is the best scheduling policy found, defined as $best\_choice = \min_f(\Delta_f)$.

## 4. Results

In this section we present the main results obtained by our work. We divided the experiments into three parts: (i) Best policy selection distribution, (ii) Best policy selection behavior, and (iii) Potential scheduling improvements.

### 4.1. Experiments Settings

We implemented the simulation framework and the scheduling algorithm presented in the previous sections using the SimGrid [Casanova et al. 2014] HPC simulator software. We evaluated six HPC workload logs (shown in Table 2) that are publicly available at the Parallel Workloads Archive [Feitelson et al. 2014]. For each workload log, we partitioned it into 50 non-overlapping sub-sequences, and applied our proposed simulation framework for each of these sub-sequences. To establish the steady point in all experiments, we adopted $\rho = 30$ minutes and $f_0 =$ FCFS. Next we present some results obtained with these experiments.

**Table 2. Workload logs used in the experiments.**

| Name | Year | # CPUs | # Jobs | Log Length |
|---|---|---|---|---|
| CTC-SP2 | 1996 | 338 | 77k | 11 Months |
| HPC2N | 2002 | 240 | 202k | 42 Months |
| KTH-SP2 | 1996 | 100 | 28k | 11 Months |
| LPC EGEE | 2004 | 140 | 234k | 9 Months |
| OSC Cluster | 2000 | 178 | 36k | 22 Months |
| Sandia Ross | 2005 | 1,524 | 57k | 37 Months |

### 4.2. Best Policy Selection Distribution

In this section we aim to give light to the following question: *Is there a clear dominance of a certain scheduling policy, that always provide the best results?*

We evaluated how often a scheduling policy $f$ from Table 1 was the $best\_choice$ for the sub-sequences of each workload log, when using different time periods $\delta = \{1, 6, 12, 24\}$ hours. Figure 2 shows the distribution of the $best\_choice$ found over the experiments. We can see that, with the exception of the CTC-SP2 log — where the SPF policy showed a certain dominance — there is no clear distinction of a certain scheduling policy being dominant in all scenarios, and sometimes there is a relative homogeneity among the policies, notably in the Sandia Ross workload log. These results indicate that choosing a single best scheduling policy, even when considering a single machine and analyzing the workload logs, may not be the best strategy.

Another important observation is that the $best\_choice$ distribution can change substantially, when considering different time periods $\delta$ and workload logs from different machines. For example in the LPC EGEE log, the more frequent $best\_choice$ was different for all of the time periods considered.

### 4.3. Best Scheduling Policy Selection Behavior

In this section we focus on the following question: *If one could always select the* $best\_choice$ *policy, how this would reflect in the scheduling performance?*

We compared the distribution of $\Delta_f$ values over all 50 non-overlapping sub-sequences for each HPC platform, workload sub-sequence size and scheduling policy $f$. Figure 3 shows that in most scenarios, the median of $\Delta_f$ of different policies was comparable, but the range of $\Delta_f$ values between the first and third quartiles (box boundaries) is substantial. Also, in the scenarios where there was a dominance of a scheduling policy

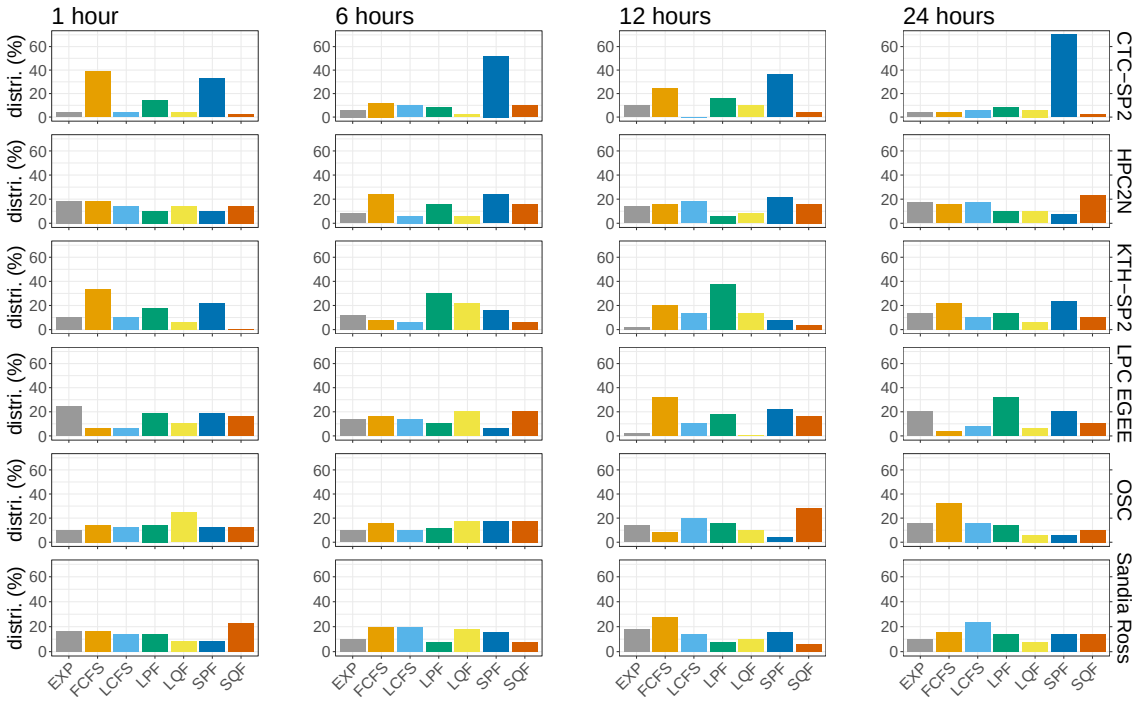



**Figure 2. Distribution of $best\_choice$ among the evaluated scheduling policies for six HPC workload logs and four sub-sequence periods ($\delta = \{1, 6, 12, 24\}$).**

— which is the case of the SPF policy for the CTC-SP2 workload log — this dominant policy showed a lower median and smaller variations from the median.

When we selected the $best\_choice$ policy for each non-overlapping sub-sequence, we obtained consistently lower median and a small range of $\Delta_f$ values between the first and third quartiles. This indicates that selecting the best policy results not only in better scheduling performance, but results also in a more predictable performance.

Using the turnaround time as scheduling performance metric resulted in a very similar behavior, as shown in Figure 4. This occurs because the turnaround is defined as the sum of the waiting and processing times, and the former dominates the sum.

### 4.4. Potential Scheduling Improvements

In this section we assess the question: *What is the potential for scheduling performance improvement obtainable by always selecting the* $best\_choice$ *policy?*

Similarly to Section 4.3, we used $n = 50$ non-overlapping sub-sequences for each HPC platform and sub-sequence length. We defined $ratio_f$ (Equation 3) for each policy $f$, which measures the potential improvements when using the $best\_choice$ policy, in comparison to the original policy $f$. The sums are over $i = 1 \ldots n$ and $\Delta_f^i$ is the value of $\Delta_f$ for sub-sequence $i$. A $ratio_f$ value 1 means that the mean improvement was comparable in value to the original $\Delta_f$ and 0 means no improvement.

$$ratio_f = \frac{\sum_{i=1}^{n}(\Delta_f^i - \Delta_{best\_choice}^i)}{\sum_{i=1}^{n} \Delta_f^i} \quad (3)$$

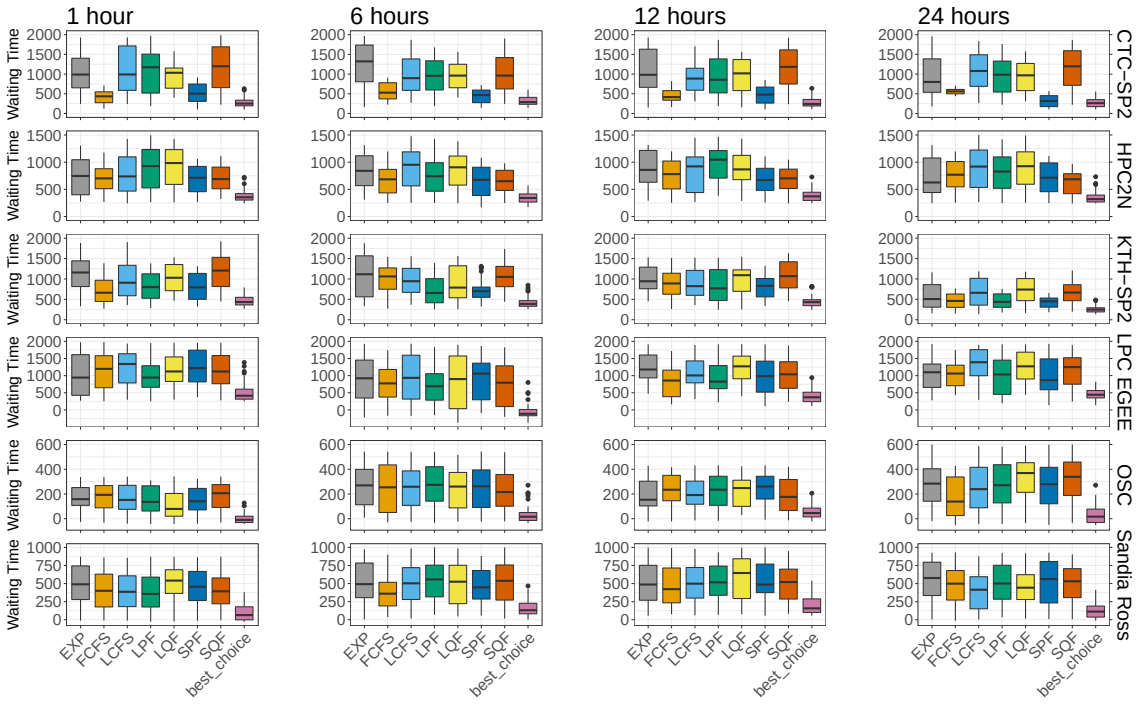



**Figure 3. Boxplot chart for six workload logs, seven fixed scheduling policies and the best policy $best\_choice$ found, for time periods $\delta = \{1, 6, 12, 24\}$, for the waiting time performance metric.**

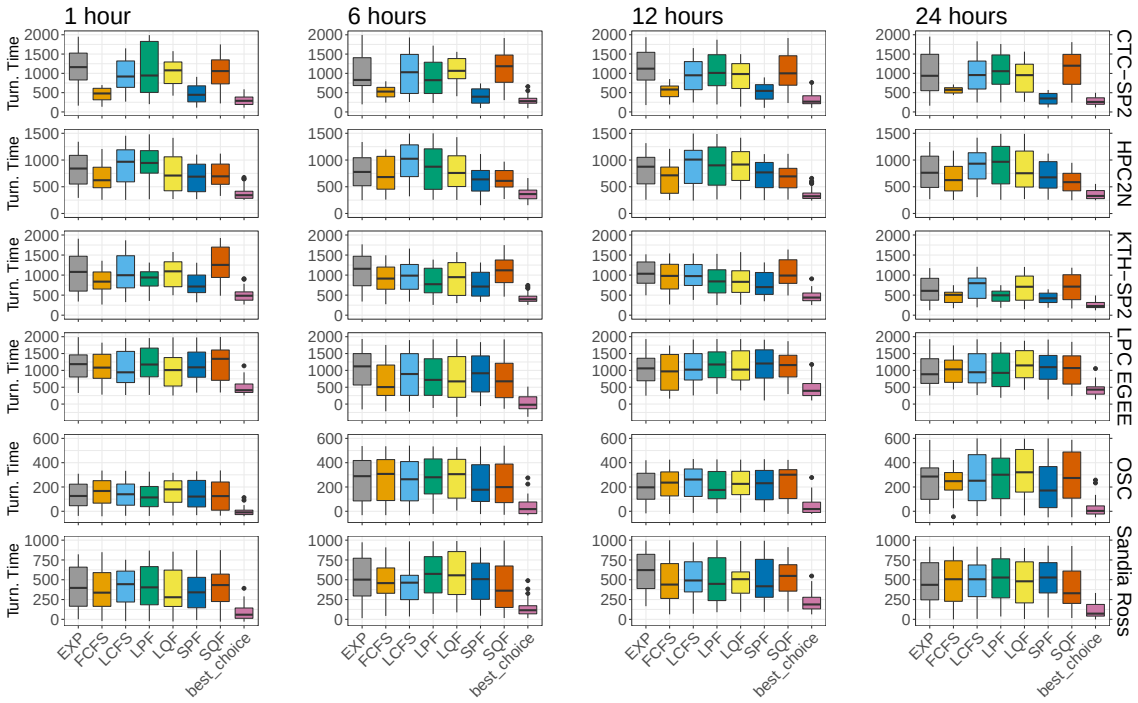



**Figure 4. Same as Figure 3, but with turnaround time performance metric.**

The potential gains, denoted by $ratio_f$, varied depending on the HPC architecture (Figure 5). The $ratio_f$ for the turnaround and waiting times are very similar, as in Section 4.3. For some architectures, such as Sandia Ross and OSC, the waiting times

changes $\Delta_f$ had a mean reduction always above $50\%$. For others the reduction was about $40\%$, while for the CTC, where the SPF algorithm performed well, the gain was only $12.5\%$. These results indicate that scheduling performance metrics could be greatly improved in most of the evaluated architectures by choosing the best scheduling policy for each sub-sequence.

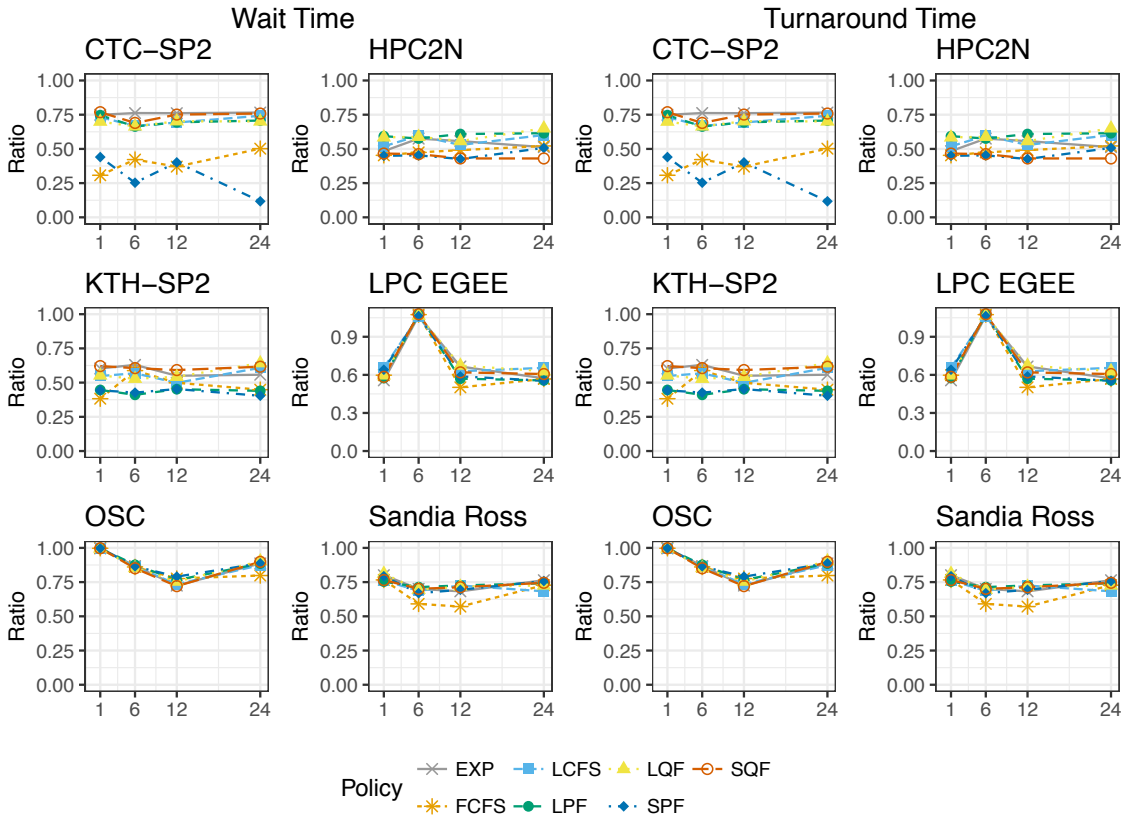



**Figure 5. Values of $ratio_f$, which shows the relative improvement obtained when using the $best\_choice$ policy, for six HPC platform logs and seven scheduling policies, for the waiting and turnaround time metrics.**

## 5. Conclusions

In this work we examined on both quantitative and qualitative ways how scheduling policies affect the performance of the on-line parallel job scheduling in High Performance Computing (HPC) platforms, and how the scheduling behaves when these policies are changed in real-time during the operation of the platform. For this, we proposed a simulation framework that enables the evaluation of such scheduling scenario, relying on information present in real HPC workload logs to maintain an adequate representation of the real platform into the simulations.

We applied our simulation framework using workload logs from six HPC platforms. The first showed that its not possible to find an universal best choice of best scheduling policy, and the policy who gives the best performance highly depends on both the HPC workload log and the time period evaluated. This indicates that an approach that tries to discover such best policy would face generalization hindrances, on both workload and time period levels.

However, our results also indicate that important gains in scheduling performance metrics could be obtained if one can find this best scheduling policy. It could provide better and more consistent scheduling performance, with improvements above 50% and lower variances in the average waiting time, in comparison with the best performing fixed scheduling policy. These are encouraging results, giving directions for future research work towards finding machine learning based algorithms to automatically select the best (or close to) scheduling policy, based on information such as HPC platform state and queue and workload characteristics.

## Acknowledgment

The authors would like to thank UFABC and FAPESP (Proc. n. 2013/26644-1) for the financial support.

## References

Agarwal, A., Colak, S., Jacob, V. S., and Pirkul, H. (2006). Heuristics and augmented neural networks for task scheduling with non-identical machines. *European Journal of Operational Research*, 175(1):296 – 317.

Al-Daoud, H., Al-Azzoni, I., and Down, D. G. (2010). Power-aware linear programming based scheduling for heterogeneous computer clusters. In *International Conference on Green Computing*, pages 325–332.

Bougeret, M., Dutot, P.-F., Jansen, K., Robenek, C., and Trystram, D. (2011). Approximation Algorithms for Multiple Strip Packing and Scheduling Parallel Jobs in Platforms. *Discrete Mathematics, Algorithms and Applications*, 3(4):553–586.

Brucker, P. (2007). *Scheduling Algorithms*. Springer, 5th edition.

Carastan-Santos, D. and de Camargo, R. Y. (2017). Obtaining dynamic scheduling policies with simulation and machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 32:1–32:13, New York, NY, USA. ACM.

Casanova, H., Giersch, A., Legrand, A., Quinson, M., and Suter, F. (2014). Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917.

Chakaravarthy, V. T., Choudhury, A. R., Roy, S., and Sabharwal, Y. (2013). Scheduling jobs with multiple non-uniform tasks. In *European Conference on Parallel Processing*, pages 90–101. Springer.

Deng, K., Song, J., Ren, K., and Iosup, A. (2013). Exploring portfolio scheduling for long-term execution of scientific workloads in iaas clouds. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–12. IEEE.

Feitelson, D. G. and Rudolph, L. (1998). Metrics and benchmarking for parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–24. Springer.

Feitelson, D. G., Rudolph, L., Schwiegelshohn, U., Sevcik, K. C., and Wong, P. (1997). Theory and practice in parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer.

Feitelson, D. G., Tsafrir, D., and Krakov, D. (2014). Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967 – 2982.

Floudas, C. A. and Lin, X. (2005). Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications. *Annals of Operations Research*, 139(1):131–162.

Gaussier, E., Glesser, D., Reis, V., and Trystram, D. (2015). Improving backfilling by using machine learning to predict running times. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 64:1–64:10, New York, NY, USA. ACM.

Hou, E. S. H., Ansari, N., and Ren, H. (1994). A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113–120.

Hurink, J. L. and Paulus, J. J. (2007). Online algorithm for parallel job scheduling and strip packing. In *International Workshop on Approximation and Online Algorithms*, pages 67–74. Springer.

Lelong, J., Reis, V., and Trystram, D. (2017). Tuning EASY-Backfilling Queues. In *21st Workshop on Job Scheduling Strategies for Parallel Processing*, 31st IEEE International Parallel & Distributed Processing Symposium, Orlando, United States.

Mu'alem, A. W. and Feitelson, D. G. (2001). Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543.

Pecero, J. E., Trystram, D., and Zomaya, A. Y. (2009). A new genetic algorithm for scheduling for large communication delays. In *European Conference on Parallel Processing*, pages 241–252. Springer.

Skovira, J., Chan, W., Zhou, H., and Lifka, D. (1996). The EASY—LoadLeveler API project. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 41–47. Springer.

Srinivasan, S., Kettimuthu, R., Subramani, V., and Sadayappan, P. (2002). Characterization of backfilling strategies for parallel job scheduling. In *Parallel Processing Workshops, 2002. Proceedings. International Conference on*, pages 514–519. IEEE.

Tang, W., Lan, Z., Desai, N., and Buettner, D. (2009). Fault-aware, utility-based job scheduling on BlueGene/P systems. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE.

Xhafa, F. and Abraham, A. (2010). Computational models and heuristic methods for grid scheduling problems. *Future Generation Computer Systems*, 26(4):608 – 621.

Ye, D., Han, X., and Zhang, G. (2011). Online multiple-strip packing. *Theoretical Computer Science*, 412(3):233 – 239. Combinatorial Optimization and Applications.

Zhuk, S. (2006). Approximate algorithms to pack rectangles into several strips. *Discrete Mathematics and Applications*, 16(1):73–85.

# Avaliação de Performance para Fornecer StaaS a Dispositivos IoT em Ambiente Fog Computing

**José dos Santos Machado, Danilo Souza Silva, Raphael Silva Fontes, Adauto Cavalcante Menezes, Edward David Moreno, Admilson de Ribamar Lima Ribeiro**

Departamento de Computação, Universidade Federal de Sergipe – UFS – Brasil

jsmac18@hotmail.com, {danilosilva.se, raphaelf.ti, adauto.cavalcant, edwdavid}@gmail.com, admilson@ufs.br

***Resumo.*** *Este trabalho apresenta a Fog Computing, sua contextualização teórica, os trabalhos correlatos e realiza avaliação de uma Fog Computing, para fornecer StaaS (Storage as a Service), a dispositivos IoT utilizando plataformas de sistemas embarcados e compara seus resultados com os obtidos por um servidor de alto desempenho. Os resultados demonstraram que a implementação desse serviço em dispositivos de sistemas embarcados pode ser uma boa alternativa para reduzir um desses problemas, no caso, o armazenamento de dados, que atinge hoje os dispositivos IoT.*

## 1. Introdução

Nas últimas décadas, observa-se que serviços de computação como armazenamento, processamento de dados e controle foram transferidos para a "nuvem". A oportunidade de computação ilimitada na nuvem permite que os usuários finais acessem amplas informações facilmente, também é possível visualizar que os dispositivos móveis e sensores, como *smartphones*, se tornaram poderosos equipamentos, o que levou ao surgimento de novos sistemas e aplicações.

Estes sistemas e aplicações introduzem novas demandas funcionais em computação e redes que a "nuvem" sozinha não pode atender. Neste cenário percebe-se que a computação local na borda da rede é muitas vezes necessária [7]. Por exemplo, para processar dados em tempo real, criar contextos de reconhecimento de localização a partir de sensores locais e maximizar a eficiência das comunicações sem fio na borda da rede. Em geral, a nuvem está muito longe dos dispositivos para satisfazer requisitos de latência e, é muito centralizada para lidar com a heterogeneidade e diversidade contextual em uma área local [14].

Para ultrapassar estas limitações, porções da capacidade de computação da nuvem podem ser deslocados para a borda da rede e formam um ambiente de computação local, isto é, uma "*Fog Computing*" chamada também de "nevoeiro" [7]. Ao distribuir os serviços de computação e de rede mais próximos de onde os dados do usuário são gerados, a *Fog* atende melhor às demandas emergentes [13].

A *Fog Computing* apresenta uma nova arquitetura que "leva processamento para os dados", enquanto a nuvem "leva os dados para o processamento" [1]. Dessa maneira dispositivos de borda e dispositivos móveis podem estar interligados dentro de uma rede local e executar colaborativamente tarefas de armazenamento, processamento de dados de rede e de controle [4].

A *Fog Computing* pode vir a resolver muitos problemas da Internet das Coisas (IoT), por exemplo, os serviços da *Fog* serão capazes de melhorar a largura de banda e as restrições de custo das comunicações de longo alcance. No entanto, muitos desafios ainda permanecem na computação em *Fog*, como modelar uma arquitetura de sistema para a *Fog*; como implementar, organizar e gerenciar dispositivos da *Fog*; como a *Fog* interage com a nuvem e com os dispositivos; e como gerenciar a conectividade física e lógica da *Fog*; entre outros.

Este trabalho apresenta o conceito da *Fog Computing*, os trabalhos correlatos e realiza a análise do fornecimento StaaS (*Storage as a Service*), a dispositivos IoT utilizando plataformas de sistemas embarcados em um ambiente *Fog Computing* e compara seus resultados com os obtidos por um servidor de alto desempenho.

O artigo está organizado em seis seções, a seção 2 apresenta o conceito e características da *Fog Computing*, na seção 3 é dedicada a revisão da literatura, a seção 4 temos o cenário de teste, na seção 5 avaliação e resultados e por fim na seção 6 as conclusões e trabalhos futuros.

## 2. Fog Computing

Devido à latência de rede frequentemente imprevisível, especialmente em um ambiente móvel, muitas vezes a computação em nuvem não pode atender aos requisitos rigorosos de latência, segurança e privacidade dos aplicativos em área restrita geograficamente [16]. Por outro lado, a crescente quantidade de dados gerados por dispositivos e sistemas, com poucos recursos pode se tornar impraticável para transportar dados através de redes para nuvens remotas [11].

Para isso, surgiu um novo paradigma, a *Fog Computing*. O conceito de computação em *Fog* foi adotado pela *Cisco Systems* como um novo paradigma em 2012 [4]. A *Fog Computing* é a computação em nuvem que distribuirá serviços avançados de computação, armazenamento, rede e gerenciamento mais próximos dos usuários finais, enviando informações dos dispositivos IoT para *Cloud Computing*, formando assim uma plataforma distribuída e virtualizada, assim, também é referido como computação de borda [6].

### 2.1 Características da Fog Computing

A computação *Fog* é um paradigma inovador que realiza computação distribuída, serviços de rede e armazenamento, além da comunicação entre *Cloud Computing Data Centers* até os dispositivos ao longo da borda da rede. Essa comunicação amplia as operações e serviços inerentes à computação em nuvem, permitindo assim uma nova geração de aplicativos [11]. A principal função é filtrar e agregar dados para os centros de dados da *Cloud* e aplicar inteligência lógica a dispositivos finais [7]. A figura 1 mostra a localização entre *Fog Computing* e *Cloud Computing*. A figura 2 apresenta arquitetura da *Fog Computing* com a sua localização.

Fig. 1 - Fog Localizada Entre Borda e Nuvem [14]

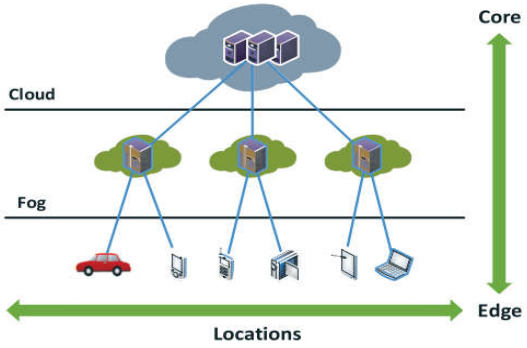
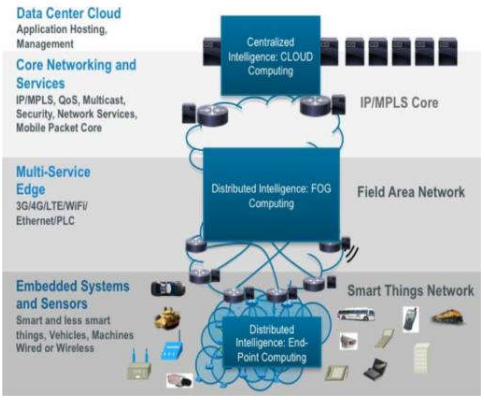



Fig. 2 - Arquitetura da Fog Computing [16]




*Fog Computing* é semelhante à computação em nuvem em muitos aspectos, no entanto, pode ser diferenciado do anterior pelo fato de estar próximo dos dispositivos finais, a distribuição espacial geograficamente menor e a possibilidade de apoio à mobilidade [7]. Como o processamento baseado em *Fog* ocorre ao longo da borda da rede, os resultados finais refletem uma percepção de localização altamente melhorada, baixa latência e Qualidade de Serviço (QoS), em aplicações de streaming e tempo real, os nós de nevoeiro são dispositivos heterogêneos, que vão desde servidores, pontos de acesso, roteadores de borda até dispositivos finais como telefones celulares, relógios inteligentes, sensores e etc [1].

### 2.2 Armazenamento como Serviço (StaaS)

A computação em nuvem e, em particular, os serviços de armazenamento em nuvem tornaram-se uma parte cada vez mais importante do setor de tecnologia da informação nos últimos tempos. O número de opções de armazenamento em nuvem está aumentando, com a maioria dos fornecedores fornecendo uma quantidade variada de armazenamento livre antes de cobrar por níveis de armazenamento maiores, devido ao número crescente desses serviços disponíveis, muitos pesquisadores usaram a frase *Storage as a Service* (StaaS), como uma extensão do *Software as a Service*, para descrever esse tipo de serviço [9].

## 3. Trabalhos Analisados

Um total de oito artigos foram identificados na implementação da *Fog Computing*, para melhorar alguns dos serviços da integração entre a *Cloud* e os dispositivos IoT, apresentaremos seus resultados e seus respectivos trabalhos futuros de pesquisas.

ZHU *et al*. (2013) [16], apresentaram a otimização da web dentro do contexto *Fog Computing*. Aplicaram métodos existentes para otimização da web de uma maneira inovadora, de tal forma que, esses métodos podem ser combinados com conhecimento exclusivo que está disponível apenas nos nós de borda (*Fog*). Como trabalho futuro sinalizaram aplicar seus conceitos propostos para desenvolver um sistema de prova de conceito.

No trabalho de CRACIUNESCU *et al.* (2015) [2], apresentaram uma implementação em laboratório de e-Saúde, onde o processamento em tempo real é realizado pelo PC doméstico, enquanto os metadados extraídos são enviados para a nuvem para processamento posterior. Como trabalho futuro sinalizaram adicionar mais sensores e mais dispositivos *off-the-shelf*, que são atualmente *mainstream*, e os dados de fusão desses dispositivos.

Em [15] VERBA *et al*. (2016), analisaram as plataformas existentes e suas deficiências, bem como propuseram uma plataforma de gateway modular, baseada em mensagens que permite o agrupamento de *gateways* e a abstração dos detalhes do protocolo de comunicação periférica. E sinalizaram como trabalho futuro desenvolver um ambiente de laboratório inteligente com diversos dispositivos mais complexos e cenários de controle para testar completamente o sistema.

FAN *et al*. (2016) [3], apresentaram a capacidade do recurso de *Web Caching* adicionado ao dispositivo de borda para servir como servidor *proxy* de armazenamento em cache, para obter mais armazenamento em cache. Os dispositivos finais também são explorados para fornecer algum espaço de cache. Como trabalho futuro sinalizaram adicionar mais funcionalidades ao dispositivo de borda, como a segurança e implementar o trabalho no mundo real.

No trabalho de HAO *et al.* (2017) [5], apresentaram um design do WM-FOG, uma estrutura de computação para ambientes *Fog*, que engloba essa arquitetura de *software* e avalia seu protótipo do sistema. Como trabalho futuro sinalizaram adicionar mais recursos ao WM-FOG para melhor atender às aplicações de computação em *Fog*.

Em [8] LI *et al.* (2017), discutiram dois conceitos de codificações recentemente propostos, códigos de mínima largura de banda e códigos de mínima latência, e ilustram seus impactos na computação em *Fog*, também analisaram uma estrutura de codificação unificada que inclui as duas técnicas de codificação acima descritas. Como trabalho futuro sinalizaram a necessidade de pesquisar computação heterogênea; redes com topologia de camada múltipla e estruturada; tarefas de computação em várias etapas; custos de computação codificados; verificar a computação distribuída; explorar as estruturas algébricas das tarefas computacionais; aplicações pesadas de comunicação e nós de *Fog plug-and-play*.

OSANAIYE *et al.* (2017) [11], apresentaram uma abordagem conceitual de migração em tempo real de pré cópia para a migração de VM e sinalizou como trabalho futuro a implantação do *framework* no mundo real ou ambiente de teste, com o objetivo de sua validação.

E por fim, POPENTIU-VLADICESCU et al. (2017) [12], analisaram os modelos de arquiteturas e práticas existentes em *Fog Computing* visando a confiabilidade e segurança dos sistemas de *Fog*, uma abordagem considerada integradora de três componentes da confiabilidade do sistema: a confiabilidade dos nós, a confiabilidade da rede e a confiabilidade do *software*, a arquitetura de referência *OpenFog* e os esquemas AJIA e BDSC. Como trabalho futuro sinalizaram resolver problemas técnicos e de algoritmos no paradigma de *Fog Computing*.

No quadro 1 os dados dos artigos analisados são sumarizados e comparados com este trabalho, foram organizados na ordem crescente cronologicamente para demostrar a

evolução em relação ao tema *Fog Computing*. Os artigos foram comparados quanto a implementação da *Fog Computing*, a utilização de plataforma de *Cloud*, utilização de dispositivo embarcado, uso de técnicas ou métodos de avaliação (*Benchmark*) e a implementação do StaaS.

Tabela 1 - Comparação Entre os Trabalhos

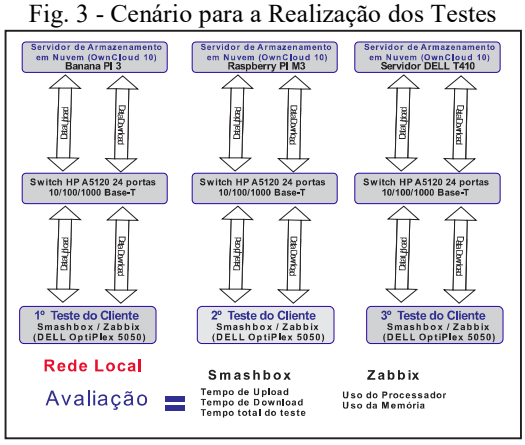
| Artigo | Fog Computing | Plataforma Cloud | Dispositivo Embarcado | Benchmark | StaaS |
|---|---|---|---|---|---|
| **[16]** ZHU (2013) | ✓ | | | | |
| **[2]** CRACIUNESCU (2015) | ✓ | ✓ | ✓ | | |
| **[15]** VERBA (2016) | ✓ | ✓ | ✓ | ✓ | |
| **[3]** FAN (2016) | ✓ | | | ✓ | |
| **[5]** HAO (2017) | ✓ | ✓ | | ✓ | |
| **[8]** LI (2017) | ✓ | | | ✓ | |
| **[11]** OSANAIYE (2017) | ✓ | | | | |
| **[12]** POPENTIU-VLADICESCU (2017) | ✓ | | | | |
| ESTE TRABALHO | ✓ | ✓ | ✓ | ✓ | ✓ |

Fonte: Própria do Autor (2018)

## 4. Coleta de Dados

Para realizar a prototipação deve-se pressupor a existência de um modelo computacional idêntico ao ambiente real de produção. Na literatura, uma boa descrição para análise de desempenho em sistemas de nuvem para fornecer serviço de armazenamento StaaS (*Storage as a Service*) foi encontrada, como exemplo é possível citar o trabalho de MRÓWCZYŃSKI *et al.* (2017) [10]. A figura 3 ilustra a arquitetura do cenário para a realização dos testes de avaliação.

Fig. 3 - Cenário para a Realização dos Testes




Fonte: Própria do Autor (2018)

Tabela 2 - Abreviaturas dos Testes

| Nome | Abreviação | Quantidade de arquivos | Tamanho arquivo | Volume total |
|---|---|---|---|---|
| Test0 | 0/1/1 | 1 | 1B | 1B |
| Test1 | 0/1/100000000 | 1 | 100Mb | 100Mb |
| Test2 | 0/10/10000000 | 10 | 10Mb | 100Mb |
| Test3 | 0/1000/10000 | 1000 | 10Kb | 10Mb |
| Test4 | 0/1/500000000 | 1 | 500Mb | 500Mb |

Fonte: Própria do Autor (2018)

Foram realizados cinco tipos de diferentes testes para avaliar o desempenho dos equipamentos analisados, para o fornecimento do serviço StaaS de forma sequencialmente. A tabela 2 informa abreviaturas dos testes (onde-se, 1º número equivale a quantidade de diretório, 2º quantidade de arquivo e o 3º volume do arquivo) e sua correspondente distribuição de arquivos.

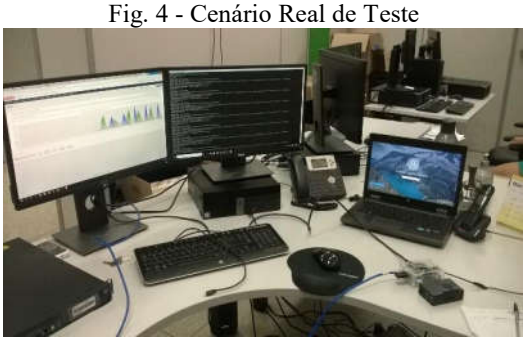
### 4.1 Benchmark Smashbox

O Smashbox é uma estrutura de *benchmarking* e monitoramento para sincronização de arquivos e serviços de compartilhamento, permitindo aos provedores de serviços monitorar o status operacional de seus serviços, entendendo o comportamento do serviço sob diferentes tipos de carga e com diferentes locais de rede para a sincronização de clientes [10]. O *container* do sistema Smashbox está disponibilizado na condição pública no Docker Hub, no endereço https://hub.docker. com/r/ jsmac/smashbox/.

## 5. Avaliação e Resultados

Essa etapa consiste em: realizar a montagem do cenário de teste com os dispositivos de sistemas embarcados e o servidor de forma individual; efetuar a abordagem dos *softwares* necessários nos dispositivos para a elaboração do experimento; proceder com o processo de coleta dos dados do tempo de *upload*, tempo de *download* e tempo total de cada teste realizado com o *benchmark* Smashbox, as métricas de utilização da CPU e memória, equivalem ao máximo obtidos nas 30 repetições dos testes. A figura 4 ilustra o cenário real em que os testes foram realizados.

Fig. 4 - Cenário Real de Teste



Fonte: Própria do Autor (2018)

Tabela 3 - Diferentes Implementações

| Equipamentos utilizados na avaliação | | | | |
|---|---|---|---|---|
| **EQUIP.** | **S.O** | **VIRTUAL** | **RAM** | **REDE** |
| Banana PI M3 | Ubuntu Server 16.04 | Não suporta | 2 Gb | 1000 Mb/s |
| Raspberry PI 3 | Raspbian Stretch Lite 9 | S/ virtualização | 1 Gb | 100 Mb/s |
| Raspberry PI 3 | Raspbian Stretch Lite 9 | Docker | 1 Gb | 100 Mb/s |
| Raspberry PI 3 | Ubuntu MATE 16.04 | Docker | 1 Gb | 100 Mb/s |
| Servidor Dell T410 | Ubuntu Server 16.04 | VMware ESXi + Docker | 16 Gb | 1000 Mb/s |
| Desktop Dell OptiPlex 5050 | Windows 10 | Docker | 16 Gb | 1000 Mb/s |

Fonte: Própria do Autor (2018)

Foram implementados 5 (cinco) cenários com diferentes sistemas operacionais, com e sem a implementação da virtualização Docker *Container*, isso possibilitou analisar o melhor conjunto implementado e o impacto da virtualização em dispositivos de sistemas embarcados. A tabela 3 mostra as diferentes implementações utilizadas com diferentes sistemas operacionais.

Totalizaram-se 750 coletas de dados, 30 repetições para os 5 diferentes testes, utilizando as 5 diferentes implementações entre os equipamentos testados e diferentes sistemas operacionais. Só foram aceitos para análise os testes concluídos sem erros. O tempo encontra-se em segundos. A tabela 4 mostra os resultados obtidos por todas as implementações para o Test0 na transferência de 1 arquivo do tamanho de 1 byte.

### 5.1 Test0 0/1/1

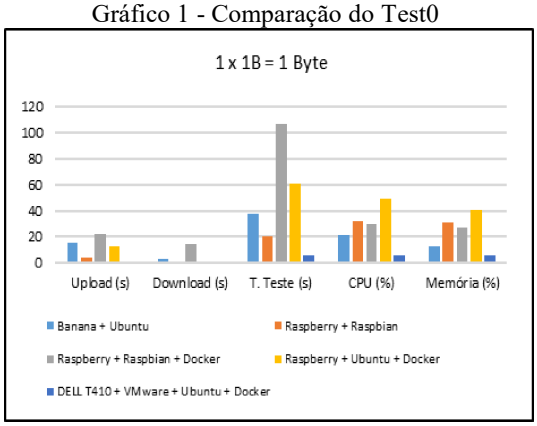
Nota-se que nesse teste todas as implementações obtiveram resultados melhores no tempo de *download* em relação ao tempo de *upload*. Também se observa que as implementações nos dispositivos de sistemas embarcados obtiveram um valor muito elevado na métrica de desvio padrão, o valor. O gráfico 1 mostra melhor visualmente a comparação entre todas as implementações.

Tabela 4 – Resultado do Test0

| Tempo segundos | SMASHBOX | | | | ZABBIX | |
|---|---|---|---|---|---|---|
| | MÉDIA | DESVIO | MÍN | MAX | CPU % | MEM % |
| **BANANA PI M3 + UBUNTU SERVER** | | | | | | |
| T. UPL | 15,13 | 5,54 | 3,00 | 26,00 | 21,50 | 12,50 |
| T. DWL | 3,33 | 2,01 | 2,00 | 9,00 | | |
| TOTAL | 38,23 | 8,65 | 21,00 | 57,00 | | |
| **RASPBERRY PI 3 + RASPBIAN STRETCH LITE** | | | | | | |
| T. UPL | 4,20 | 5,26 | 1,00 | 17,00 | 32,42 | 31,39 |
| T. DWL | 1,73 | 2,30 | 0,82 | 13,00 | | |
| TOTAL | 20,40 | 22,23 | 6,00 | 62,00 | | |
| **RASPBERRY PI 3 + RASPBIAN LITE + DOCKER** | | | | | | |
| T. UPL | 22,60 | 3,76 | 14,00 | 31,00 | 29,36 | 26,55 |
| T. DWL | 14,53 | 2,61 | 7,00 | 20,00 | | |
| TOTAL | 106,60 | 10,41 | 73,00 | 122,00 | | |
| **RASPBERRY PI 3 + UBUNTU MATE + DOCKER** | | | | | | |
| T. UPL | 12,83 | 1,64 | 9,00 | 17,00 | 49,93 | 41,09 |
| T. DWL | 7,03 | 1,27 | 4,00 | 11,00 | | |
| TOTAL | 60,87 | 2,15 | 56,00 | 64,00 | | |
| **DELL T410 + VMWARE + UBUNTU + DOCKER** | | | | | | |
| T. UPL | 1,00 | 0,00 | 1,00 | 1,00 | 6,40 | 6,00 |
| T. DWL | 0,99 | 0,02 | 0,90 | 1,00 | | |
| TOTAL | 6,00 | 0,26 | 5,00 | 7,00 | | |

Fonte: Própria do Autor (2018)

Gráfico 1 - Comparação do Test0




Fonte: Própria do Autor (2018)

A implementação do servidor DELL T410 é considerada a ideal, afinal é um equipamento de alto desempenho, porém com um alto custo aquisitivo, percebe-se que os seus resultados foram melhores em quase todos os aspectos analisados em relação às outras implementações.

## 5.2 Test1 0/1/100000000

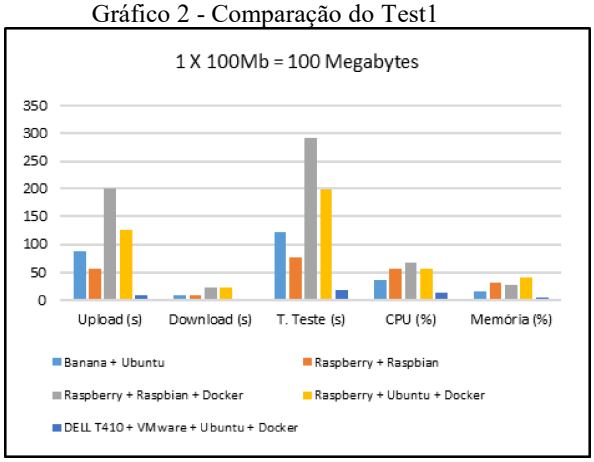
O Test1 consiste em avaliar a transferência do volume de um arquivo com o tamanho de 100 megabytes. A tabela 5 mostra os resultados obtidos por todas as implementações para o Test1 na transferência do arquivo.

Tabela 5 – Resultado do Test1

| Tempo segundos | SMASHBOX | | | | ZABBIX | |
|---|---|---|---|---|---|---|
| | MÉDIA | DESVIO | MÍN | MAX | CPU % | MEM % |
| **BANANA PI M3 + UBUNTU SERVER** | | | | | | |
| T. UPL | 88,50 | 14,45 | 67,00 | 125,00 | 36,31 | 15,50 |
| T. DWL | 10,60 | 4,58 | 6,00 | 27,00 | | |
| TOTAL | 121,63 | 19,24 | 95,00 | 159,00 | | |
| **RASPBERRY PI 3 + RASPBIAN STRETCH LITE** | | | | | | |
| T. UPL | 56,93 | 26,51 | 30,00 | 144,00 | 56,21 | 33,06 |
| T. DWL | 9,70 | 1,78 | 9,00 | 18,00 | | |
| TOTAL | 77,67 | 34,84 | 44,00 | 191,00 | | |
| **RASPBERRY PI 3 + RASPBIAN LITE + DOCKER** | | | | | | |
| T. UPL | 199,20 | 16,61 | 148,00 | 223,00 | 67,60 | 28,76 |
| T. DWL | 23,93 | 5,56 | 14,00 | 42,00 | | |
| TOTAL | 290,90 | 25,80 | 174,00 | 317,00 | | |
| **RASPBERRY PI 3 + UBUNTU MATE + DOCKER** | | | | | | |
| T. UPL | 125,13 | 7,29 | 117,00 | 153,00 | 55,96 | 40,43 |
| T. DWL | 23,57 | 4,28 | 17,00 | 32,00 | | |
| TOTAL | 197,13 | 9,67 | 184,00 | 237,00 | | |
| **DELL T410 + VMWARE + UBUNTU + DOCKER** | | | | | | |
| T. UPL | 9,23 | 0,43 | 9,00 | 10,00 | 14,03 | 6,00 |
| T. DWL | 4,00 | 0 | 4,00 | 4,00 | | |
| TOTAL | 18,87 | 0,35 | 18,00 | 19,00 | | |

Fonte: Própria do Autor (2018)

Gráfico 2 - Comparação do Test1




Fonte: Própria do Autor (2018)

A implementação Banana + Ubuntu obteve uma métrica melhor no quesito tempo médio de *upload* comparado a implementação do Raspberry com ubuntu e utilizando virtualização Docker, fato que não tinha ocorrido no teste Test0. Porém, seus resultados ficaram abaixo comparado com a implementação do Raspberry com raspbian sem virtualização. O gráfico 2 mostra melhor visualmente a comparação.

### 5.3 Test2 0/10/10000000

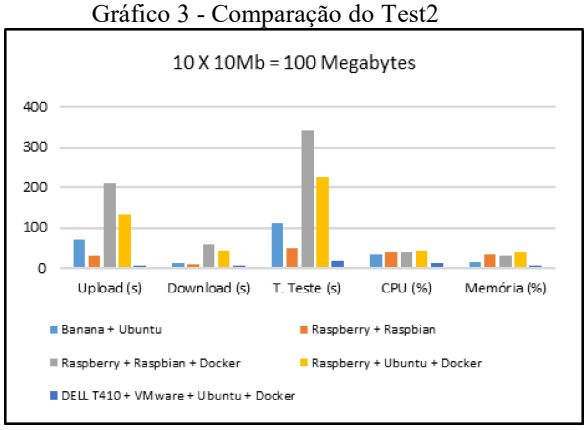
O test2 consiste em avaliar a transferência do volume de dez arquivos com o tamanho de 10 megabytes. A tabela 6 mostra os resultados obtidos por todas as implementações para o Test2 na transferência dos arquivos.

Tabela 6 – Resultado do Test2

| Tempo segundos | SMASHBOX | | | | ZABBIX | |
|---|---|---|---|---|---|---|
| | MÉDIA | DESVIO | MÍN | MAX | CPU % | MEM % |
| **BANANA PI M3 + UBUNTU SERVER** | | | | | | |
| T. UPL | 70,83 | 13,28 | 49,00 | 111,00 | 34,91 | 15,50 |
| T. DWL | 11,77 | 7,54 | 5,00 | 32,00 | | |
| TOTAL | 111,17 | 18,20 | 85,00 | 147,00 | | |
| **RASPBERRY PI 3 + RASPBIAN STRETCH LITE** | | | | | | |
| T. UPL | 30,63 | 29,74 | 16,00 | 158,00 | 40,88 | 33,85 |
| T. DWL | 10,20 | 3,92 | 9,00 | 30,00 | | |
| TOTAL | 49,83 | 34,27 | 33,00 | 199,00 | | |
| **RASPBERRY PI 3 + RASPBIAN LITE + DOCKER** | | | | | | |
| T. UPL | 209,63 | 14,72 | 184,00 | 236,00 | 41,13 | 30,37 |
| T. DWL | 59,37 | 8,273 | 50,00 | 95,00 | | |
| TOTAL | 341,77 | 19,49 | 307,00 | 414,00 | | |
| **RASPBERRY PI 3 + UBUNTU MATE + DOCKER** | | | | | | |
| T. UPL | 133,37 | 21,92 | 117,00 | 242,00 | 41,96 | 39,50 |
| T. DWL | 42,40 | 7,96 | 34,00 | 80,00 | | |
| TOTAL | 225,70 | 21,93 | 212,00 | 330,00 | | |
| **DELL T410 + VMWARE + UBUNTU + DOCKER** | | | | | | |
| T. UPL | 7,40 | 0,89 | 6,00 | 9,00 | 13,20 | 6,00 |
| T. DWL | 4,80 | 0,61 | 4,00 | 6,00 | | |
| TOTAL | 18,73 | 1,08 | 17,00 | 21,00 | | |

Fonte: Própria do Autor (2018)

Gráfico 3 - Comparação do Test2




Fonte: Própria do Autor (2018)

Na implementação Raspberry + Raspbian + Docker, nota-se que continuou obtendo os piores resultados no teste, com a média 209,63 segundos de *upload* e 59,37 segundos de *download*, e uma média de tempo de conclusão dos testes muito elevada 341,77 segundos. Percebe-se também que a implementação chegou a utilizar um nível razoável de processamento no teste 41,13% e com o uso de memória mediano 30,37%. O gráfico 3 mostra melhor visualmente a comparação.

### 5.4 Test3 0/1000/10000

O test3 consiste em avaliar a transferência do volume de mil arquivos com o tamanho de 10 kilobytes, acredita-se que esse teste seja a simulação mais próxima da realidade do funcionamento de uma *Fog Computing*, pequenos arquivos, porém em grande quantidade. A tabela 7 mostra os resultados obtidos por todas as implementações.

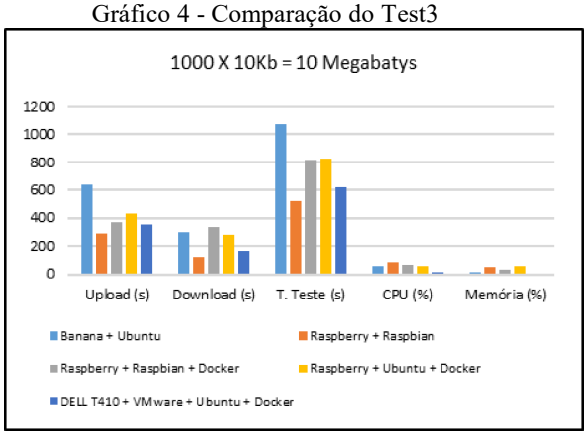
A implementação Raspberry + Raspbian surpreendeu, superando até mesmo a implementação do servidor DELL T410. Todavia nota-se que o desvio padrão dos seus resultados apresentaram uma alta discrepância, isso o torna um serviço pouco preciso. O gráfico 4 mostra melhor visualmente a comparação.

Tabela 7 – Resultado do Test3

| Tempo segundos | SMASHBOX | | | | ZABBIX | |
|---|---|---|---|---|---|---|
| | MÉDIA | DESVIO | MÍN | MAX | CPU % | MEM % |
| **BANANA PI M3 + UBUNTU SERVER** | | | | | | |
| T. UPL | 640,67 | 459,94 | 418,00 | 2496,00 | 59,67 | 18,00 |
| T. DWL | 301,67 | 121,19 | 232,00 | 687,00 | | |
| TOTAL | 1079,03 | 540,16 | 804,00 | 2889,00 | | |
| **RASPBERRY PI 3 + RASPBIAN STRETCH LITE** | | | | | | |
| T. UPL | 289,40 | 174,19 | 148,00 | 746,00 | 87,21 | 46,56 |
| T. DWL | 120,77 | 98,14 | 87,00 | 546,00 | | |
| TOTAL | 521,50 | 212,90 | 346,00 | 1175,00 | | |
| **RASPBERRY PI 3 + RASPBIAN LITE + DOCKER** | | | | | | |
| T. UPL | 369,03 | 46,41 | 346,00 | 610,00 | 67,65 | 31,14 |
| T. DWL | 334,43 | 25,82 | 327,00 | 469,00 | | |
| TOTAL | 812,60 | 56,57 | 779,00 | 1080,00 | | |
| **RASPBERRY PI 3 + UBUNTU MATE + DOCKER** | | | | | | |
| T. UPL | 430,57 | 81,83 | 335,00 | 685,00 | 61,65 | 60,82 |
| T. DWL | 278,13 | 210,58 | 106,00 | 1056,00 | | |
| TOTAL | 821,17 | 292,95 | 553,00 | 1858,00 | | |
| **DELL T410 + VMWARE + UBUNTU + DOCKER** | | | | | | |
| T. UPL | 350,63 | 6,14 | 340,00 | 364,00 | 12,00 | 7,08 |
| T. DWL | 163,93 | 0,25 | 163,00 | 164,00 | | |
| TOTAL | 620,63 | 6,36 | 610,00 | 635,00 | | |

Fonte: Própria do Autor (2018)

Gráfico 4 - Comparação do Test3




Fonte: Própria do Autor (2018)

## 5.5 Test4 0/1/500000000

O Test4 consiste em avaliar a transferência do volume de um arquivo com o tamanho de 500 megabytes. A tabela 8 mostra os resultados obtidos por todas as implementações.

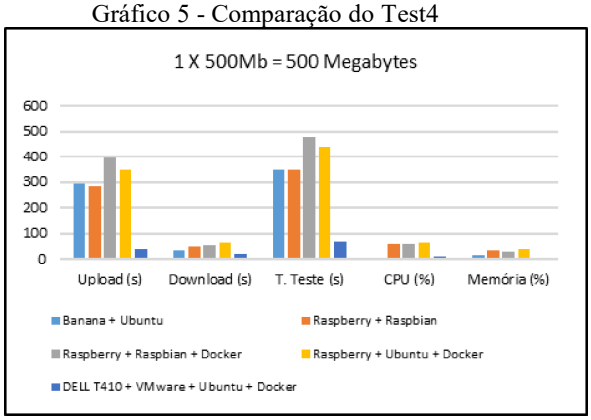
Na implementação Raspberry + Ubuntu + Docker obteve bons resultados quando comparado com a implementação de concorrência direta, Raspberry + Raspbian + Docker. Todavia nota-se que foi a implementação que obteve o mais alto nível do uso da memória do *hardware* 65,00%. O gráfico 5 mostra melhor visualmente a comparação.

Tabela 8 – Resultado do Test4

| Tempo segundos | SMASHBOX | | | | ZABBIX | |
|---|---|---|---|---|---|---|
| | MÉDIA | DESVIO | MÍN | MAX | CPU % | MEM % |
| **BANANA PI M3 + UBUNTU SERVER** | | | | | | |
| T. UPL | 294,00 | 24,62 | 237,00 | 340,00 | 40,92 | 17,00 |
| T. DWL | 36,33 | 12,69 | 21,00 | 81,00 | | |
| TOTAL | 347,47 | 31,03 | 271,00 | 394,00 | | |
| **RASPBERRY PI 3 + RASPBIAN STRETCH LITE** | | | | | | |
| T. UPL | 281,83 | 84,91 | 174,00 | 449,00 | 57,05 | 33,96 |
| T. DWL | 49,40 | 5,93 | 44,00 | 69,00 | | |
| TOTAL | 345,43 | 94,24 | 226,00 | 544,00 | | |
| **RASPBERRY PI 3 + RASPBIAN LITE + DOCKER** | | | | | | |
| T. UPL | 394,37 | 146,64 | 191,00 | 886,00 | 59,41 | 30,30 |
| T. DWL | 55,30 | 8,26 | 47,00 | 81,00 | | |
| TOTAL | 476,80 | 158,12 | 254,00 | 1018,00 | | |
| **RASPBERRY PI 3 + UBUNTU MATE + DOCKER** | | | | | | |
| T. UPL | 346,30 | 168,15 | 190,00 | 850,00 | 65,00 | 38,95 |
| T. DWL | 63,97 | 25,32 | 48,00 | 161,00 | | |
| TOTAL | 437,03 | 188,62 | 258,00 | 961,00 | | |
| **DELL T410 + VMWARE + UBUNTU + DOCKER** | | | | | | |
| T. UPL | 40,30 | 3,48 | 37,00 | 51,00 | 12,00 | 7,08 |
| T. DWL | 19,23 | 3,08 | 16,00 | 27,00 | | |
| TOTAL | 67,47 | 5,96 | 61,00 | 80,00 | | |

Fonte: Própria do Autor (2018)

Gráfico 5 - Comparação do Test4




Fonte: Própria do Autor (2018)

Diferentes aplicações de computação em *Fog* foram sugeridas na literatura. Segundo OSANAIYE *et al.* (2017) [11], as categorias das aplicações de computação em *Fog* são divididas em 3: (i) Aplicações em tempo real; (ii) Aplicações quase em tempo real; (iii) Aplicações introduzidas em redes.

Por isso, definir o nível aceitável do fornecimento do serviço de armazenamento StaaS, em uma emergente tecnologia em que se trata o ambiente *Fog Computing* é uma tarefa complexa e subjetiva, dependente da classificação da aplicação.

## 6. Conclusão e Trabalhos Futuros

Este trabalho apresenta o conceito da *Fog Computing*, sua contextualização teórica, os trabalhos correlatos, realiza análise de uma *Fog Computing*, para fornecer StaaS (*Storage as a Service*), a dispositivos IoT utilizando plataformas de sistemas embarcados e compara seus resultados com os obtidos por um servidor de alto desempenho. Foram realizados cinco (5) implementações de diferentes combinações de *softwares* e *hardwares*, e analisa seus resultados com a finalidade de encontrar a melhor opção para disponibilizar o serviço de armazenamento StaaS em um ambiente *Fog Computing*.

Os resultados satisfizeram as expectativas, na avaliação do teste Test3 0/1000/10000 (transferência de 1000 arquivos de 10 *Kilobytes*) a implementação Raspberry + Raspbian surpreendeu, obtendo ótimos resultados, superando até mesmo a implementação do servidor DELL T410. Nota-se que este tipo de implementação pode satisfazer as aplicações classificadas em aplicações quase em tempo real e introduzida na rede, não sendo adequada para as aplicações classificadas como aplicações em tempo real, devido as implementações nos dispositivos de sistemas embarcados obterem valores muito elevados nas métricas de desvio padrão, tornando o serviço pouco preciso.

Portanto percebe-se que a implementação desse serviço em dispositivos de sistemas embarcados pode ser uma boa alternativa para reduzir um desses problemas, no caso, o armazenamento de dados, que atinge hoje os dispositivos IoT, servindo como *Fog Computing* e sendo implantado num dispositivo de plataforma embarcada de baixo custo, ao invés de usar potentes e caros servidores para exercer essa função.

### 6.1 Trabalhos Futuros

Com a finalidade em dar continuidade a essa pesquisa, acredita-se que esse trabalho abriu vários cenários para futuros trabalhos, sendo:

- Analisar o serviço StaaS, com diferentes dispositivos embarcados e diferentes plataformas de serviço de armazenamento em nuvem, não utilizados neste trabalho;
- Realizar a mesma avaliação utilizando *cluster* com dispositivos de sistemas embarcados, usando a virtualização Dorcker.

## 7. Referências

[1] AL-DOGHMAN, Firas et al. A review on Fog Computing technology. In: Systems, Man, and Cybernetics (SMC), 2016 IEEE International Conference on. IEEE, 2016. p. 001525-001530.

[2] CRACIUNESCU, Razvan et al. Implementation of Fog computing for reliable E-health applications. In: Signals, Systems and Computers, 2015 49th Asilomar Conference on. IEEE, 2015. p. 459-463.

[3] FAN, Chih-Tien et al. Web Resource Cacheable Edge Device in Fog Computing. In: Parallel and Distributed Computing (ISPDC), 2016 15th International Symposium on. IEEE, 2016. p. 432-439.

[4] HAJIBABA, Majid; GORGIN, Saeid. A review on modern distributed computing paradigms: Cloud computing, jungle computing and fog computing. CIT. Journal of Computing and Information Technology, v. 22, n. 2, p. 69-84, 2014.

[5] HAO, Zijiang et al. Challenges and Software Architecture for Fog Computing. IEEE Internet Computing, v. 21, n. 2, p. 44-53, 2017.

[6] JAIN, Akshay; SINGHAL, Priyank. Fog computing: Driving force behind the emergence of edge computing. In: System Modeling & Advancement in Research Trends (SMART), International Conference. IEEE, 2016. p. 294-297.

[7] MACHADO, José dos Santos; MORENO, Edward David; RIBEIRO, Admilson de Ribamar Lima. A Review of Computing Fog and its Research Challenges. Journal on Advances in Theoretical and Applied Informatics, [S.l.], v. 3, n. 2, p. 32-39, dec. 2017. ISSN 2447-5033.

[8] LI, Songze; MADDAH-ALI, Mohammad Ali; AVESTIMEHR, A. Salman. Coding for Distributed Fog Computing. IEEE Communications Magazine, v. 55, n. 4, p. 34-40, 2017.

[9] MARTINI, Ben; CHOO, Kim-Kwang Raymond. Cloud storage forensics: ownCloud as a case study. Digital Investigation, v. 10, n. 4, p. 287-299, 2013.

[10] MRÓWCZYŃSKI, Piotr et al. Benchmarking and monitoring framework for interconnected file synchronization and sharing services. Future Generation Computer Systems, 2017.

[11] OSANAIYE, Opeyemi et al. From cloud to fog computing: A review and a conceptual live VM migration framework. IEEE Access, 2017.

[12] POPENTIU-VLADICESCU, Florin; ALBEANU, Grigore. Software reliability in the fog computing. In: Innovations in Electrical Engineering and Computational Technologies (ICIEECT), 2017 International Conference on. IEEE, 2017. p. 1-4.

[13] MCMILLIN, Bruce; ZHANG, Tao. Fog Computing for Smart Living. Computer, v. 50, n. 2, p. 5-5, 2017.

[14] STOJMENOVIC, Ivan et al. An overview of Fog computing and its security issues. Concurrency and Computation: Practice and Experience, 2015.

[15] VERBA, Nandor et al. Platform as a service gateway for the Fog of Things. Advanced Engineering Informatics, 2016.

[16] ZHU, Jiang et al. Improving web sites performance using edge servers in fog computing architecture. In: Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on. IEEE, 2013. p. 320-323.

[17] ALRAWAIS, Arwa et al. Fog Computing for the Internet of Things: Security and Privacy Issues. IEEE Internet Computing, v. 21, n. 2, p. 34-42, 2017.

[18] BABU, Shaik Masthan; LAKSHMI, A. Jaya; RAO, B. Thirumala. A study on cloud based internet of things: Cloudiot. In: Communication Technologies (GCCT), 2015 Global Conference on. IEEE, 2015. p. 60-65.

[19] BOTTA, Alessio et al. Integration of cloud computing and internet of things: a survey. Future Generation Computer Systems, v. 56, p. 684-700, 2016.

[20] CHEN, Songqing; ZHANG, Tao; SHI, Weisong. Fog Computing. IEEE Internet Computing, v. 21, n. 2, p. 4-6, 2017.

[21] CHIANG, Mung et al. Clarifying Fog Computing and Networking: 10 Questions and Answers. IEEE Communications Magazine, v. 55, n. 4, p. 18-20, 2017.

[22] DÍAZ, Manuel; MARTÍN, Cristian; RUBIO, Bartolomé. State-of-the-art, challenges, and open issues in the integration of Internet of things and cloud computing. Journal of Network and Computer Applications, v. 67, p. 99-117, 2016.

[23] KUMAR, Praveen; ZAIDI, Nabeel; CHOUDHURY, Tanupriya. Fog computing: Common security issues and proposed countermeasures. In: System Modeling & Advancement in Research Trends (SMART), International Conference. IEEE, 2016. p. 311-315.

[24] LINTHICUM, David S. Connecting Fog and Cloud Computing. IEEE Cloud Computing, v. 4, n. 2, p. 18-20, 2017.

[25] NWOBODO, Ikechukwu (2015). A Comparison of Cloud Computing Platforms. International Symposium on Circuits and Systems (ISCAS). Lisbon, Portugal, 24-27 May 2015. Atlantis Press.

[26] PRINCY, S. Emima; NIGEL, K. Gerard Joe. Implementation of cloud server for real time data storage using Raspberry Pi. In: Green Engineering and Technologies (IC-GET), 2015 Online International Conference on. IEEE, 2015. p. 1-4.

[27] LONGO, Francesco et al. Stack4things: An openstack-based framework for iot. In: Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on. IEEE, 2015. p. 204-211.

[28] ZHANG, Qi; CHENG, Lu; BOUTABA, Raouf. Cloud computing: state-of-the-art and research challenges. Journal of internet services and applications, v. 1, n. 1, p. 7-18, 2010.

# A Fast and Generic GPU-Based Parallel Reduction Implementation

**Walid A. R. Jradi**[1], **Hugo A. D. do Nascimento**[1], **Wellington S. Martins**[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Alameda Palmeiras, Quadra D, Campus Samambaia
CEP 74690-900 – Goiânia – Goiás – Brazil

walid.jradi@gmail.com,{hadn,wellington}@inf.ufg.br

***Abstract.*** *Reduction operations are extensively employed in many computational problems, where a finite set of numeric elements are combined into a single value, using for this a combining function. A parallel reduction, in turn, is the operation concurrently performed when multiple execution units are available. The present work depicts a GPU-based parallel approach for it, which employs techniques like loop unrolling, persistent threads and algebraic expressions to avoid thread divergence, and was able to outperform the methods currently in use. Experiments conducted to evaluate the approach show that the strategy performs efficiently on both AMD and NVidia's hardwares, as well as using OpenCL and CUDA, making it portable.*

## 1. Introduction

Reduction is a basic step in many algorithms, including classical ones such as Stream Compaction [Billeter et al. 2009], Golden Section and Fibonacci Methods [Kiefer 1953] and Count and Radix Sort [Cormen et al. 2002]. A *reduction operation* can be formally defined as follows [Parhami 1999]: given a set *X* of *n* elements, $X = \{x_0, x_1, ..., x_{n-1}\}$, compute $x_0 \otimes x_1 \otimes ... \otimes x_{n-1}$, with $\otimes$ (also known as a *combining function*) being an associative and commutative operator. Examples of $\otimes$ when $X$ is composed of numbers are the operators $+, \times, \wedge, \vee, \oplus, \cap, \cup, \max$ and $\min$. Algorithm 1 illustrates a reduction process through the summation of a set of values.

**Algorithm 1:** *Summation(X)*

**Input:** Set $X = \{x_1, x_2, \ldots, x_n\}$ of numeric elements
**Output:** The sum of all elements

```
1 accumulator ← 0
2 for i ← 1 to n do
3 |   accumulator ← accumulator + x_i
4 return accumulator
```

A parallel version of a reduction consists of computing the $\otimes$ operator concurrently for the set of elements using multiple execution units. At a first glance of Algorithm 1, it may seem that the reduction operator sum (+) is inherently sequential since the variable *accumulator* depends on values computed in previous steps. However, it is possible to parallelize that operator because of its associative property. In fact, the associativity of $\otimes$, sometimes combined with the commutative property, allows to divide the

calculations hierarchically into subproblems that can be solved in parallel and then having their partial computations combined to produce the final result. The order in which the partial reductions are combined does not affect the final result[1].

Since the arrival of programmable GPUs, some strategies to accelerate the reduction operation on such devices appeared. Two famous ones are described by Mark Harris [Harris 2007] and Bryan Catanzaro [Catanzaro 2014]. Most recently, Justin Luitjens [Luitjens 2014] presented some improvements to the strategies described in [Harris 2007]. Unfortunately, the approaches adopted by [Harris 2007] and [Luitjens 2014], although very efficient, are limited to hardware and software provided by NVidia. On the other hand, the proposal of [Catanzaro 2014] is based on the open standard OpenCL [Group et al. 2008], adopted by a myriad of manufacturers, which makes it portable. Nevertheless, the code presented in [Catanzaro 2014] also has a weakness, as it does not adopt some strategies that could significantly improve its performance.

In the present paper, we describe a new parallel implementation for reduction operators that provides both portability and very good performances. It adequately combines strategies that were used in the approaches mentioned above.

The remainder of this paper is organized as follows: Section 2 briefly describes the existing techniques for parallel reduction in GPUs; Section 3 explains our approach; Section 4 details the experiments performed for testing the approach; Finally, Section 5 draws our conclusions and points to future investigations in this area.

## 2. Parallel Reduction in GPUs

As explained before, the basic idea for parallelizing a reduction is to "split" the problem into smaller pieces and to solve them in parallel. The GPU hardware, however, has features which impose some restrictions that must be considered in order to maximize the *speedup* of the parallel code [Wilt 2013].

The approaches of [Harris 2007] and [Catanzaro 2014] to deal with reductions in GPUs operate in a very similar way, using a tree-based structure.

One of the aspects to be considered is the number of elements to be reduced. If this amount is sufficiently small and can be stored in the local memory of each SM (*Symmetric Multiprocessor*), then the reduction becomes simple. In [Catanzaro 2014], the author presents some strategies for this case and conducts performance comparisons between them. After describing how reductions can be efficiently performed in small sets, Catanzaro shifts the focus to the discussion of cases in which a large volume of data must be handled. Three strategies are presented and a winner, called "*Two-Stage Parallel Reduction*", is elected. [Harris 2007] deals only with reduction of large datasets.

Our approach is mainly based on a proposal from [Catanzaro 2014]. Therefore, a more detailed description of it is presented. First, however, we explain the strategies

[1]Although, mathematically, this is true for numbers in any set, in computational terms this is more complicated. For instance, that property holds for the set of integers, but the same does not happen for the floating point numbers due to the inherent imprecision that arises when combining (adding, multiplying, etc.) numbers with different exponents, which leads to the absorption of the lower bits during the combine operation. As an example, mathematically the result of $(1.5 + 4^{50} - 4^{50})$ is always the same, no matter the order the terms are added, whereas the floating point computation can result in 0 or 1.5, depending on the sequence in which the operations are performed [Goldberg 1991, Higham 2002, Muller et al. 2009].

employed by [Harris 2007] since some ideas for speeding up the computation come from that work. We also comment about the research of [Luitjens 2014], because it is another way of dealing with the reduction problem.

### 2.1. Mark Harris' Work

The work presented by [Harris 2007] focuses on techniques for performing reductions on large data volumes. The author shows, through successive versions of the same algorithm, how bad decisions or an incorrect way of mapping the problem to the target platform can negatively impact the application performance.

Harris performed experiments using a G80 GPU. That video card has a 384-bit memory interface, with a 900 MHz DDR memory, which leads to a theoretic $384 * 1800/8 = 86.4$ GB/s of memory bandwidth. All tests were conducted using a vector with $2^{22}$ (4M) integer values.

Problems like shared memory bank conflict, lack of communication between thread blocks (making it impossible for a kernel to reduce a large array at once) and highly divergent warps are addressed. Starting with a naive version, step by step improvements are described. Harris tested strategies such as: (1) Replacing conditional commands by new instructions that avoid divergent branching; (2) Keeping all work-items active, doing work as much as possible; (3) Unrolling loops until maximum efficiency is reached; (4) Using CUDA C++ template parameters on device and host functions for the specification of the workgroup size, so that only the necessary amount of work-items are launched; and (5) Allowing each work-item to individually compute as many reduction operations sequentially as possible in order to reduce synchronization between threads, among other benefits. As a result of applying all these optimizations, the final version ran in 0.268 ms (30.04x times faster than its initial version) and reached a memory bandwidth of 62.671 GB/s. In our work, we use all strategies proposed by Harris except the fourth one.

### 2.2. Justin Luitjens' Work

In [Luitjens 2014], Luitjens shows how a feature of the NVidia's Kepler (and newer) GPU architecture, the shuffle (SHFL) instruction, can be used to make reductions even faster when compared to the strategies presented in [Harris 2007].

Usually, work-items inside the same SM use the local memory when they need to communicate. This involves a three-step process: writing the data to the local memory, perform a synchronization barrier and then reading the data back from local memory. The Kepler and newer architectures implement the *shuffle* instruction, which enables a work-item to directly read private data from another work-item in the same wave-front. According to the author, there are four main advantages in using this instruction:

- It ultimately allows work-items inside a wave-front to collectively exchange or broadcast data;
- It replaces the three-step process by a single instruction, effectively increasing the bandwidth and decreasing the latency;
- It does not use the local memory at all;
- A sync barrier is implicit in the instruction and, hence, a synchronization step inside a workgroup is not necessary.

Using this instruction, several versions of the reduction process were proposed, implemented and compared. However, although Luitjens states that the adopted strategies lead to faster reductions than those described by [Harris 2007], no comparative studies between the two approaches were conducted.

### 2.3. Bryan Catanzaro's Work

Now, we describe Catanzaro's two-stage parallel reduction approach for large datasets, as presented in [Catanzaro 2014].

The technique is based on dividing the data set in *p* pieces (or "*chunks*"), where *p* is large enough to keep all GPU cores busy. It is also necessary to limit the number of *work-items* to the maximum amount that the GPU can handle in total without having to switch between them (from now on, that maximum will be called *GS* – or *global size*). Each chunk is then processed by a work-group.

**Listing 1. Two-stage parallel reduction of Catanzaro – stage 1**

```
__kernel void reduce(__global float* buffer,
            __local float* scratch,
            __const int length,
            __global float* result) {

  int global_index = get_global_id(0);
  float accumulator = INFINITY;
  // Loop sequentially over chunks of input vector
  while (global_index < length) {
    float element = buffer[global_index];
    accumulator = (accumulator < element) ? accumulator : element;
    global_index += get_global_size(0);
  }
  int local_index = get_local_id(0);
  scratch[local_index] = accumulator;
  barrier(CLK_LOCAL_MEM_FENCE);
  // Perform parallel reduction
  for(int offset=get_local_size(0)/2; offset>0; offset=offset/2) {
    if (local_index < offset) {
      float other = scratch[local_index + offset];
      float mine = scratch[local_index];
      scratch[local_index] = (mine < other) ? mine : other;
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (local_index == 0) {
    result[get_group_id(0)] = scratch[0];
  }
}
```

Since the sum operation has the properties of associativity and commutativity, each *work-item* can perform its own reduction in parallel. The work-item takes, as the starting point, its global identifier and accumulates its partial sum in a private variable. It reads the data from a vector stored in the GPU's global memory, intercalating the access to the values with the other work-items in a sequential order. Basically, the work-item $i$, $i = 0, 1, GS - 1$, starts reading the position $i$ and skips *GS* positions at every step.

After having completed a pass through the data set, the *work-items* in each workgroup write the result of their own reduction in a scrap vector located in local/shared memory which, in turn, will also be reduced in parallel. At the end of the process, each working group will have its own scrap containing, in its position 0, the result of the reduction so far. This partial result is then copied to another vector, this time stored in the GPU global memory, which size must be equal to $|SM|$. The first stage is then complete. Its source code, extracted from [Catanzaro 2014], is presented in Listing 1.

The second stage is simpler. Since now there is a vector with $|SM|$ elements in the global memory – with the result of a partial sum in each position – just the first $|SM|$ *work-items* of the first SM copy their corresponding value to an array allocated in the local memory. Then, the *work-items* perform a new parallel sum of the elements in the vector. After copying the value in position 0 back to global memory, the reduction is complete.

The overall approach of Catanzaro was the underlying architecture of the present work. It was, however, improved with the extensive usage of the advanced techniques described in the next section, in order to further explore parallelism.

## 2.4. Loop Unrolling

*Loop Unrolling* (also known as *Loop Unwinding* and *Loop Unfolding*) is an optimization technique – performed by the compiler or manually by the programmer – applicable to certain kinds of loops in order to reduce (or even prevent) the occurrence of execution branches and to minimize the cost of instructions for controlling the *loop* [Fog 2013, Huang and Leng 1997, Sarkar 2001]. In general, it optimizes the program's execution speed at the expense of increasing the size of the generated code (*space-time tradeoff*). It is easily applicable to loops where the number of executions is previously known, like routines of vector manipulation where the number of elements is fixed.

Basically the technique consists of reusing the sequence of instructions within the loop, so as to include more of an iteration of the code every time the *loop* is repeated, reducing the amount of these repetitions.

This reuse is done by manually replicating the code inside the *loop* a certain amount of times or through the "*#pragma unroll* ***n***"[2] positioned immediately before the beginning of the loop. The number of times the loop is unrolled is called *Unrolling Factor* and, with the pragma directive, it is given by the parameter "***n***".

It is worth noting that with the pragma directive we leave the decisions of how the loop should be unrolled to the compiler, which may lead to a not so optimized resulting code. In the experiments performed as part of this work, the best results were always achieved using manual loop unrolling.

Unrolling, when applicable, offers several advantages over non-unrolled code. Besides the decrease in the number of iterations, an increase occurs in the amount of work done each time through the loop. This also opens ways for the exploration of parallelism by the compiler in machines with multiple execution units, since each instruction within the *loop* can be handled by an independent thread.

[2] A *directive pragma* is a language construct that provides additional information to the compiler, specifying how to process its input. This additional information usually is beyond what is conveyed in the language itself.

However, these are only the most easily perceivable benefits. Fog [Fog 2013] lists several others, as well as some observations about when this technique should be avoided. Such factors (advantages and disadvantages) must be considered by the programmer when deciding to use loop unrolling or not.

### 2.5. Persistent Threads

All the current programmable GPUs follow the "*Single Instruction Multiple Thread"* (SIMT) and "*Single Program Multiple Data*" (SPMD) paradigms, hiding the details of the underlying *hardware* where the code runs in an attempt to ease the development task.

Some authors [Gupta et al. 2012] argue that the usage of these paradigms greatly limits the actions of the programmer, because all control of the execution flow is in the power of the *scheduler's* video card. They claim that, while these abstractions reduce the effort for new developers in the GPGPU field, they also create obstacles for experienced programmers, who normally face problems for which workload is inherently irregular, therefore making it much more difficult to efficiently parallelize when compared to problems whose parallel solution is more regular.

They argue that this uncovers a serious drawback of the current programming styles, which is not able to ensure *order*, *location* and *timing*. It also does not allow the software developer to regulate these three parameters without completely avoiding the GPU scheduler. Thus, to overcome these limitations, developers have been using a programming style called *Persistent Threads ("PT")*, whose low level of abstraction allows performance gains by directly controlling the scheduling of work-groups.

Basically, what the PT style changes is the *lifetime* of a *work-item* [Nasre et al. 2013], by letting it to keep running longer and giving it much more work than in the traditional "non-PT" style [Steinberger et al. 2012]. This is done by circumscribing the kernel logic (or part of it) in a loop, which remains running while there are items to be processed.

Briefly, from the point of view of the developer, all work-items are active while the kernel is running. As a direct consequence of PT, a *kernel* should be triggered using only the amount of *work-items* that can be executed concurrently by each Streaming Multiprocessor. All these actions will prevent constant return of control to the host and the cost of new kernel invocations [Nasre et al. 2013].

[Gupta et al. 2012] acknowledge, however, that the PT technique is not a panacea, and its use should be carefully evaluated. In particular, the technique fits well when the amount of memory accesses is limited (i.e., few reading/writing to global memory and a large volume of computation) and the problem being solved has not many initial input elements or the growth in the number of elements in the input set is fairly limited. Beyond these conditions, the traditional non-PT style tends to outperform the PT style.

### 2.6. Thread Divergence

Current GPUs are able to deliver massive computational power at a reasonably low cost. However, due to the way they are constructed, some obstacles must be overcome for the effective use of such power. One of the main and hardest obstacles to avoid is the presence of conditional statements [Zhang et al. 2010] potentially leading to branches in the execution flow of the various work-items [Han and Abdelrahman 2011].

By default, GPUs try to run all the work-items inside the wave-fronts in the SIMD model. However, if the code being executed has conditional statements that lead to divergences in program flow, the divergent work-items will be stalled and its execution will only happen after the non-stalled work-items have completed their runs, which ultimately compromises the desired *speedup*. This phenomenon is called *Thread Divergence* [Chakroun et al. 2013, Narasiman et al. 2011].

Trying to circumvent this problem, some strategies have been proposed in order to minimize or even eliminate the effects of such phenomena. Among them, we cite [Chakroun et al. 2013, Fung et al. 2007, Han and Abdelrahman 2011, Meng et al. 2010, Narasiman et al. 2011, Zhang et al. 2010].

Therefore it became necessary to employ a method to prevent flow divergence, which could ultimately compromise the performance of such a step of computation. The adopted method is detailed at the end of Section 3.

## 3. The New Approach

The improvements proposed in our work focus on Steps 1 and 3 of the first stage of the reduction presented in Section 2.3. The improvements employ the same strategies proposed by Harris [Harris 2007] to increase the performance of the approach originally presented by Catanzaro [Catanzaro 2014] but with appropriately chosen interventions.

In step 1 of the original implementation, the vector in global memory containing the data to be reduced is entirely traversed by the *work-items*, each one performing its own reduction.

This step already uses the "Persistent-Thread" strategy, but its performance can be improved by adopting loop unrolling (Section 2.4). As it can be seen, instead of doing the unroll when the data is in local memory, as proposed by [Harris 2007], our improvement performs the unroll in the global memory.

The code presented in Listing 2 shows the modified loop, assuming an unrolling factor (F) equals to 4, $iGlobalID$ as the *work-item* global identifier and $iLength$ as the number of elements to be reduced.

**Listing 2. Unrolling the step 1**

```
for (iPos = iGlobalID*iUnrollingFactor; iPos < iLength;
     iPos += iGlobalSize*iUnrollingFactor)
{
 i0 = iPos;   i1 = iPos+1; i2 = iPos+2; i3 = iPos+3;
 accumulator +=
 ((i0<iLength)*(aVector[i0])+
  (i1<iLength)*(aVector[i1])+
  (i2<iLength)*(aVector[i2])+
  (i3<iLength)*(aVector[i3]));
}
```

Special attention must be given to how the data is brought from global (*aVector*) to private memory (*accumulator*), through the use of algebraic expressions that prevent reading from invalid memory locations, thus avoiding the usage of "ifs" and potential

divergences in the execution flow. The expression $i_n < iLength$ expands to integers 1 or 0 whether it is, respectively, true or false. In the first case $(i_n < iLength) * (aVector[i_n])$ is interpreted as $(1) * (aVector[i_n])$, adding the value stored in location $i_n$ to the partial sum (*accumulator*). In the second case, the expression is interpreted as $(0) * (aVector[0])$, ensuring that – regardless of the data stored in the first position of the vector – value 0 is added to *accumulator*, keeping the partial sum correctness.

At the beginning of Step 3, the resulting values of the previous sums are stored in local memory. Then, each SM performs its own local reduction with its work-items.

In the solutions presented by [Harris 2007] and [Catanzaro 2014], in this step all *work-items* are kept synchronized through the use of barriers. However, with minor conceptual changes, it is possible to completely eliminate the overhead caused by the barriers, not only in the last 6 iterations of the loop, as proposed by [Harris 2007].

Our strategy is to use algebraic expressions to keep all the *work-items* in the same execution step, maintaining its desired behavior and algorithm correctness, as can be seen in Listing 3. Here, the conditional statement is completely eliminated but yet the function returns the right result of the comparison.

**Listing 3. Algebraic "if-then-else"**

```
int smallestValue(int a, int b)
{ return (a < b) * a + (a >= b) * b; }
```

Note that the two boolean operations ($(a < b)$ and $(a >= b)$) are mutually exclusive, being interpreted internally by the compiler as 0 (false) or 1 (true). So, assuming that ***a*** is smaller than ***b***, the result of the algebraic operation is ***(1) * a + (0) * b*** which, ultimately, will return only the value of ***a***.

The same strategy can be applied to lines 18 to 24 of Listing 1, that represent the third step of the first stage. The new code is shown in Listing 4, where $iLocalSize$ stores the number of active local work-items and $iLI$ represents the *work-item's* local identifier.

**Listing 4. Avoiding Divergences**

```
for (iPos = iLocalSize/2; iPos > 0; iPos >>= 1)
{
 bFlag = iLI < iPos;
 scratch[iLI] += (bFlag)*(scratch[iLI + (bFlag)*iPos]);
}
```

Here, in each iteration of the loop, $iPos$ is divided by 2 (*iPos >>= 1*) and $bFlag$ is expanded to either 1 or 0, thus reducing by half the number of *work-items* doing a useful job. If, for the current *work-item*, the expression $iLI < iPos$ becomes true, then the expression in the last line will be interpreted as $scratch[iLI] += (1) * (scratch[iLI + (1) * iPos])$, ensuring that the value stored in position $iLI + iPos$ will be added to the value in position $iLI$. On the other hand, if the expression becomes false, it will be interpreted as $scratch[iLI] += (0) * (scratch[iLI + (0) * iPos])$, ensuring that the value in position $iLI$ will not be considered. Since all *work-items* are always in the same step of computation – doing exactly the same job (useful or not), independently of being in the

same wavefront – sync barriers are unnecessary.

## 4. Computational Experiments

In this section we evaluate the new approach against the proposals of Harris and Catanzaro. Their original codes[3] were public available and, therefore, used in the current study.

Table 1 and Figures 1 and 2 depict the speedup gains achieved against the code presented in [Catanzaro 2014], where $F = 1$ is the running time of the original code.

The algorithms were coded in the C++ language and compiled using a GNU compiler (g++ version 4.8.2 with parameters "`-O3 -mcmodel=medium -m64 -g -W -Wall`"). The AMD version used OpenCL 1.2 with the Software Development Kit 2.9.1. All experiments were performed on a computer with an AMD FX-9590 Black Edition Octa Core CPU, with clock ranging from 4.7GHz to 5.0GHz, 32GB of RAM, running Ubuntu 16.04 64-bits operating system. The computer had a Radeon SAPPHIRE R9 290X Tri-X OC GPU video card, with 4GB of memory. The architecture of such a video card provides 2816 stream processing units and an enhanced engine clock of up to 1040Mhz. Its memory is clocked at 1300MHz (5.2GHz effectively). At this speed, the theoretical memory bandwidth is close to 332.8GB/sec.

All tests were run on two vectors, one of integers and one of single precision floating points, containing 5533214 elements. There were no measurable differences, regarding the execution times, between the vector types.

The values listed in Table 1 were obtained with the OpenCL profiler CodeXL, version 2.0.12400.0, and are the averages of five consecutive executions for each $\boldsymbol{F}$.

As can be seen, these results show that the version of the algorithm with $F = 8$ reached a *speedup* pretty close to 2.8x, when compared with the proposal of [Catanzaro 2014]. It may also be noted that such *speedup* stabilizes around this value ($F = 16$ provided just over 1.5% gain when compared to $F = 8$).

| F | Time (ms) | Speedup | Memory Bandwidth (GB/s) | Bandwidth Usage (%) |
|---|---|---|---|---|
| 1 | 0.249780 | 1 | 88.6094002722 | 26.63 |
| 2 | 0.173930 | 1.4360949807 | 127.2515149773 | 38.24 |
| 3 | 0.139260 | 1.7936234382 | 158.9318971708 | 47.76 |
| 4 | 0.127700 | 1.955990603 | 173.3191542678 | 52.08 |
| 5 | 0.113930 | 2.1923988414 | 194.2671464935 | 58.37 |
| 6 | 0.100810 | 2.4777303839 | 219.5502033528 | 65.97 |
| 7 | 0.093740 | 2.6646042245 | 236.1089822914 | 70.95 |
| 8 | 0.089490 | 2.7911498491 | 247.3221142027 | 74.32 |
| 16 | 0.088160 | 2.8332577132 | 251.0532667877 | 75.44 |

**Table 1. Parallel reduction execution times. New approach compared against Catanzaro's original code.**

The same code was implemented in CUDA and tests were performed against the Kernel 7 presented in [Harris 2007]. The GPU used was a Tesla C2075 with 6GB of

[3]Luitjens' strategy is limited to NVidia's Kepler and newer architectures. Since we didn't have access to a video board using this architecture, it was not possible to compare our approach against it.

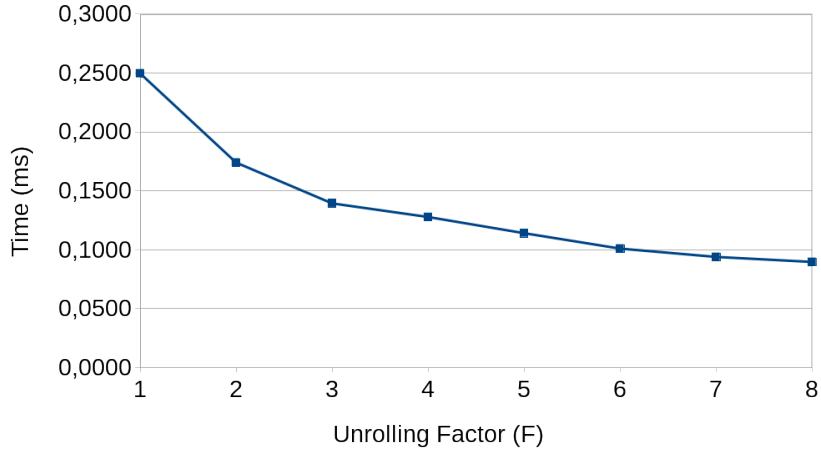



**Figure 1. Chart of the parallel reduction execution times.**

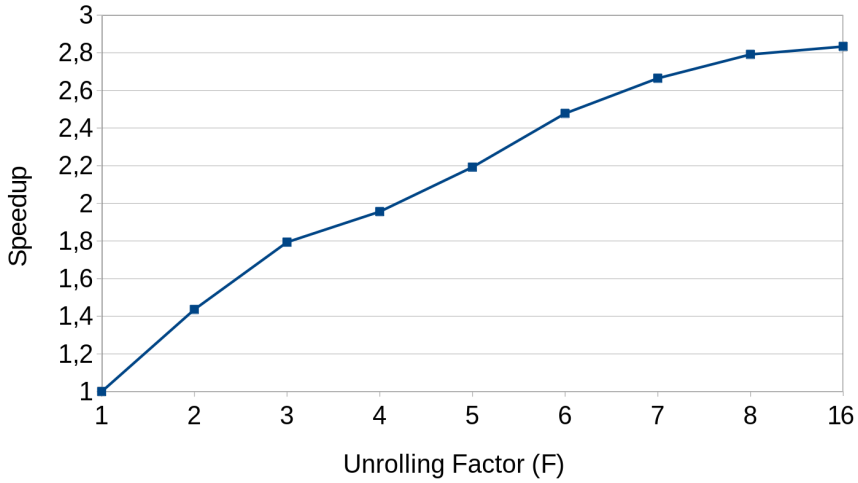



**Figure 2. Chart of the parallel reduction speedup.**

memory. Its architecture provides 448 CUDA cores, a GPU clock of 575MHz and a shader clock of 1150Mhz. Its memory is clocked at 750MHz (3.0GHz effective).

| Time – Kernel 7 | Time – New Approach | % of Performance |
|---|---|---|
| 0.17766 ms | 0.17867 ms | 99.4 |

**Table 2. Parallel reduction execution times – new approach (with unrolling factor equals to 8) compared against Harris' code.**

The experiments employed the two aforementioned vectors. Several values of the unrolling factor ($F$) were used in order to find the optimal value for such a video board. It was determined that up to $F = 6$ the performance gains were substantial and, with $F \geq 8$, the gains were very discrete. According to this, all experiments were conducted using $F = 8$, including the ones presented by [Harris 2007]. Table 2 presents the running time (in milliseconds) of both approaches and the percentage of performance (given by the formula $\frac{100 * T_{new}}{T_{k7}}$). The execution times of the two approaches are very similar and we assume them to be equivalent, particularly considering that the new code is simpler to

implement and works for CUDA and OpenCL.

## 5. General Remarks

Reduction operations are widely employed in many computational problems. This paper showed how such operations can be performed in a parallel fashion using graphics processing units and detailed the main approaches for them nowadays.

All parallel reduction techniques currently in use suffer from some basic issues. Several only reach their peak performance by employing proprietary strategies and/or technologies. That ends up limiting their use to the platform for which they were designed. Others, though generic, do not adopt certain procedures that could increase their performance without loss of generality.

The strategy presented here combines the best of both worlds: It is generic enough to be used with both CUDA and OpenCL and can run on hardware of the two major GPU manufacturers with minimal changes, just being adapted to the particularities of each platform. The implemented code, besides simpler, offered a performance equivalent to the best strategy described by [Harris 2007].

It is worth mentioning that the techniques here discussed are of general use, being the reduction only one of its possible applications.

## References

Billeter, M., Olsson, O., and Assarsson, U. (2009). Efficient Stream Compaction on Wide SIMD Many-Core Architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 159–166, New York, NY, USA. ACM.

Catanzaro, B. (2014). OpenCL Optimization Case Study: Simple Reductions. http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/. published by Advanced Micro Devices. Last accessed in January 05, 2014.

Chakroun, I., Mezmaz, M., Melab, N., and Bendjoudi, A. (2013). Reducing Thread Divergence in a GPU-Accelerated Branch-and-Bound Algorithm. *Concurrency and Computation: Practice and Experience*, 25(8):1121–1136.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2002). *Algoritmos: Teoria e Prática*. Editora Campus, 2 edition.

Fog, A. (2013). *Optimizing Subroutines in Assembly Language: An Optimization Guide for x86 Platforms*. Technical University of Denmark.

Fung, W. W. L., Sham, I., Yuan, G., and Aamodt, T. (2007). Dynamic warp formation and scheduling for efficient GPU control flow. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 407–420.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48.

Group, K. O. W. et al. (2008). The OpenCL specification. *version*, 1(29):8.

Gupta, K., Stuart, J. A., and Owens, J. D. (2012). A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14. IEEE.

Han, T. D. and Abdelrahman, T. S. (2011). Reducing branch divergence in GPU programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8, New York, NY, USA. ACM.

Harris, M. (2007). Optimizing Parallel Reduction in CUDA. https://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-parallel-reduction. published by NVidia Corporation. Last accessed in September 10, 2018.

Higham, N. (2002). *Accuracy and Stability of Numerical Algorithms: Second Edition*. Society for Industrial and Applied Mathematics.

Huang, J. C. and Leng, T. (1997). Generalized Loop-Unrolling: a Method for Program Speed-Up. In *in Proc. IEEE Symp. on Application-Specific Systems and Software Engineering and Technology*, pages 244–248.

Kiefer, J. C. (1953). Sequential Minimax Search for a Maximum. *Proc. Am. Math. Soc.*, 4:502–506.

Luitjens, J. (2014). Faster Parallel Reductions on Kepler. *White Paper*. published by NVidia Inc. Last accessed in July 25, 2014.

Meng, J., Tarjan, D., and Skadron, K. (2010). Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. *SIGARCH Comput. Archit. News*, 38(3):235–246.

Muller, J., Brisebarre, N., de Dinechin, F., Jeannerod, C., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., and Torres, S. (2009). *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston.

Narasiman, V., Shebanow, M., Lee, C. J., Miftakhutdinov, R., Mutlu, O., and Patt, Y. N. (2011). Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 308–317, New York, NY, USA. ACM.

Nasre, R., Burtscher, M., and Pingali, K. (2013). Data-driven versus topology-driven irregular computations on GPUs. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 463–474.

Parhami, B. (1999). *Introduction to Parallel Processing: Algorithms and Architectures*. Plenum series in computer science. Plenum Press.

Sarkar, V. (2001). Optimized Unrolling of Nested Loops. *Int. J. Parallel Program.*, 29(5):545–581.

Steinberger, M., Kainz, B., Kerbl, B., Hauswiesner, S., Kenzel, M., and Schmalstieg, D. (2012). Softshell: Dynamic scheduling on GPUs. *ACM Trans. Graph.*, 31(6):161:1–161:11.

Wilt, N. (2013). *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education.

Zhang, E. Z., Jiang, Y., Guo, Z., and Shen, X. (2010). Streamlining GPU Applications on the Fly: Thread Divergence Elimination Through Runtime Thread-data Remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 115–126, New York, NY, USA. ACM.

# Uma Abordagem Paralela usando GPGPU de Simulated Annealing para Solução do Problema Quadrático de Alocação

**Lucas Arakaki Takemoto**[1], **Bianca de Almeida Dantas**[1], **Henrique Mongelli**[1]

[1] Faculdade de Computação – Universidade Federal de Mato Grosso do Sul (UFMS)
Campo Grande – MS – Brasil

lucas.arakaki@ufms.br, {bianca,mongelli}@facom.ufms.br

***Abstract.*** *In the area of combinatorial optimization, the Quadratic Assignment Problem stands out due to its complexity and its range of applications. The main goal of this work, is to analyze the performance of Simulated Annealing when applied to the QAP. In an attempt to achieve better results, besides the sequential implementations, parallel strategies using GPUs have also been adopted.*

***Resumo.*** *Na área de otimização combinatória, o Problema Quadrático de Alocação se destaca pela sua complexidade e sua ampla gama de aplicações. O objetivo deste trabalho é analisar o desempenho da metaheurística Simulated Annealing quando aplicada ao PQA. Na tentativa de obter resultados mais competitivos, além das implementações sequenciais, estratégias de paralelização utilizando GPUs também foram adotadas.*

## 1. Introdução

O Problema Quadrático de Alocação (PQA), ou *Quadratic Assignment Problem (QAP)*, é um dos problemas mais explorados na área de otimização combinatória. O seu estudo é motivado pela sua fácil aplicação em uma infinidade de problemas reais. Além disso, Sahni e Gonzales [Sahni and Gonzalez 1976] provaram que o PQA pertence a uma classe de problemas conhecida como NP-difícil. Portanto, a não ser que P = NP, é impossível encontrar soluções ótimas para todas as instâncias em tempo de execução polinomial.

Entretanto, existem situações nas quais uma solução relativamente boa, quando obtida em tempo viável para o tamanho da instância, pode já ser suficiente. Nesse cenário, as metaheurísticas vêm se destacando como uma técnica interessante para solucionar não só o PQA, mas também diversos outros problemas de otimização combinatória [Gendreau and Potvin 2005], obtendo soluções de boa qualidade somadas a custos computacionais relativamente mais baixos.

Motivados pelos bons resultados obtidos em trabalhos anteriores como, por exemplo, em [Dantas and Cáceres 2018], ao usar a metaheurística *Simulated Annealing* (SA) na solução do problema da mochila multidimensional, o objetivo deste trabalho é avaliar a sua eficácia na obtenção de boas soluções para o PQA. Além da implementação sequencial, também foram adotadas estratégias de paralelização para reduzir o tempo de execução e melhorar a qualidade das soluções. Devido ao alto poder computacional das placas de vídeo atuais, foi implementado um algoritmo paralelo em CUDA (*Compute Unified Device Architecture*), uma API para a programação GPGPU criada pela NVIDIA.

O PQA e algumas de suas aplicações são apresentados na Seção 2. Na Seção 3, analisamos o algoritmo do *Simulated Annealing*. Uma nova proposta do SA é exibida na

Seção 4. Apresentamos os experimentos e uma avaliação comparativa na Seção 5. Por fim, a Seção 6 é destinada às conclusões e trabalhos futuros.

## 2. Problema Quadrático de Alocação (PQA)

Em 1957, Koopmans e Beckmann [Koopmans and Beckmann 1957] introduziram o PQA como um modelo matemático relacionado à localização de atividades econômicas. Considera-se a tarefa de alocar $n$ instalações em $n$ locais, onde o custo é o produto entre os fluxos das instalações e a distâncias dos locais, somado ao custo associado por alocar uma instalação em um determinado local. O objetivo é alocar cada uma das instalações em locais distintos, de tal modo que o custo total de alocação seja o menor possível.

Por ser facilmente resolvido e não mudar a complexidade do problema em questão [Burkard et al. 1998], a maioria dos autores desconsideram o custo associado por alocar uma instalação em um determinado local. Sendo assim, o PQA pode ser estabelecido da seguinte maneira [Burkard et al. 1998]:

$$\min_{\varphi \in S_n} \left( \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{\varphi_{(i)}\varphi_{(j)}} \right), \tag{1}$$

sendo $S_n$, o conjunto de todas as permutações possíveis dos inteiros 1, 2, ..., n. Cada produto $f_{ij} d_{\varphi_{(i)}\varphi_{(j)}}$ representa o custo de alocar a instalação $i$ no local $\varphi_{(i)}$ e a instalação $j$ no local $\varphi_{(j)}$.

Em adição aos problemas de *layout* de instalações, o PQA também aparece em aplicações de diferentes áreas. Em 1994, Phillips e Rosen [Phillips and Rosen 1994], empregaram o PQA em bioquímica, no problema da conformação molecular. Miranda *et al.* [Miranda et al. 2005] aplicaram o PQA para minimizar a quantidade de ligações entre componentes de placas de circuitos eletrônicos. Em 2009, Dell'Amico *et al.* [Dell'Amico et al. 2009] utilizaram o PQA para aperfeiçoar o *layout* de teclados de dispositivos pequenos, reduzindo o tempo necessário para digitar mensagens. Além disso, diversos problemas de otimização combinatória como o do caixeiro viajante, o da clique máxima e o do arranjo linear mínimo podem ser formulados como um PQA.

## 3. *Simulated Annealing*

Proposto originalmente por Kirkpatrick *et al*. [Kirkpatrick et al. 1983], o *Simulated Annealing* (SA) é uma analogia ao processo de recozimento aplicado aos sólidos. Introduzido por Metropolis *et al*. [Metropolis et al. 1953], este processo é caracterizado pelo aquecimento de um sólido até o seu ponto de fusão, seguido de um resfriamento lento até que sua rigidez seja alcançada.

Quando aplicado aos problemas de otimização combinatória, o SA pode ser modelado utilizando a teoria de cadeias de Markov [Eglese 1990]. Esta versão será referenciada como SA Genérico. O algoritmo é iniciado a uma temperatura inicial $t_0$ e uma solução inicial é construída aleatoriamente. A cada iteração, uma nova solução pertencente à vizinhança da solução corrente é gerada de forma aleatória. Essa solução é imediatamente aceita se sua qualidade for superior à solução corrente. Entretanto, soluções inferiores também podem ser aceitas com uma certa probabilidade. Após $m$ iterações, em que $m$ é o tamanho da cadeia de Markov, a temperatura é diminuída seguindo uma

taxa $\alpha$ de resfriamento. O algoritmo é encerrado quando a temperatura final $t_f$ é atingida. A probabilidade de uma solução $S'$ inferior à atual $S$ ser aceita, é dada pela seguinte equação:

$$p = e^{\frac{-\Delta}{T}}, \tag{2}$$

em que $T$ é a temperatura atual e $\Delta$ é a diferença entre a qualidade de $S'$ e $S$.

Analisando a Equação 2, observa-se que quanto menor a diferença entre a qualidade das soluções, maior é a chance de aceitação e com a diminuição da temperatura, a probabilidade de rejeição aumenta. Ao aceitar soluções de pior qualidade permite-se uma exploração em um maior espaço de busca, evitando que o algoritmo fique preso em ótimos locais. O Algoritmo 1 ilustra os passos executados pelo *Simulated Annealing Genérico*.

**Algoritmo 1:** SIMULATEDANNEALING($t_0$, $t_f$, $m$, $\alpha$)

```
1  LêEntrada();
2  Solução ← ConstróiSoluçãoAleatória();
3  MelhorSolução ← Solução;
4  T ← t_0;
5  enquanto T ≥ t_f faça
6      para i = 1 até m faça
7          NovaSolução ← ConstróiSoluçãoVizinhaAleatória(Solução);
8          Δ ← f(NovaSolução) − f(Solução);
9          r ← valor aleatório no intervalo [0, 1];
10         se Δ < 0 ou r < e^(−Δ/T) então
11             Solução ← NovaSolução;
12             se f(Solução) < f(MelhorSolução) então
13                 MelhorSolução ← Solução;
14             fim
15         fim
16     fim
17     T ← T ∗ α;
18 fim
19 retorna MelhorSolução;
```

Como se pode comprovar, pela análise das etapas ilustradas no Algoritmo 1, o SA depende da configuração de uma série de parâmetros. Wilhelm e Ward [Wilhelm and Ward 1987] concluíram que a efetividade do SA depende fortemente da configuração dessas variáveis, ou seja, a escolha equivocada desses parâmetros pode acarretar um impacto negativo no desempenho da metaheurística, tanto na qualidade das soluções como no tempo de execução.

Diante desta situação, Connolly [Connolly 1990] sugeriu um procedimento para ajustar os parâmetros do SA de acordo com as características da instância a ser resolvida. Considerando $m$, o número total de iterações, o método consiste em executar $\frac{m}{100}$ iterações do SA e considerar apenas movimentos que piorem a qualidade da solução, ou seja, quando $\Delta > 0$. O maior valor de $\Delta$ obtido é associado a $\delta_{max}$ enquanto o menor é associado a $\delta_{min}$. Assim, os parâmetros são definidos da seguinte forma [Connolly 1990]:

- $t_0 = \delta_{min} + \frac{\delta_{max} - \delta_{min}}{10}$;

- $t_f = \delta_{min}$;
- $\alpha = \frac{t_0 - t_f}{m * t_0 * t_f}$.

O Algoritmo 2 ilustra os passos do procedimento que calcula os valores dos parâmetros do SA.

**Algoritmo 2:** CALCULAPARÂMETROS($t_0$, $t_f$, *m*, $\alpha$)

```
1  Solução ← ConstróiSoluçãoAleatória();
2  tam ← m/100;
3  δ_max ← 0;
4  δ_min ← ∞;
5  para i=1 até tam faça
6      NovaSolução ← ConstróiSoluçãoVizinhaAleatória(Solução);
7      Δ ← f(NovaSolução) − f(Solução);
8      se Δ > 0 então
9          δ_min ← min(δ_min, Δ);
10         δ_max ← max(δ_max, Δ);
11     fim
12     Solução ← NovaSolução;
13 fim
14 t_0 ← δ_min + (δ_max − δ_min)/10;
15 t_f ← δ_min;
16 α ← (t_0 − t_f)/m ∗ t_0 ∗ t_f;
```

Com os parâmetros definidos, o *Simulated Annealing* pode ser iniciado. No algoritmo proposto por Connolly, o SA é finalizado em exatamente $m$ iterações, o que implica em algumas modificações em relação ao Algoritmo 1. A temperatura do sistema passa a ser resfriada a cada iteração, de acordo com a seguinte equação:

$$T_{i+1} = \frac{T_i}{1 + \alpha T_i}. \tag{3}$$

A aceitação de soluções de pior qualidade também é alterada. Além da probabilidade $p$ de se aceitar uma solução inferior, se um número máximo de rejeições $MaxFalhas$ consecutivas for atingido, então:

- a próxima solução é aceita;
- o resfriamento é pausado, ou seja, $\alpha = 0$;
- a temperatura é retornada para a temperatura $MelhorT$ na qual a melhor solução atual foi encontrada.

Connolly define $MaxFalhas = \frac{n(n-1)}{2}$ e introduz o conceito de temperatura ótima, ou seja, a existência de um temperatura fixa na qual o desempenho do SA é otimizado. Neste esquema, $MelhorT$ é um indicador de que as melhores soluções são obtidas nessa temperatura. O SA proposto por Connolly é ilustrado no Algoritmo 3.

## 4. Uma nova proposta de *Simulated Annealing* para o PQA

O *Simulated Annealing Modificado* (SAM) proposto neste trabalho consiste na junção das principais características das abordagens previamente discutidas Seção 3. O procedimento apresentado no Algoritmo 2, que determina os parâmetros do SA, é mantido. Entretanto, a taxa de resfriamento $\alpha$ deve ser escolhida manualmente e, preferencialmente,

**Algoritmo 3:** CONNOLLYSIMULATEDANNEALING($t_0$, $m$, $\alpha$)

```
1  LêEntrada();
2  Solução ← ConstróiSoluçãoAleatória();
3  MelhorSolução ← Solução;
4  T ← t_0;
5  tam ← m − m/100;
6  MaxFalhas ← n(n−1)/2;
7  para i = 1 até tam faça
8      NovaSolução ← ConstróiSoluçãoVizinhaAleatória(Solução);
9      Δ ← f(NovaSolução) − f(Solução);
10     r ← valor aleatório no intervalo [0, 1];
11     se Δ < 0 ou r < e^(−Δ/T) ou MaxFalhas = falhas então
12         falhas ← 0;
13         Solução ← NovaSolução;
14         se f(Solução) < f(MelhorSolução) então
15             MelhorSolução ← Solução;
16             MelhorT ← T;
17         fim
18         se MaxFalhas = falhas então
19             α ← 0;
20             T ← MelhorT;
21         fim
22     senão
23         falhas ← falhas + 1;
24     fim
25     T ← T/(1 + αT);
26 fim
27 retorna MelhorSolução;
```

utiliza-se um valor próximo de 1, para simular um resfriamento lento. Essa alteração é necessária pelo fato de que no SAM a temperatura do sistema é atualizada após a execução da cadeia de Markov. Além disso, o algoritmo só é finalizado quando a temperatura final é atingida, como ocorre no Algoritmo 1.

As condições de aceitação de uma solução permanecem inalteradas em relação ao Algoritmo 3, ou seja, uma nova solução só é aceita nos seguintes casos:

- $\Delta < 0$, ou seja, a qualidade da nova solução é melhor que a qualidade da solução corrente;
- $r < p$, em que $p$ é a probabilidade de aceitação dada pela Equação 2 e $r$ é um número aleatório entre 0 e 1;
- $MaxFalhas = falhas$, ou seja, se um número máximo de rejeições consecutivas tenham ocorrido anteriormente, a nova solução é aceita.

O Algoritmo 4 ilustra os passos realizados pelo *Simulated Annealing Modificado*.

### 4.1. Implementação paralela do *Simulated Annealing* para o PQA

Por ter apresentado os melhores resultados, em questão de qualidade, a estratégia de paralelização foi aplicada apenas ao algoritmo proposto *Simulated Annealing Modificado*.

**Algoritmo 4:** SIMULATEDANNEALINGMODIFICADO($t_0$, $t_f$, $m$, $\alpha$)

```
1  LêEntrada();
2  Solução ← ConstróiSoluçãoAleatória();
3  MelhorSolução ← Solução;
4  T ← t_0;
5  MaxFalhas ← n(n−1)/2;
6  enquanto T > t_f faça
7      para i = 1 até m faça
8          NovaSolução ← ConstróiSoluçãoVizinhaAleatória(Solução);
9          Δ ← f(NovaSolução) − f(Solução);
10         r ← valor aleatório no intervalo [0, 1];
11         se Δ < 0 ou r < e^(−Δ/T) ou MaxFalhas = falhas então
12             falhas ← 0;
13             Solução ← NovaSolução;
14             se f(Solução) < f(MelhorSolução) então
15                 MelhorSolução ← Solução;
16             fim
17         senão
18             falhas ← falhas + 1;
19         fim
20     fim
21     T ← T * α;
22 fim
23 retorna MelhorSolução;
```

Dentre as diversas técnicas de paralelização já desenvolvidas para o *Simulated Annealing* [Lee and Lee 1996, Onbaşoğlu and Özdamar 2001, Chen et al. 2007], a estratégia de Múltiplas Cadeias de Markov é a mais explorada [Ferreiro et al. 2013] e foi utilizada no desenvolvimento deste trabalho. Essa técnica pode ser abordada de maneira assíncrona ou síncrona. Em ambas as abordagens, múltiplas cadeias de Markov são executadas paralelamente. A diferença está na frequência de comunicação entre as *threads*.

Na abordagem assíncrona, as *threads* só se comunicam no final do processo de resfriamento, ou seja, após a temperatura final ser atingida. Na abordagem síncrona, as *threads* trocam informações após certos períodos de tempo, podendo ser a cada mudança de temperatura ou após um número fixo de iterações. Uma frequência maior de comunicações entre as *threads* resulta em pequenos acréscimos de tempo de execução. Em contrapartida, nos testes realizados, também proporcionou soluções de melhor qualidade. Sendo assim, foi escolhida a abordagem síncrona com trocas de informações a cada iteração da cadeia de Markov.

No SAM paralelo, uma solução inicial aleatória é construída e compartilhada com todas as *threads*. De forma paralela, cada *thread* constrói uma solução vizinha diferente. A função SincronizaThreads() é nativa da plataforma CUDA e garante que todas as *threads* tenham finalizado o processo anterior. Então, através de uma operação *reduce*, a melhor solução entre as *threads* é extraída. A *thread* $t_1$ age como o mestre e atualiza a solução corrente com a solução extraída no passo anterior, caso ela tenha sido aceita. No-

vamente, a função SincronizaThreads() é invocada para garantir que na próxima iteração todas as *threads* possuam a mesma solução corrente.

O Algoritmo 5 ilustra os passos realizados no *Simulated Annealing Modificado* utilizando GPGPU. Note que o conceito de $MaxFalhas$ é retirado, pois na abordagem paralela, o tamanho da cadeia de Markov é diminuído. Com os parâmetros utilizados nos experimentos, o número de iterações realizadas não superava o número de falhas consecutivas necessárias para uma solução inferior ser aceita.

**Algoritmo 5:** SIMULATEDANNEALINGMODIFICADO_GPGPU($t_0$, $t_f$, $m$, $\alpha$)

```
1  LêEntrada();
2  Solução ← ConstróiSoluçãoAleatória();
3  MelhorSolução ← Solução;
4  T ← t_0;
5  para cada thread_k, 1 ≤ k ≤ N_Threads faça em paralelo
6      enquanto T > t_f faça
7          para i = 1 até m faça
8              NovaSolução_k ← ConstróiSoluçãoVizinhaAleatória(Solução);
9              SincronizaThreads();
10             MelhorSoluçãoThreads ← melhor solução encontrada entre as threads;
11             se k = 1 então
12                 Δ ← f(MelhorSoluçãoThreads) − f(Solução);
13                 r ← valor aleatório no intervalo [0, 1];
14                 se Δ < 0 ou r < e^(−Δ/T) então
15                     Solução ← MelhorSoluçãoThreads;
16                     se f(Solução) < f(MelhorSolução) então
17                         MelhorSolução ← Solução;
18                     fim
19                 fim
20             fim
21             SincronizaThreads();
22         fim
23         T ← T * α;
24     fim
25 fim
26 retorna MelhorSolução;
```

## 5. Experimentos e Avaliação Comparativa

Todos os testes foram executados em uma máquina com processador Intel i7-4790k a 4,00 Ghz, 16 GB de memória RAM e uma placa de vídeo NVIDIA GeForce GTX 970 (4GB de memória e 1664 CUDA *Cores*).

Com o propósito de analisar a eficiência dos algoritmos e comparar os resultados obtidos, as instâncias utilizadas nos experimentos foram baseadas nas escolhidas por Fingler [Fingler 2013] e retiradas da biblioteca QAPLIB [Burkard et al. 1997], assim como as melhores soluções encontradas até o momento [QAPLIB 2018]. Utilizando a estatística *flow dominance* [Gambardella et al. 1999], é possível classificar as instâncias em duas categorias: regulares e irregulares.

De acordo com Taillard [Taillard 1995], instâncias regulares possuem matrizes de distância e fluxo com valores gerados aleatoriamente e distribuidos uniformemente. Como não possuem uma estrutura, as boas soluções estão bastante espalhadas no espaço de busca, dificultando a descoberta de bons resultados. Por outro lado, as instâncias irregulares apresentam características semelhantes aos problemas da vida real. Geralmente, as matrizes possuem valores bastante distintos e muitos zeros, imitando as distribuições observadas em aplicações reais.

Para medir a qualidade das soluções, utilizou-se o conceito de *gap*, que representa a porcentagem da diferença entre a solução obtida (SO) e a melhor solução conhecida (MSC) até o momento. Sendo assim, o *gap* é calculado da seguinte maneira:

$$gap = \frac{SO - MSC}{MSC} * 100. \tag{4}$$

Para cada algoritmo, foram calculados o *gap* mínimo, o *gap* médio e o tempo de execução em segundos. Para evitar que valores incomuns influenciassem de forma inadequada nos resultados finais, os dados apresentados nas Tabelas 1, 2 e 3 foram obtidos através da média aritmética dos resultados de 30 execuções sobre cada instância de tamanho $n$. Na coluna *Gap Mínimo*, o número entre parênteses indica a quantidade de vezes que a solução ótima foi encontrada, enquanto na coluna *Gap Médio* valores em negrito evidenciam os melhores resultados. Para os algoritmos paralelos, a coluna *Speedup* é acrescentada, indicando o grau de desempenho, em termos de tempo de execução da versão paralela em relação à versão sequencial.

A Tabela 1 mostra os resultados obtidos pelas três versões sequenciais do SA discutidas neste trabalho: SA Genérico, SA por Connolly e SA Modificado. Por causa da configuração manual de parâmetros, o desempenho do SA Genérico é prejudicado. Temperaturas iniciais muito altas implicam em um tempo de execução maior enquanto valores muito baixos restringem a aceitação de soluções com qualidade inferior, ou seja, a chance do algoritmo parar em um ótimo local aumenta.

O SA proposto por Connolly se destaca pelo seu baixo custo computacional. Entretanto, como a temperatura passa a ser resfriada a cada iteração, poucas soluções vizinhas são examinadas, prejudicando o processo de busca.

A proposta do SA Modificado é suprir as deficiências dos dois SAs anteriormente citados. Com a utilização do procedimento que calcula os valores dos parâmetros iniciais, agregado a uma estratégia que constrói diversas soluções vizinhas na mesma temperatura, o SA Modificado obteve boas soluções em tempos de execução relativamente baixos. Em 15 das 20 instâncias testadas, foram encontradas soluções com menos de 1% de *gap* em no máximo 1 segundo.

A Tabela 2 exibe os resultados do SA Modificado paralelo ao lado dos resultados da sua versão sequencial. Configuramos a taxa de resfriamento $\alpha$=0.995, permitindo que mais iterações fossem executadas e, consequentemente, um espaço maior de soluções fosse explorado.

Durante os experimentos, observamos que utilizar grandes quantidades de *threads* implicava em um crescimento significante na frequência de trocas de informações e, consequentemente, no tempo de execução. Por outro lado, fixar um número menor de *threads*

**Tabela 1. Resultados obtidos pelas versões sequenciais dos algoritmos *Simulated Annealing*.**

| Instância | $n$ | SA Genérico ($t_0$=1000; $t_f$=0.00001 m=10000; $\alpha$=0,95) | | | SA por Connolly ($t_0$=NA; $t_f$=NA; m=1000000; $\alpha$=NA) | | | SA Modificado ($t_0$=NA; $t_f$=NA; m=10000; $\alpha$=0,95) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | *Gap* Mínimo | *Gap* Médio | Tempo (s) | *Gap* Mínimo | *Gap* Médio | Tempo (s) | *Gap* Mínimo | *Gap* Médio | Tempo (s) |
| nug20 | 20 | 0 (12) | 0,1945 | 0,8863 | 0 (25) | 0,0259 | 0,2285 | 0 (27) | **0,0155** | 0,1268 |
| nug30 | 30 | 0 (2) | 0,3527 | 1,1769 | 0 (15) | **0,1382** | 0,3138 | 0 (4) | 0,2591 | 0,1929 |
| sko42 | 42 | 0 (4) | 0,2327 | 1,5377 | 0 (6) | **0,1463** | 0,4163 | 0 (3) | 0,2782 | 0,2712 |
| sko72 | 72 | 0,1539 | **0,3885** | 2,6808 | 0,0815 | 0,4007 | 0,7214 | 0,1509 | 0,4454 | 0,5409 |
| tai20a | 20 | 1,5096 | 3,5435 | 0,8723 | 0 (2) | **0,5671** | 0,2288 | 0 (3) | 0,5757 | 0,2301 |
| tai30a | 30 | 1,8636 | 3,4169 | 1,1814 | 0,0162 | **1,1599** | 0,3235 | 0,5965 | 1,3161 | 0,3056 |
| tai40a | 40 | 2,7811 | 3,9316 | 1,4651 | 0,8204 | 2,0214 | 0,3993 | 1,0607 | **1,7679** | 0,3768 |
| tai50a | 50 | 2,6124 | 4,0766 | 1,7627 | 1,3861 | 2,6248 | 0,4943 | 1,2058 | **2,2891** | 0,4542 |
| tai60a | 60 | 2,9738 | 3,9146 | 2,1509 | 1,6153 | 2,6357 | 0,5895 | 1,2964 | **2,3834** | 0,5501 |
| tai80a | 80 | 2,8471 | 3,8231 | 2,9827 | 1,5857 | 2,6103 | 0,8109 | 1,4523 | **2,4186** | 0,7552 |
| bur26a | 26 | 0 (3) | 0,1509 | 1,0549 | 0,0203 | 0,2456 | 0,2728 | 0 (2) | **0,1041** | 0,3997 |
| chr25a | 25 | 1,8441 | 14,8208 | 1,0282 | 10,1159 | 21,7843 | 0,2732 | 2,0547 | **13,5001** | 0,2692 |
| els19 | 19 | 0,8992 | 28,4306 | 0,8345 | 4,2089 | 25,7124 | 0,2117 | 0 (13) | **1,4954** | 0,3538 |
| kra30a | 30 | 0 (7) | 1,2332 | 1,1827 | 0 (4) | 2,0056 | 0,3213 | 0 (8) | **1,1837** | 0,2422 |
| tai20b | 20 | 1,2894 | 16,5104 | 0,8717 | 0 (6) | 6,2487 | 0,2275 | 0 (26) | **0,0415** | 0,3691 |
| tai30b | 30 | 0,6311 | 12,6991 | 1,1721 | 0,5801 | 12,7626 | 0,3084 | 0 (7) | **1,0562** | 0,5015 |
| tai40b | 40 | 1,9754 | 9,6481 | 1,4561 | 0 (1) | 8,2607 | 0,3969 | 0 (10) | **1,3416** | 0,5287 |
| tai50b | 50 | 1,6531 | 7,1211 | 1,7774 | 0,5265 | 5,8571 | 0,4764 | 0,0022 | **0,4911** | 0,6592 |
| tai60b | 60 | 1,3475 | 7,7143 | 2,1225 | 0,5624 | 5,9968 | 0,5755 | 0 (1) | **0,5238** | 0,8316 |
| tai80b | 80 | 3,0601 | 5,4191 | 2,9622 | 1,8176 | 4,9321 | 0,7971 | 0,0695 | **1,2529** | 1,0402 |

afetava a qualidade das soluções, pois o número de iterações da cadeia de Markov executadas era reduzido. Diante destas circunstâncias e analisando os resultados experimentais, ao empregar um *grid* de 20 blocos com 100 *threads* cada, o algoritmo paralelo foi capaz de obter soluções de boa qualidade somadas a baixos custos computacionais.

Em relação ao SAM sequencial, que encontrou soluções com menos de $1\%$ de *gap* em 19 casos de testes, o SAM paralelo apresentou *gaps* médios menores ou iguais em 17 das 20 instâncias testadas. Em termos de tempo de execução, o SAM paralelo foi mais eficiente em todas as instâncias testadas, obtendo um *speedup* médio de 1,39 em relação ao SAM sequencial. Por ser um algoritmo inerentemente sequencial [Ferreiro et al. 2013], abordar o SA de forma paralela é uma tarefa desafiadora. O algoritmo paralelo proposto neste trabalho foi desenvolvido de forma que ele pudesse ser facilmente adaptado para diferentes plataformas e, consequentemente, não explora integralmente os benefícios cedidos pela computação em GPUs.

Para comparar o desempenho do SAM paralelo diante de outras metaheurísticas, fornecemos na Tabela 3, os resultados obtidos por duas implementações paralelas encontradas na literatura: *Ant Colony Optimization* (ACO) [Fingler 2013] e *Cooperative Parallel Tabu Search* (CPTS) [James et al. 2009]. Notamos que o SAM encontrou soluções ótimas para 14 das 20 instâncias testadas, superando o ACO na qualidade das soluções (0,4747 contra 0,6448), porém perdendo para o CPTS (0,102). Em termos de tempo de execução, o SAM se mostrou dominante, com uma média de 3,36 segundos enquanto o ACO e o CPTS levaram 19,398 e 1403,6 segundos, respectivamente.

**Tabela 2. Resultados obtidos pelas versões sequencial e paralela do algoritmo *Simulated Annealing Modificado*.**

| Instância | $n$ | SA Modificado Sequencial ($t_0$=NA; $t_f$=NA; m=10000; $\alpha$=0,995; n_threads=NA) | | | SA Modificado GPGPU ($t_0$=NA; $t_f$=NA; m=100; $\alpha$=0,995; n_threads=2000) | | | |
|---|---|---|---|---|---|---|---|---|
| | | *Gap* Mínimo | *Gap* Médio | Tempo (s) | *Gap* Mínimo | *Gap* Médio | Tempo (s) | *Speedup* |
| nug20 | 20 | 0 (30) | **0** | 1,2244 | 0 (30) | **0** | 0,8579 | 1,4272 |
| nug30 | 30 | 0 (26) | 0,0087 | 1,8557 | 0 (30) | **0** | 1,2299 | 1,5088 |
| sko42 | 42 | 0 (18) | 0,0455 | 2,6687 | 0 (26) | **0,0033** | 1,8195 | 1,4667 |
| sko72 | 72 | 0,0061 | 0,1905 | 5,3499 | 0,0061 | **0,0415** | 3,9651 | 1,3492 |
| tai20a | 20 | 0 (14) | **0,1806** | 2,2841 | 0 (7) | 0,2751 | 1,5331 | 1,4898 |
| tai30a | 30 | 0 (1) | 0,5131 | 3,0074 | 0 (13) | **0,2166** | 1,9311 | 1,5573 |
| tai40a | 40 | 0,3063 | 1,1842 | 3,7551 | 0,3751 | **0,8039** | 2,5072 | 1,4977 |
| tai50a | 50 | 0,9082 | 1,5648 | 4,6372 | 0,7198 | **1,1422** | 3,3477 | 1,3851 |
| tai60a | 60 | 0,9938 | 1,7721 | 5,5416 | 0,8191 | **1,2332** | 4,4875 | 1,2348 |
| tai80a | 80 | 1,3525 | 1,7926 | 7,5112 | 1,0859 | **1,4409** | 5,8698 | 1,2796 |
| bur26a | 26 | 0 (18) | 0,0225 | 4,1175 | 0 (26) | **0,0076** | 2,8264 | 1,4568 |
| chr25a | 25 | 0 (9) | 4,0201 | 2,7275 | 0 (8) | **2,0758** | 1,8492 | 1,4749 |
| els19 | 19 | 0 (30) | **0** | 3,5784 | 0 (21) | 1,2627 | 2,7248 | 1,3132 |
| kra30a | 30 | 0 (23) | 0,2969 | 2,3751 | 0 (27) | **0,1338** | 1,7831 | 1,3320 |
| tai20b | 20 | 0 (30) | **0** | 3,7997 | 0 (30) | **0** | 2,5944 | 1,4645 |
| tai30b | 30 | 0 (20) | **0,0284** | 5,1001 | 0 (11) | 0,1746 | 3,7112 | 1,3742 |
| tai40b | 40 | 0 (24) | 0,4036 | 5,4101 | 0 (25) | **0,2035** | 3,7521 | 1,4418 |
| tai50b | 50 | 0 (4) | 0,3223 | 6,7493 | 0 (9) | **0,0986** | 5,1784 | 1,3033 |
| tai60b | 60 | 0 (8) | 0,1311 | 8,6118 | 0 (10) | **0,0558** | 7,0187 | 1,2269 |
| tai80b | 80 | 0 (2) | 0,4421 | 10,5091 | 0 (1) | **0,3266** | 8,2326 | 1,2765 |

## 6. Conclusões e Trabalhos Futuros

Neste trabalho, para solucionar o PQA, apresentamos uma nova proposta de *Simulated Annealing* que se mostrou competitiva em relação aos algoritmos SA Genérico e SA por Connolly. Em adição à versão sequencial do SAM, também adotamos estratégias de paralelização em GPGPU, que proporcionaram ganhos significantes nos resultados. Por fim, realizamos uma análise comparativa com as metaheurísticas CPTS e ACO, que evidenciou o potencial do algoritmo SAM em obter soluções de boa qualidade em tempos de execução viáveis. Até o momento da conclusão deste trabalho, não havíamos encontrado soluções para o PQA, usando SA e GPGPU.

Como trabalhos futuros, pretende-se explorar novos métodos de busca local, com o intuito de explorar um maior espaço de soluções de maneiras mais eficientes. Outras técnicas para a construção da solução inicial também podem ser estudadas, como a utilização do conceito de lista restrita de candidatos (LRC) da metaheurística GRASP [Dantas and Cáceres 2018]. No que diz respeito às implementações GPGPU, novas maneiras de dividir as tarefas designadas às *threads* podem ser desenvolvidas. Além disso, ter um cuidado especial com as estruturas utilizadas e na forma em que a memória é acessada, é fundamental para que o poder de processamento das GPUs seja extraído ao máximo, de forma a acelerar a execução das metaheurísticas.

**Tabela 3. Resultados obtidos pelo SAM paralelo, ACO e CPTS. (Valores sublinhados indicam os menores tempos de execução)**

| | | SA Modificado GPGPU (t0=NA; tf=NA; m=100; a=0.995; n_threads=2000) | | ACO GPGPU | | CPTS OpenMP | |
|---|---|---|---|---|---|---|---|
| Instância | *n* | *Gap* Médio | Tempo (s) | *Gap* Médio | Tempo (s) | Gap Médio | Tempo (s) |
| nug20 | 20 | **0 (30)** | 0,857 | **0** | 0,984 | - | - |
| nug30 | 30 | **0 (30)** | 1,229 | 0,0065 | 3,038 | **0** | 102 |
| sko42 | 42 | 0,0033 (26) | 1,819 | **0** | 9,482 | **0** | 318 |
| sko72 | 72 | 0,0415 | 3,965 | 0,0631 | 63,049 | **0** | 4176 |
| tai20a | 20 | 0,2751 (7) | 1,533 | 0,2737 | 0,984 | **0** | 6 |
| tai30a | 30 | 0,2166 (13) | 1,931 | 0,5671 | 3,040 | **0** | 96 |
| tai40a | 40 | 0,8039 | 2,507 | 1,9371 | 7,834 | **0,148** | 210 |
| tai50a | 50 | 1,1422 | 3,347 | 3,1779 | 18,722 | **0,440** | 618 |
| tai60a | 60 | 1,2332 | 4,487 | 3,4494 | 32,709 | **0,476** | 1584 |
| tai80a | 80 | 1,4409 | 5,869 | 3,4116 | 88,992 | **0,570** | 5688 |
| bur26a | 26 | 0,0076 (26) | 2,826 | **0** | 2,020 | - | - |
| chr25a | 25 | 2,0758 (8) | 1,849 | **0** | 1,806 | - | - |
| els19 | 19 | 1,2627 (21) | 2,724 | **0** | 0,867 | **0** | 6 |
| kra30a | 30 | 0,1338 (27) | 1,783 | **0** | 3,038 | - | - |
| tai20b | 20 | **0 (30)** | 2,594 | **0** | 0,981 | **0** | 6 |
| tai30b | 30 | 0,1746 (11) | 3,711 | **0** | 3,040 | **0** | 72 |
| tai40b | 40 | 0,2035 (25) | 3,752 | **0** | 7,844 | **0** | 270 |
| tai50b | 50 | 0,0986 (9) | 5,178 | 0,0014 | 18,622 | **0** | 828 |
| tai60b | 60 | 0,0558 (10) | 7,018 | 0,0024 | 32,780 | **0** | 1824 |
| tai80b | 80 | 0,3266 (1) | 8,232 | 0,0075 | 88,139 | **0** | 6654 |
| Média | | 0,4747 | 3,360 | 0,6448 | 19,398 | **0,102** | 1403,6 |

## Referências


Burkard, R. E., Cela, E., Pardalos, P. M., and Pitsoulis, L. S. (1998). The quadratic assignment problem. In *Handbook of Combinatorial Optimization*, pages 1713–1809. Springer.

Burkard, R. E., Karisch, S. E., and Rendl, F. (1997). QAPLIB – A Quadratic Assignment Problem Library. *Journal of Global optimization*, 10(4):391–403.

Chen, D.-J., Lee, C.-Y., Park, C.-H., and Mendes, P. (2007). Parallelizing simulated annealing algorithms based on high-performance computer. *Journal of Global Optimization*, 39(2):261–289.

Connolly, D. T. (1990). An improved annealing scheme for the QAP. *European Journal of Operational Research*, 46(1):93–100.

Dantas, B. A. and Cáceres, E. N. (2018). An experimental evaluation of a parallel simulated annealing approach for the 0–1 multidimensional knapsack problem. *Journal of Parallel and Distributed Computing*, 120:211 – 221.

Dell'Amico, M., Diaz, J. C. D., Iori, M., and Montanari, R. (2009). The single-finger keyboard layout problem. *Computers & Operations Research*, 36(11):3002–3012.

Eglese, R. (1990). Simulated annealing: a tool for operational research. *European Journal of Operational Research*, 46(3):271–281.

Ferreiro, A., García, J., López-Salas, J. G., and Vázquez, C. (2013). An efficient implementation of parallel simulated annealing algorithm in GPUs. *Journal of Global Optimization*, 57(3):863–890.

Fingler, H. (2013). Otimização de colônias de formigas em CUDA: O Problema da Mochila Multidimensional e o Problema Quadrático de Alocação. Dissertação de Mestrado, Faculdade de Computação - Universidade Federal de Mato Grosso do Sul.

Gambardella, L. M., Taillard, É. D., and Dorigo, M. (1999). Ant colonies for the quadratic assignment problem. *Journal of the Operational Research Society*, 50(2):167–176.

Gendreau, M. and Potvin, J.-Y. (2005). Metaheuristics in combinatorial optimization. *Annals of Operations Research*, 140(1):189–213.

James, T., Rego, C., and Glover, F. (2009). A cooperative parallel tabu search algorithm for the Quadratic Assignment Problem. *European Journal of Operational Research*, 195(3):810–826.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680.

Koopmans, T. C. and Beckmann, M. (1957). Assignment problems and the location of economic activities. *Econometrica: Journal of the Econometric Society*, pages 53–76.

Lee, S.-Y. and Lee, K. G. (1996). Synchronous and asynchronous parallel simulated annealing with multiple Markov chains. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):993–1008.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953). Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092.

Miranda, G., Luna, H. P. L., Mateus, G. R., and Ferreira, R. P. M. (2005). A performance guarantee heuristic for electronic components placement problems including thermal effects. *Computers & operations research*, 32(11):2937–2957.

Onbaşoğlu, E. and Özdamar, L. (2001). Parallel simulated annealing algorithms in global optimization. *Journal of Global Optimization*, 19(1):27–50.

Phillips, A. T. and Rosen, J. B. (1994). A quadratic assignment formulation of the molecular conformation problem. *Journal of Global Optimization*, 4(2):229–241.

QAPLIB (2018). QAPLIB - A Quadratic Assignment Problem Library. `http://anjos.mgi.polymtl.ca/qaplib/`. Acessado em Julho de 2018.

Sahni, S. and Gonzalez, T. (1976). P-complete approximation problems. *Journal of the ACM (JACM)*, 23(3):555–565.

Taillard, E. D. (1995). Comparison of iterative searches for the quadratic assignment problem. *Location Science*, 3(2):87–105.

Wilhelm, M. R. and Ward, T. L. (1987). Solving Quadratic Assignment Problems by 'simulated annealing'. *IIE Transactions*, 19(1):107–119.

# Escalonamento de Transações a Nível de Usuário em Haskell

**Rodrigo M. Duarte**[1]*, **André R. Du Bois**[1], **Gerson G.H. Cavalheiro**[1], **Maurício L. Pilla**[1]

[1]Universidade Federal de Pelotas (UFPel)
Programa de Pós Graduação em Computação (PPGC)
Gomes Carneiro 1 - CEP 96010-610 – Pelotas – RS – Brasil

{rmduarte,dubois,gerson.cavalheiro,pilla}@inf.ufpel.edu.br

***Abstract.*** *Transactional Memory is an abstraction that helps concurrent programming, however, in high contention sceneries, it presents low performance because of the high conflict rate between transactions. In this work, we present four transactional schedulers implemented entirely in Haskell using different abstraction levels. The results present, despite the inherent overhead of high-level implementations, a reduction in the conflict rates.*

***Resumo.*** *Memória Transacional é uma abstração que facilita a programação concorrente. Entretanto, em cenários de alta contenção, apresenta baixo desempenho devido as altas taxas de cancelamento. Neste artigo são apresentadas quatro implementações de escalonadores de transações em Haskell sobre diferentes níveis de abstração. Os resultados mostram que, apesar do overhead imposto pelas implementações em alto nível, as mesmas conseguem reduzir a quantidade de cancelamentos.*

## 1. Introdução

Memória Transacional (MT) é um modelo de sincronização entre *threads* que fornece ao programador um elevado nível de abstração. Em MT as seções críticas de um código concorrente são tratadas como transações, parecidas com as presentes em bancos de dados. Neste modelo o programador somente define a região que deve ser protegida, ficando a sincronização a cargo do sistema transacional.

Apesar da facilidade da programação usando MT, a mesma ainda sofre perda de desempenho quando o sistema possui alta contenção de memória, pois diferentes transações acessam as mesmas regiões de memória, aumentando o número de conflitos (cancelamentos), degradando o desempenho [Yoo and Lee 2008].

Com o objetivo de suprimir este problema, o escalonamento de transações vem sendo empregado [Maldonado et al. 2010]. O objetivo é reduzir a quantidade de cancelamentos re-ordenando a execução das transações. A principal ideia é fazer com que as transações conflitantes sejam serializadas, evitando assim que as mesmas conflitem novamente pelos mesmos motivos. Executar transações conflitantes de forma sequencial em sistemas de alta contenção, permite que outros *cores* do processador possam ser usados por outras transações que não são conflitantes, fazendo com que o sistema transacional utilize de forma mais racional os recursos computacionais.

*O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001

Implementar escalonadores é uma tarefa de relativa dificuldade, ações como a correta sincronização e gerência de memória são itens críticos a serem realizados pelo escalonador. Linguagens funcionais, como Haskell, no entanto trazem abstrações que facilitam a criação de programas concorrentes, entre elas a ausência de efeitos colaterais e um forte tratamento de tipos [Marlow 2013], estas características permitem a implementação de programas mais robustos e livres de erros. Como exemplo de implementações de abstrações de alto nível para programação concorrente em Haskell, podemos citar duas bibliotecas de memórias transacionais, (TL2 [Du Bois 2011] e TINY [Duarte et al. 2015]).

A ferramenta mais completa para a programação em Haskell é o *Glasgow Haskell Compiler* (GHC) [Peyton-Jones et al. 2017] que, apesar de esta ferramenta possuir MT implementada em seu *Run Time System* (RTS), o mesmo não possui nenhum tipo de escalonamento específico para transações. Assim, programas desenvolvidos usando MT no GHC e que possuam alta contenção de memória, tendem a perder desempenho.

A fim de estudar o comportamento de escalonadores de transações implementados a nível de usuário em uma linguagem de elevada abstração, este trabalho apresenta dois modelos de escalonadores, implementados completamente em Haskell usando duas diferentes bibliotecas de MT também implementadas nesta linguagem. Além disso implementou-se os mesmos algoritmos de escalonamento em uma modificação do GHC chamada de *Lightweight Concurrency* (LWC) [Li et al. 2013]. O LWC possui funções, em uma interface de alto nível em código Haskell, possibilitando a manipulação direta do RTS, permitindo assim a criação de diferentes modelos de escalonadores na máquina virtual Haskell, gerando escalonadores mais eficientes do que se implementados totalmente em Haskell.

Assim, com o exposto acima, este trabalho faz as seguintes contribuições:

- Apresenta a implementação em Haskell de quatro escalonadores de transações usando dois modelos de escalonamento diferentes
- Faz uma análise dos *overheads* impostos por usar ferramentas com elevada abstração na implementação de escalonadores de transações
- Verifica o comportamento (em relação ao nível de conflitos) dos escalonadores ao usar diferentes algoritmos de versionamento em STM.

Os resultados obtidos mostram que apesar do elevado *overhead* imposto pela implementação em alto nível dos escalonadores, estes conseguiram reduzir a quantidade de cancelamentos das aplicações, demostrando a possibilidade de implementar tarefas complexas em uma linguagem de maior abstração. Também, ao se usar primitivas que se comunicam diretamente com o RTS, consegue-se reduzir ainda mais o custo inserido pelo escalonador no sistema transacional.

## 2. Memórias Transacionais, STM Haskell e LWC

MT usam dois conceitos chave para realizar a correta sincronização entre *threads*, o versionamento de dados e a detecção de conflitos. O Versionamento de dados define a forma como os dados especulativos serão tratados pela transação e este pode ser feito de forma adiantada ou tardia. Na forma adiantada os dados especulativos são gravados diretamente na memória e o dado original em um *undo-log* na transação. Em caso de efetivação o mesmo já se encontra na memória e no caso de cancelamento, há a necessidade de restaurar o dado original na memória. Já no versionamento tardio os dados especulativos são

guardados em um *log* na transação, em caso de efetivação os dados devem ser gravados na memória e em caso de cancelamento, o *log* é somente descartado. A detecção de conflitos também pode ser realizada da mesma forma (adiantada ou tardia). Na adiantada o conflito é detectado no momento em que a transação acessa o dado na memória, já na tardia, no momento da validação da transação.

Estas características são o que garantem as propriedades de atomicidade (as transações devem ser executadas como se fossem um único passo do programa) e isolamento (nenhuma transação deve enxergar o resultado intermediário de outra). O uso de cada uma das opções acima (adiantada ou tardia) depende do tipo da aplicação que usará MT [Rigo et al. 2007]. Em programas que possuem alta contenção de memória (alta taxa de conflitos), privilegia o uso de versionamento tardio, pois o dado especulativo somente dever ser descartado em um conflito. Já em ambientes de baixa contenção, versionamento adiantado é melhor, pois os dados já se encontram em memória e nada mais precisa ser feito em uma efetivação.

GHC (*Glasgow Haskell Compiler*) [Peyton-Jones et al. 2017] é um compilador, interpretador e máquina virtual para Haskell que inclui uma biblioteca de MT implementada em sua máquina virtual, o STM-Haskell. Nesta, a abstração de varíaveis transacionais, ou `TVars` [Harris et al. 2008], é apresentada. Uma `TVar` só pode ser modificada por duas funções que são a `readTVar` (leitura) e `writeTVar` (escrita). A combiação dessas funções gera ações transacionais que só podem ser executadas em um bloco atômico criado pela primitiva `atomically`. O sistema de tipos do Haskell garante que operações transacionais não serão realizadas fora de um `atomically`. Isso ajuda o programador a corrigir erros já no momento da compilação. O STM Haskell também disponibiliza duas primitivas para composição de ações transacionais, o `retry` que permite reinicializar transações e o `orElse` que combina ações que podem gerar um `retry`.

Apesar das qualidades citadas anteriormente, o RTS do GHC não possui um escalonador específico para transações, e a implementação de STM possui apenas versionamento tardio com deteção tardia de conflitos.

LWC [Li et al. 2007], é uma modificação do RTS do GHC na qual são inseridas primitivas para manipular diretamente a criação de *threads* e geração de filas de escalonamento. Nesta são fornecidos dois tipos de dados que são o `SCont` e a `PTM`. Um `SCont` (*stack continuation*) é uma espécie de ponteiro para um TSO (*Thread State Object*), o que permite a criação/manipulação direta de *threads* a nível de usuário, ou seja, um `SCont` pode ser manipulado pelo programador e não mais de forma automática e isolada, como é realizado pelo RTS do GHC. Também, para a manipulação segura dos `SConts`, uma primitiva transacional é definida, `PTM` (*Primitive Transactional Memory*), sendo este o único mecanismo de sincronização exposto pelo RTS no LWC.

## 3. Trabalhos Relacionados

A ideia de escalonamento de transações surge a partir da necessidade de reduzir o número de cancelamentos em sistemas transacionais. Um cancelamento pode ocorrer quando duas ou mais transações acessam uma mesma área de memória e pelo menos um destes acessos é de escrita. Para reduzir a quantidade de conflitos os escalonadores transacionais, em sua maioria, usam a ideia de serialização. Executar transações conflitantes de forma sequencial, em sistemas de alta contenção, permite que outros *cores* do processador possam ser

usados por outras transações não conflitantes, fazendo com que o sistema transacional use de forma mais racional os recursos computacionais.

Um dos primeiros trabalhos a usar o escalonamento de transações foi ATS [Yoo and Lee 2008]. Neste trabalho os autores utilizam uma heurística que define a intensidade de conflitos no sistema. Se a intensidade de conflitos aumenta, o sistema começa a serializar todas as transações, quando a intensidade diminui, o sistema começa a aumentar a concorrência entre as transações novamente. Esta intensidade de conflitos é mensurada pelo histórico de cancelamentos das transações.

CAR-STM [Dolev et al. 2008] é um escalonador que apresenta duas formas distintas de lidar com cancelamentos. Na primeira, serializa transações conflitantes em uma mesma fila de execução de um mesmo processador. Na outra é usando instrumentação para analisar quais dados as transações estão acessando e tentar prever futuros conflitos entre as mesmas. Caso um conflito seja detectado, a transação é migrada para a fila de tarefas de outro processador.

Em [Dragojević et al. 2009], é apresentado SHRINK, um escalonador que serializa as transações através da análise das regiões de memória que as transações acessam. Se as transações acessam as mesmas áreas de memória e o nível de contenção começa a aumentar, o escalonador começa a serializar todas as transações, caso o nível de contenção diminua as transações voltam a ser executadas de forma concorrente.

No trabalho de [Nicácio et al. 2013], um escalonador de transações com dois modelos de decisão é apresentado. O primeiro modelo utiliza uma heurística para decidir a quantidade de transações a serem executadas de forma concorrente e o segundo, utiliza uma tabela de probabilidades para decidir qual transação deve ser escalonada. A heurística é usada para transações pequenas, para não onerar o sistema transacional. Já a tabela é usada para transações longas, pois o tempo de execução das transações sobrepõem o *overhead* do escalonador.

Já em [Di Sanzo et al. 2016], é apresentado um escalonador que define a quantidade de concorrência entre as transações através de uma cadeia de Markov. Quando uma transação é iniciada, o escalonador muda o estado da cadeia, modificando a probabilidade para iniciar uma nova transação, através da análise do histórico anterior. As transições da cadeia definem a probabilidade de quantas transações podem ser executadas concorrentemente no sistema, evitando o efeito de *thrashing* (aumento do número de cancelamentos).

## 4. Escalonadores Implementados

Este trabalho apresenta a implementação no nível de usuário de quatro escalonadores, usando dois modelos de escalonamento de transações. O dois primeiros foram completamente desenvolvidos em Haskell e os outros dois utilizando as primitivas da biblioteca LWC. O objetivo é mostrar a possibilidade da implementação de escalonamento de transações no nível de usuário e verificar qual o *overhead* imposto por cada nível de abstração (uso de Haskell puro e da bilioteca LWC).

Os dois modelos de escalonamento implementados são baseados nos trabalhos apresentados na Seção 3, um que serializa todas as transações conflitantes na mesma fila de execução de um único processador (modelo ATS) e o outro, que migra cada transação cancelada para a fila de tarefas do processador que está executando a transação que gerou

o conflito (modelo CAR-STM).

Para a implementação dos escalonadores, duas versões de bibliotecas de STM Haskell, também implementadas em alto nível, foram utilizadas, são elas a TL2 [Du Bois 2011], que usa versionamento tardio e detecção de conflitos tardia e a TINY [Duarte et al. 2015], que usa versionamento adiantado com detecção de conflitos também adiantada. Optou-se por usar estas bibliotecas porque as mesmas estão implementadas em Haskell, facilitando sua modificação para utilizá-las nos escalonadores. Nestas foram inseridas funções, para a comunicação com o escalonador desenvolvido, que informam onde e com quem as transações conflitam.

### 4.1. Escalonadores Desenvolvidos completamente em Haskell

A primeira implementação dos escalonadores foi completamente desenvolvida em Haskell, não utilizando nenhuma comunicação com o RTS do GHC. Nesta, o escalonador dispara, para cada *core* existente no processador, uma *thread* fixada. Essas *threads* possuem a função específica de executar transações oriundas de *threads* que tiveram suas transações canceladas, ou seja, as mesmas recebem transações e retornam o resultados destas para as *threads* de origem. Também, junto a cada *core*, são escalonadas *threads* Haskell, (*threads* essas gerenciadas pelo escalonador do RTS do GHC). Assim, para cada conflito que ocorre entre as transações, o escalonador identifica o conflito e, dependendo da regra de escalonamento definida (serialização/migração), direciona a transação conflitante para a *thread* auxiliar específica. Como exemplo de operação do escalonador, a Figura 1 mostra um cenário hipotético com um processador de dois *cores*, usando a técnica de migração.

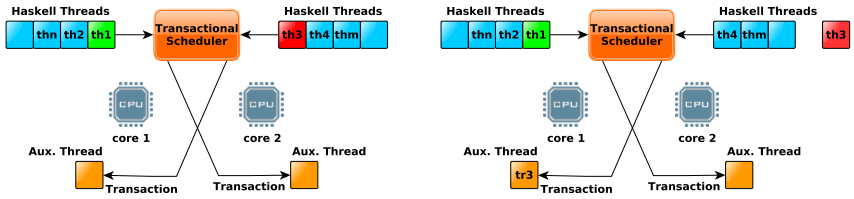



**Figura 1. Funcionamento do escalonador com migração**

No exemplo, a *thread* `th3` possui uma transação que conflita com outra presente na *thread* `th1`. `th3` informa ao escalonador que sua transação conflitou e solicita a migração de sua transação. A *thread* `th3` tem sua transação migrada para a *thread* auxiliar do *core 1*, representada na figura por `tr3`. Logo após, o escalonador coloca `th3` no final da fila de execução, ficando esta em uma espera por bloqueio, até que sua transação seja concluída. Cada *thread* auxiliar possui uma fila de tarefas que recebe as transações conflitantes oriundas do escalonador. As transações presentes nesta fila são resolvidas por regras do tipo FIFO, ou seja, a primeira transação a chegar é a primeira a ser executada. O escalonador que usa a técnica de serialização, migra toda a transação conflitante para a fila de tarefas do *core 1*, assim como no escalonador ATS.

O motivo pela opção da criação das *threads* auxiliares foi, o fato de evitar o custo de instanciação de uma nova *thread* para cada transação migrada e a impossibilidade de, a nível de usuário, migrar uma *thread* completa para a fila do escalonador de outro processador.

### 4.2. Escalonadores Desenvolvidos usando as primitivas do LWC

Os escalonadores desenvolvidos usando a biblioteca LWC são semelhantes aos anteriores, no entanto não são mais necessárias as *threads* auxiliares, pois no LWC é possível mover as transações e coloca-las diretamente na fila de execução de um outro *core*. Porém, quando se usa o LWC, o RTS não controla mais o escalonamento das *threads* de forma automática e este também deve ser realizado no nível de usuário. Assim, um escalonador para as *threads* no modelo *work-stealing*, parecido com o presente no RTS do GHC foi desenvolvido usando as primitivas do LWC onde, para cada *core* disponível, são criadas filas de tarefas que recebem *threads* para execução. Todo o código que deve ser executado por uma *thread* deve ser obrigatoriamente colocado em um `SCont`. Cada `SCont` somente pode ser colocado em execução ou movido de uma fila para outra através de funções específicas do LWC. O modelo de escalonamento de *threads* desenvolvido usou como base um modelo presente em [Sivaramakrishnan et al. 2016].

Já para o desenvolvimento dos escalonadores transacionais, também são utilizados `SConts` para receber as transações conflitantes. Quando uma transação conflita, esta é colocada em um `SCont` e, como no modelo implementado em Haskell, é migrada para a fila de tarefas do *core* que contém a transação que gerou o conflito. Porém, agora a transação migrada é colocada na mesma fila de tarefas das *threads*. A *thread* que contém a transação conflitante também é migrada para o fim da fila de tarefas do *core* em que se encontra e é colocada em uma espera em bloqueio. No modelo de serialização, assim como no implementado em Haskell, toda a transação conflitante é colocada na fila de execução de um mesmo *core*.

## 5. Metodologia

Os testes foram realizados comparando a utilização dos escalonadores com a versão original sem escalonamento de transações. Foram escolhidas as seguintes aplicações do STM-Haskell Benchmark [Perfumo et al. 2007] para verificar qual o comportamento dos escalonadores em diferentes níveis de contenção:

- **SI** - *Shared Int*: uma aplicação que usa um inteiro compartilhando entre n-transações e estas realizam operações de incremento no mesmo (alta contenção);
- **LL** - *Linked List*: Uma implementação de lista encadeada onde transações realizam operações de escrita e leitura, sendo 50% de escritas e 50% de leituras (contenção média);
- **BT** - *Binary Tree*: Uma estrutura de árvore binaria onde, operações de escrita e leitura são realizadas (baixa contenção).

Cada aplicação foi testada com os dois modelos de escalonamento usando as duas bibliotecas STM apresentadas anteriormente (TL2 e TINY):

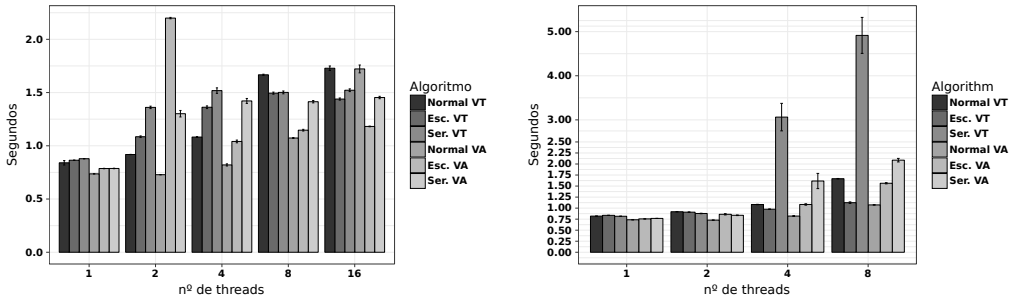
- **Normal VT**: sem utilização do escalonador, versionamento/detecção tardia (TL2)
- **Esc. VT**: usando a técnica de migração, com versionamento/detecção tardia
- **Ser. VT**: usando a técnica de serialização, com versionamento/detecção tardia
- **Normal VA**: sem escalonamento, versionamento/detecção adiantada (TINY)
- **Esc. VA**: usando migração e versionamento/detecção adiantada
- **Ser. VA**: serialização e versionamento/detecção adiantada

Cada aplicação foi executada 30 vezes para cada configuração em uma máquina com processador i7 (4 *cores* físicos e 4 lógicos), frequência de 3.4 Ghz, 8 GB de RAM e Ubuntu 14.04 64 bits. Foram usadas de 1 a 16 *threads*, com uma *thread* por *core* até 8 *threads*. A versão do GHC usada nos testes foi a 7.6.3. Para o LWC foi usada uma versão especial do compilador, a única que suporta as primitivas LWC[1].

## 6. Resultados

Todos os gráficos apresentados nesta Seção possuem o número de *threads* como eixo horizontal, com o eixo vertical variando dependendo do experimento.

Os tempos de execução da aplicação SI podem ser vistos na Figura 2. O tempo de execução com o uso dos escalonadores implementados em Haskell (Figura 2(a)) foram menores somente quando o número de *threads* é maior que o número de *cores*. Isto se deve ao fato de que, ao aumentar-se muito a concorrência, o *overhead* imposto pelo escalonador começa a ser suprimido pelo tempo de execução das transações. Os escalonadores usando LWC conseguem melhores resultados em relação ao tempo (Figura 2(b)), pois a implementação em baixo nível das primitivas do LWC reduzem o *overhead* dos escalonadores. Porém, com 8 *threads* as versões usando LWC tiveram um tempo de execução muito maior. Isso se deve ao fato de que como os escalonadores conseguem resolver a migração/serialização muito rapidamente, os mesmos não conseguiram evitar que novos conflitos ocorressem devido a novas transações disparadas em outros *cores*. Pode-se perceber isso principalmente no caso do uso de LWC, usando serialização com versionamento tardio (Ser. VT).

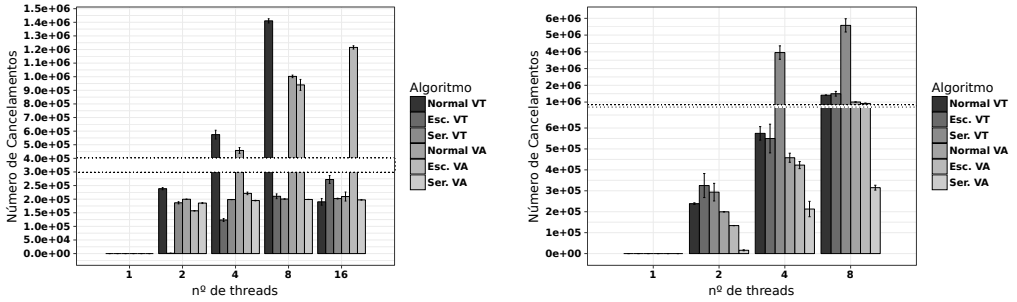



(a) Tempos dos escalonadores em Haskell (b) Tempos dos escalonadores em LWC

**Figura 2. Resultados da execução da aplicação SI.**

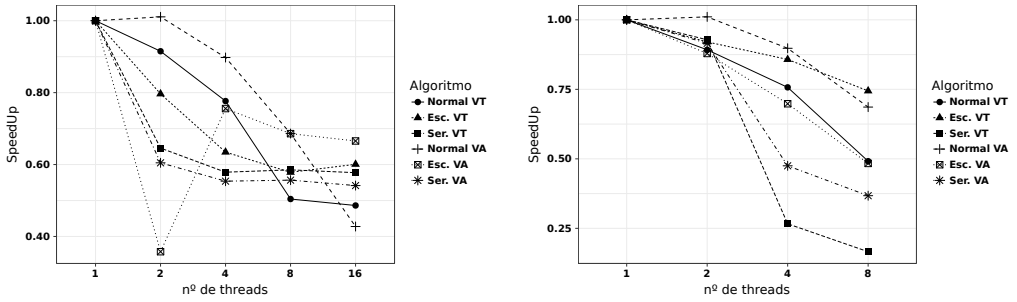
A Figura 3 apresenta o número de cancelamentos no eixo vertical. Na Figura 3(a), o uso das técnicas de migração e serialização reduziram o número de cancelamentos na aplicação como esperado. Os melhores resultados foram conseguidos com o uso de migração e versionamento tardio (Esc. VT.) na implementação em Haskell. Porém, ao aumentar muito a concorrência, as técnicas de escalonamento não conseguiram reduzir a quantidade de cancelamentos, pois mesmo migrando a transação conflitante para outro *core*, outras transações acabam conflitando devido ao alto nível de contenção da aplicação. O mesmo observa-se no caso do uso de LWC (Figura 3(b)), onde a única implementação que conseguiu reduzir o número de cancelamentos efetivamente foi a que utiliza serialização com versionamento adiantado (Ser. VA.). As demais versões não conseguiram reduzir a quantidade de cancelamentos de forma significativa.

[1] `https://github.com/ghc/ghc/tree/ghc-lwc2`




(a) Cancelamentos nos escalonadores em Haskell (b) Cancelamentos nos escalonadores em LWC

**Figura 3. Quantidade de cancelamentos da aplicação SI.**



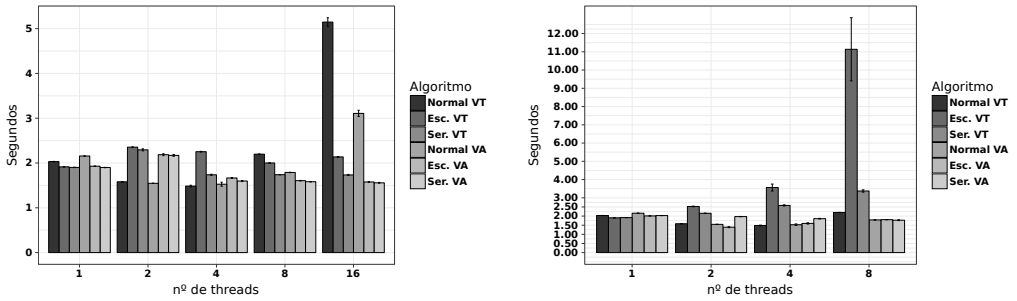

(a) Speedup dos escalonadores em Haskell (b) Speedup dos escalonadores em LWC

**Figura 4. Speedup Relativo da aplicação SI.**

Ao observar o gráfico de *speedup* relativo na Figura 4, observa-se que nenhuma das aplicações apresentou escalabilidade, devido à alta contenção característica desta aplicação. Porém, observa-se que as implementações usando LWC (Figura 4(b)) apresenta curvas menos acentuadas.

A aplicação LL tem seus tempos de execução apresentados na Figura 5. O uso das técnicas de escalonamento nas implementações em Haskell somente apresentaram resultados positivos a partir de 8 *threads* (Figura 5(a)), sendo que os melhores resultados foram obtidos com o uso de serialização de transações com 8 e 16 *threads* e versionamento adiantado. O *overhead* imposto pelos escalonadores novamente onerou muito o sistema transacional mas, apesar dos elevados tempos de execução, as técnicas de escalonamento conseguiram reduzir a quantidade de conflitos (Figura 6(a)). A versão usando serialização com versionamento adiantado (Ser. VA.) foi a que apresentou a média de cancelamentos mais constante. Ao percorrer uma lista encadeada para inserção/remoção, falsos conflitos podem ocorrer entre as transações [Sönmez et al. 2007], porque transações que modificam posições distintas da lista encadeada entram em falso conflito por uma delas estar alterando uma posição que esta presente no *read set* da outra. O uso da serialização apresentou a menor taxa de conflitos pelo fato de executar as transações em sequência, reduzindo o número de falsos conflitos.

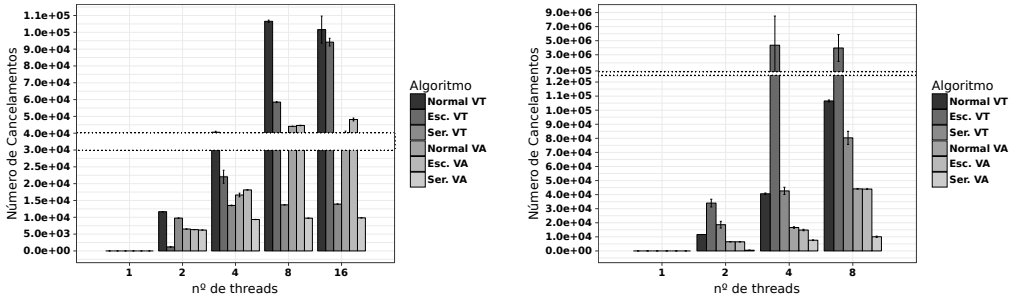
A implementação usando LWC conseguiu tempos de execução equivalentes às versões sem escalonamento (Figura 5(b)). Porém, como descrito no caso do SI, mesmo o escalonador movendo threads mais rapidamente, este não evita novos conflitos gerados por outras transações, o que acaba refletindo na taxa de cancelamentos (Figura 6(b)). No-




(a) Tempos dos escalonadores em Haskell (b) Tempos dos escalonadores em LWC

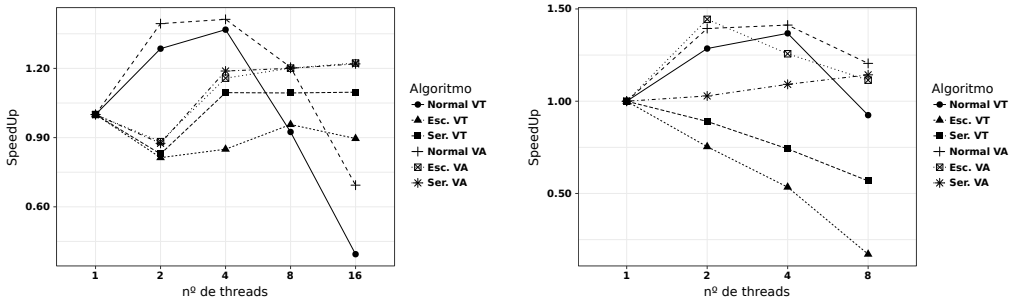
**Figura 5. Resultados da execução da aplicação LL.**

vamente a redução na quantidade de cancelamentos só ocorreu com o uso de serialização com versionamento adiantado.




(a) Cancelamentos nos escalonadores em Haskell (b) Cancelamentos nos escalonadores em LWC

**Figura 6. Quantidade de cancelamentos da aplicação LL.**



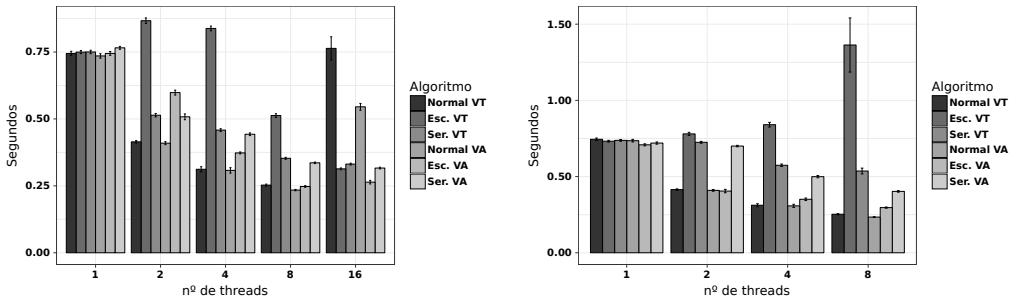

(a) Speedup dos escalonadores em Haskell (b) Speedup dos escalonadores em LWC

**Figura 7. Speedup Relativo da aplicação LL.**

Na Figura 7 pode-se observar que, quando a execução dos testes sem escalonamento de transações começaram a perder escalabilidade, o uso dos métodos de escalonamento de transações começaram a escalar. Isso se deve ao fato de o escalonador explorar melhor os recursos disponíveis, o que não acontece com poucas *threads*, onde o escalonador onera muito o sistema transacional.

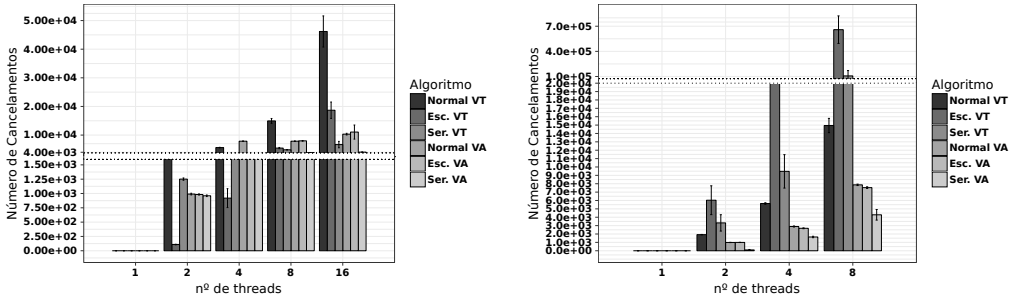
Nos resultados de tempo da aplicação BT (Figura 8), observa-se que novamente que os escalonadores somente apresentaram resultados positivos quando houveram mais

*threads* que *cores*. O comportamento negativo abaixo de 16 *threads* está relacionado ao fato de que, como as transações nesta aplicação são muito pequenas (bloco transacional com poucas operações e baixa contenção), os escalonadores acabaram inserindo um *overhead* proibitivo. Porém, quando observado o número de cancelamentos (Figura 9(a)), as técnicas de escalonamento conseguiram resultados positivos, sendo os melhores obtidos com o uso de 2 *threads* usando a técnica de migração com versionamento tardio (Esc. VT.). Quando o número de *threads* excede o número de *cores*, o melhor resultado é apresentado com o uso de serialização e versionamento adiantado (Ser.VA.). Nesta aplicação, o número de cancelamentos com o uso de escalonamento de transacões foi menor ou igual aos resultados obtidos sem a utilização do mesmo. Já na implementação usando LWC, não só os tempos de execução foram piores como a quantidade de conflitos aumentou (Figuras 8(b) e 9(b)). Somente com o método de serilaização com versionamento adiantado (Ser. VA) é que se conseguiu uma redução expressiva da quantidade de cancelamentos.




(a) Tempos dos escalonadores em Haskell (b) Tempos dos escalonadores em LWC

**Figura 8. Resultados da execução da aplicação BT.**



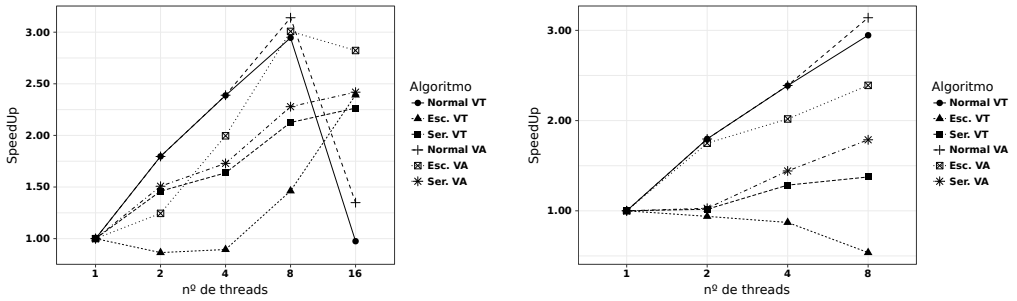

(a) Cancelamentos nos escalonadores em Haskell (b) Cancelamentos nos escalonadores em LWC

**Figura 9. Quantidade de cancelamentos da aplicação BT.**

Quanto à escalabilidade (Figura 10), observa-se que as implementações dos escalonadores em Haskell apresentaram melhores resultados (Figura 10(a)). Já a implementação usando LWC não apresenta escalabilidade efetiva, sendo que em alguns casos (Esc.VT.) a implementação nem mesmo apresenta *speedups* maiores que 1 (Figura 10(b)).

## 7. Conclusão

O principal objetivo deste trabalho foi apresentar a possibilidade da implementação de escalonadores de transações em um elevado nível de abstração, o que se demostrou viável e,




(a) Speedup dos escalonadores em Haskell (b) Speedup dos escalonadores em LWC

**Figura 10. Speedup Relativo da aplicação BT.**

apesar da implementação simples de decisão do escalonador, conseguiu-se alguns resultados positivos. A redução nos números de cancelamentos demostra que a implementação de escalonadores de transações na máquina virtual de Haskell pode gerar resultados interessantes e até mesmo reduzir o tempo de execução de algumas aplicações. Programar escalonadores em Haskell é simples, pois permite modificações mais rápidas sem a necessidade de recompilação de toda a máquina virtual para se adequar as novas regras de escalonamento.

Como pode-se observar nos resultados, o uso do escalonador de transações conseguiu reduzir o número de cancelamentos na maioria dos casos. Porém, o custo inserido pelo escalonador acabou impactando muito o sistema transacional, fazendo com que o tempo de execução fosse aumentado. O uso do LWC, por ter uma comunicação direta com o RTS do GHC reduz o *overhead* do escalonador. Porém, este efeito não se reflete na quantidade de cancelamentos. A redução do *overhead* do escalonador não evitou que novas transações viessem a conflitar devido ao eficiente gerenciamento. Assim, observa-se que técnicas mais refinadas para as decisões do escalonador de quando migrar ou serializar uma transação tornam-se necessárias.

Em trabalhos futuros, pretende-se implementar novas regras de escalonamento presentes na literatura como as vistas em [Nicácio et al. 2013] e [Di Sanzo et al. 2016], como também a modificação do escalonador do RTS do GHC para a inserção de um escalonador de transações.

## Referências

Di Sanzo, P., Sannicandro, M., Ciciani, B., and Quaglia, F. (2016). Markov chain-based adaptive scheduling in software transactional memory. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 373–382. IEEE.

Dolev, S., Hendler, D., and Suissa, A. (2008). Car-stm: Scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 125–134, New York, NY, USA. ACM.

Dragojević, A., Guerraoui, R., Singh, A. V., and Singh, V. (2009). Preventing versus curing: Avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 7–16, New York, NY, USA. ACM.

Du Bois, A. R. (2011). *An Implementation of Composable Memory Transactions in Haskell*, pages 34–50. Springer Berlin Heidelberg, Berlin, Heidelberg.

Duarte, R. M., Du Bois, A. R., Pilla, M. L., and Cavalheiro, G. G. H. (2015). Composable memory transactions with eager version management. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 2093–2098, New York, NY, USA. ACM.

Harris, T., Marlow, S., Jones, S. P., and Herlihy, M. (2008). Composable memory transactions. *Commun. ACM*, 51:91–100.

Li, P., Marlow, S., Peyton Jones, S., and Tolmach, A. (2007). Lightweight concurrency primitives for ghc. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 107–118, New York, NY, USA. ACM.

Li, P., Marlow, S., Peyton Jones, S., and Tolmach, A. (2013). Lightweight concurrency in ghc. Disponível em: https://ghc.haskell.org/trac/ghc/wiki/LightweightConcurrency. Acesso em: Agosto de 2017.

Maldonado, W., Marlier, P., Felber, P., Suissa, A., Hendler, D., Fedorova, A., Lawall, J. L., and Muller, G. (2010). Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–90, New York, NY, USA. ACM.

Marlow, S. (2013). *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming*. "O'Reilly Media, Inc.", CA, USA.

Nicácio, D., Baldassin, A., and Araújo, G. (2013). Transaction scheduling using dynamic conflict avoidance. *International Journal of Parallel Programming*, 41(1):89–110.

Perfumo, C., Sonmez, N., Cristal, A., Unsal, O., Valero, M., and Harris, T. (2007). Dissecting transactional executions in haskell. In *TRANSACT 07: Second ACM SIGPLAN Workshop on Transactional Computing*, Portland, Oregon, USA. ACM.

Peyton-Jones, S., Marlow, S., et al. (2017). Glasgow haskell compiler. Disponivel em https://www.haskell.org/ghc/. Acesso em: Agosto de 2017.

Rigo, S., Centoducatte, P., and Baldassin, A. (2007). Memórias transacionais: Uma nova alternativa para programação concorrente. In *Minicursos do VIII Workshop em Sistemas Computacionais de Alto Desempenho, WSCAD 2007*, Gramado,RS,Brasil. SBC.

Sivaramakrishnan, K. C., Harris, T., Marlow, S., and Peyton Jones, S. (2016). Composable scheduler activations for haskell. *Journal of Functional Programming*, 26:e9.

Sönmez, N., Perfumo, C., Stipic, S., Cristal, A., Unsal, O. S., and Valero, M. (2007). unreadtvar: Extending haskell software transactional memory for performance. *Trends in Functional Programming*, 8:89–114.

Yoo, R. M. and Lee, H.-H. S. (2008). Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 169–178, New York, NY, USA. ACM.

# Reducing Global Schedulers' Complexity Through Runtime System Decoupling

**Alexandre de Limas Santana**[1] **, Vinicius de Freitas**[1] **,**
**Márcio Castro**[1] **, Laércio Lima Pilla**[1,2] **, Jean-François Méhaut**[2]

[1]Federal University of Santa Catarina (UFSC), INE, PPGCC,
Florianópolis, Brazil

[2]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG.
Grenoble, France

alexandre.santana@posgrad.ufsc.br

***Abstract.*** *Global schedulers are components used in parallel solutions, specially in dynamic applications, to optimize resource usage. Nonetheless, their development is a cumbersome process due to necessary adaptations to cope with the programming interfaces and abstractions of runtime systems. This paper proposes a model to dissociate schedulers from runtime systems to lower software complexity. Our model is based on the scheduler breakdown into modular and reusable concepts that better express the scheduler requirements. Through the use of meta-programming and design patterns, we were able to achieve fully reusable workload-aware scheduling strategies with up to 63% fewer lines of code with negligible run time overhead.*

## 1. Introduction

The efforts to provide advances in parallel components, programming models and architectural design by the high performance computing community has led to solutions able to reach unprecedented computational landmarks. Unavoidably, future parallel components will be required to seamlessly benefit from improvements in multiple scientific fronts, preferably with low re-implementation efforts. Of special interest, within the context of dynamic applications, are global schedulers. They are specialized resource management components required to guarantee an adequate allocation of resources in a parallel solution. For that, they must be aware of parallelism intricacies in order to distribute the application workload among available processing elements (PEs).

Scheduling strategies may consider different information, like topology data [Hoefler et al. 2014], power consumption [Langer et al. 2015], or communication affinity [Jeannot et al. 2014, Cruz et al. 2015] to achieve their goals. As applications and scheduling strategies became more complex, runtime systems (RTS) such as *Charm++* [Kale et al. 2007], *OpenMP* [Chapman et al. 2008] and *OpenACC* [Wienke et al. 2012], have been applied as containers and frameworks to simplify the development of applications' parallel behavior and their relationship with global schedulers. As a result, these systems provide reusability to their components and provide a beneficial disconnection of application and scheduler code.

A RTS depends on software hooks to assemble components into a parallel solution and common approaches are strict APIs and code annotations. As consequence, RTS components are required to be adapted into the system's workflow and parallelism abstractions

(*e.g.,* threads, tasks, chares). However, scheduler components are composed by algorithms that each have their own functional requirements but does not depend on parallelism nor data abstractions. As such, by enforcing such traits through strict software hooks, global schedulers' software become bloated with adaptations, becoming more complex and less resilient to system modifications.

As novel parallel platforms are proposed, larger portions of RTSs are dedicated to exploit their particularities to maximize applications' performance. The exploitation of individual traits in parallel solutions leads to an increase in software complexity and component specialization, ultimately limiting their reusability [Dongarra et al. 2005]. Classic techniques such as aspect-oriented programming [Kiczales et al. 1997] and component-based software engineering [Heineman and Councill 2001] have been used to compose very large systems[1] based on reusable components. However, due to possible resource competition among parallel segments of code, these techniques can not be directly applied [Grossman et al. 2017]. We believe that the lack of studies in how to properly compose global schedulers with other components and RTSs will eventually result in bloated systems that are too complex to manage and challenging to port into future parallel programming models and tools.

To counteract this problem, we propose to exploit the sequential execution flow within RTSs to extract the scheduler component into a self-contained module. Isolated from its context, we are able to create a system-independent global scheduler model based on reusable and specialized concepts. As a result, this model can be used to implement scheduling policies that are less complex due to their isolation from specific technologies, RTSs and external scheduling-unrelated libraries. To achieve this results without high overheads, our proposal is based solely on modern language meta-programming facets and the *Adapter Design Pattern* [Vlissides et al. 1994] to link smaller segments of code into a global scheduler.

We evaluated our proposed model by comparing two re-implemented versions of scheduling policies from *Charm++* and *OpenMP* against the original versions native to these systems. Our global scheduler implementations are independent of RTS which requires them to be packed in external containers. Therefore, a global scheduler library called *Meta-programmed-Oriented Global Scheduler Library* (MOGSLib) was developed to portray a collection of reusable scheduling concepts that can be assembled to form system-specific global schedulers. The main focus of our experimental analysis is the comparison between identical scheduling strategies to evaluate discrepancies in performance, complexity and reusability. The proposed model was able to achieve the original behavior of the strategies in regards to application execution and strategy decision times, and schedule quality, while also lowering the number of lines of code (LoC) needed to express schedulers.

The remainder of this paper is structured as follows: Section 2 presents our global scheduler model. Section 3 describes our experiments. Section 4 discusses our results and analysis. Section 5 presents related work. Finally, Section 6 concludes this paper.

[1] Systems featuring millions of source lines of code.

## 2. System-Independent Scheduler Model

The implementation of a global scheduling algorithm depends on a multitude of factors and design choices aside from the scheduling policy such as: (i) data structure selection, (ii) third-party library usage, (iii) memory placement (*e.g.,* Data- or Object-Oriented Design) and (iv) target RTS. These decisions are important as they provide optimizations for a scheduler in regards to its target environment. However, each design decision further specializes the global scheduler implementation and limit its reusability as a whole.

Regardless of the different designs an implementation can portray, each runtime system and library also offer its own set of capabilities for schedulers (*e.g.,* load balancing database in *Charm++* [Kale et al. 2007]). The divergent interfaces among tools results in different implementations even when accessing a common functionality in distinct systems. As a consequence, modifications in scheduling policies are required when experimenting with different designs choices (RTS, data structures, libraries, etc.).

The contemporary relationship between runtime system and global schedulers is depicted in Figure 1. This figure expresses the dependencies (directed arrows) from software abstractions (inner boxes) to components within their context (outer boxes). As such, the problematic relationships are characterized by dependencies that spans out of the component's source context. Those relationships require unrelated code to be injected into a component, further binding its implementation and increasing its complexity as its code increases.

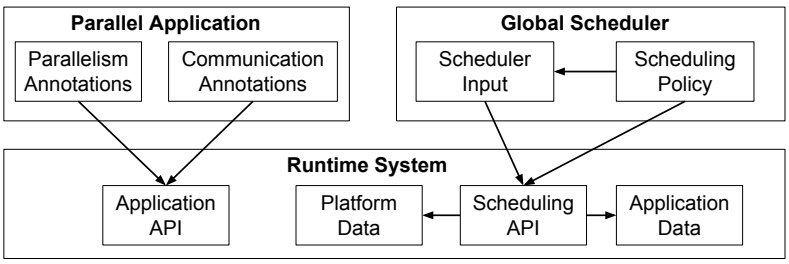



**Figure 1. Simplification of parallel components' dependencies.**

As a practical example of the aforementioned problem, we propose a scenario where a developer would implement a workload-aware scheduling policy. In this scenario, the scheduler requires the application data regarding its tasks' workload. An implementation of this policy in the *OpenMP* loop-scheduling interface would rely on user provided data as *OpenMP* has neither a method in its scheduling API to query the application workload, nor a method to inform it on the application API. On the other hand, *Charm++* presents data structures on its scheduling API that contain these and other data dynamically collected by the system. Regardless of RTS, the scheduling policy must query its required data from some source. As frameworks for developing global schedulers on these systems must be as flexible as possible, the same scheduling API and data structures are displayed for all policies to obtain their own set of data. This design forces scheduling policies to contain scheduling-unrelated code responsible for manipulating RTS structures to fulfill their functional requirements. Nonetheless, we envision that the exposure of a global scheduler's requirements through scheduling concepts is a solution that not only simplifies the development of schedulers but isolates the policy code from external functionalities.

## 2.1. Scheduling Concepts

Scheduling concepts are code segments which provide scheduling-unrelated functionalities that may be specialized for different RTSs or contexts. Different specializations of concepts must express its functionalities through functions with equal names/syntax. However, specializations must be sensible to its target environment in order to call the correct procedures needed to fulfill its functionality in the target RTS, platform or library.

The adapter design pattern is a software modeling technique that fits the aforementioned characteristics of scheduling concepts. As an example, a Unified Modeling Language class diagram (UML) is displayed in Figure 2 depicting classes representing both an abstract concept and the specialized concepts for the functionality of querying the application's workload. Both *Static Workload* and *Dynamic Workload* represents specialized concepts for accessing the application workload with different semantics, the former gathers static data and the latter dynamic workload through RTS data structures. Finally, both classes are implementations of the *Application Workload* interface, which defines a layer of functions for accessing the specialization's methods.

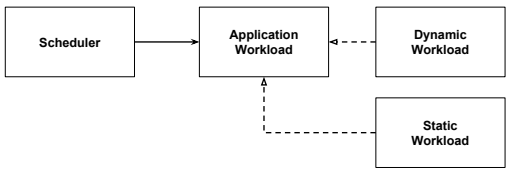



**Figure 2. Adapter pattern example.**

Although there is the possibility of implementing scheduling concepts solely through the adapter pattern, its usage incurs overheads as it relies on runtime type checks and virtual functions. We propose that scheduling policies can be better declared as partially defined template structures that depend on auxiliary data-types to implement functions that are sensitive to a given context. As template structures, scheduling concepts must be attached to a data-type that contains all the methods the concept requires during compilation. This way, both the scheduling concept and the specialized structure that provides its functionality are loosely linked until compilation, when data-types must be resolved. After the compilation, a direct association links both structures to construct a concrete scheduling concept that can provide its functionality without virtual calls nor dynamic type checks, avoiding their overhead.

To exemplify the proposed approach, we present a snippet in Figure 3, which showcases a concept that exposes the workload of an application. In the snippet, lines 1-6 portray the declaration of a concept that depends on a *Concrete* data-type (line 1) to properly provide its functionality through the *workloads* method (line 4). In lines 8-20, two classes that contain all the required methods to be a *Concrete* type for the *WorkloadConcept* are presented. The first class (lines 8-13) represents a class that packages the semantics to obtain the application's workload from the *Charm++* RTS. The *WorkloadCharm* is capable of querying the load balancing database contained in Charm++ (line 11) and obtain the workload data from its parallelism abstractions, represented by structures named *chares*. On the other hand, the second class (lines 15-20) portrays an auxiliary data structure to the *OpenMP* RTS that registers the application's workload data informed by the user (lines 17-19). With those definitions, a scheduling policy can make use of

two complete *WorkloadConcept*, one for the *Charm++* system and other for *OpenMP* depending of the *Concrete* template parameter. The advantage of this approach is that the concept is entirely responsible for gathering, storing, and manipulating the data structures for exposing its functionality. Ultimately, this design allows for a less complex scheduling strategy that requires no changes if the semantics of acquiring the application workload is changed.

```
1 template<typename Concrete>
2 class WorkloadConcept {
3 public:
4     Concrete data;
5     Load* workloads() { return data.workloads(); }
6 };
7
8 class WorkloadCharm {
9 public:
10    LDBD *charm_data;
11    inline Load* workloads() { return charm_data->chares.loads(); }
12    inline void set_data(LBDB *data) { charm_data = data; }
13 };
14
15 class WorkloadOmp {
16 public:
17    Load *_loads;
18    inline Load* workloads() { return _loads; }
19    inline void set_workloads(Load *loads) { _loads = loads; }
20 };
```

**Figure 3. Meta-programmed scheduling concept.**

Our approach of using modular and smaller scheduling concepts that compose a larger component displays advantages beyond alleviating the software complexity. Through the addition of dummy classes (like the one in the example) containing testing workloads to schedulers, it is possible to validate the scheduling policy independently from applications or RTSs. That way, we can find flaws in the implementation code on early stages of prototyping more precisely.

## 2.2. Global Scheduler Model

Similar to the process of declaring a scheduling concept, a global scheduler can be declared as a template structure that depends on one or multiple scheduling concepts. A *C++* example of this approach is depicted in Figure 4. Lines 1-6 serve as a declaration of the scheduler template model. The first line states that the `Scheduler` template will require an arbitrary number of parameters. The collection of parameters forms the *Concepts* type which is used in line 4 to construct the `work()` function signature. Moreover, as seen in lines 7-11, every scheduler policy has a specialized `work()` function signature that depends on its requirements rather than being defined by an external API.

## 2.3. The Role of MOGSLib

As concepts are defined as incomplete template structures, there must be concrete classes capable of providing the necessary functionalities for the concepts. These structures must be sensitive to the parallel solution's contexts (the target RTS, chosen libraries and execution environment) but they should remain decoupled from those. Our proposal is to encapsulate this software stack into a library that exposes a configuration interface that

```
template<typename ... Concepts>
class Scheduler {
public:
    virtual TaskMap work(Tuple<Concepts> concepts);
};

template<typename Workload, typename PEs>
class Greedy : public Scheduler<Workload, PEs> {
public:
    override TaskMap work(Tuple<Workload, PEs> concepts);
};
```

**Figure 4. Global scheduler model abstraction.**

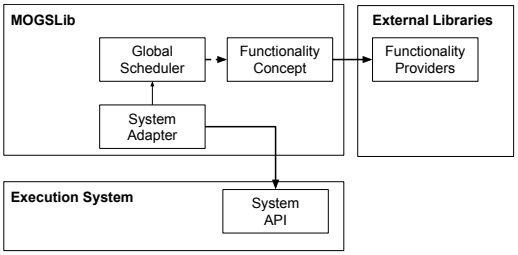



**Figure 5. *MOGSLib*'s components overview.**

can be easily composed into different contexts. Our implementation of such library is the *Meta-programming-Oriented Global Scheduler Library* (*MOGSLib*), an extensible and open-source library developed in *C++14*[2].

A global scheduler in *MOGSLib* is represented by a tuple $(P, F, S)$ where $P$ is the scheduling policy, $F$ is a set of concrete scheduling concepts and $S$ is a target context. Through this representation, it is possible for a given scheduling policy $P_i$ to generate different global schedulers by utilizing different concrete concepts or being targeted to a different context. As reusability is encouraged, previously developed functionalities can be used to compose new global schedulers, reducing the effort to develop them (coding, testing, etc.) and providing better reproducibility in scientific experiments.

A careful explanation of how the library operates is out of the scope of this work and interested readers are encouraged to check its public repository. Nonetheless, an overview of *MOGSLib* components and their interactions with external technologies is provided in Figure 5 where the components are denoted by labeled boxes and their dependencies by directed arrows. Objectively, *MOGSLib* is attached to the RTS and uses pre-compilation scripts to prompt the user for compilation and template parameters, ultimately generating a global scheduler that can interoperate with external RTSs and libraries (*e.g.,* Load balancers for *Charm++* and loop schedulers for *OpenMP*).

## 3. Experiments

In order to obtain precise information about overhead, we chose to implement centralized and greedy scheduling policies due to their predictable quasi-linear execution time. This class of schedulers is capable of making fast and precise decisions in smaller scenarios while displaying little variations in policy execution time when receiving the

[2]Available at: `https://github.com/ECLScheduling/lb-framework`

same input data. As such, greedy schedulers are ideal to observe small overhead variations between different global scheduler models.

In this work, we have chosen two greedy policies to implement in our model to compare against the native versions found in runtime systems. In this work, we selected two policies implemented within runtime systems to re-implement in our model Those policies are: (i) *Charm++*'s native greedy scheduler (*GreedyLB*), and (ii) a workload-aware loop scheduler implemented in *libGOMP* (*BinLPT*) [Penna et al. 2016]. The *GreedyLB* strategy iteratively pulls tasks from a task load *max-heap* and assigns to the top element of a PE load *min-heap* until there are no more unassigned tasks. The loop scheduler *BinLPT* groups adjacent iterations of a loop in up to $k$ task packs (defined by the user) and iteratively assign the heaviest group to the least overloaded PE. Although different, these strategies share the same scheduling concepts requirements, which also serve as an example of code reuse. The required scheduling concepts are: (i) application workload data retrieval and (ii) PE workload data retrieval.

### 3.1. Evaluated Metrics

Given the intent of analyzing a development model rather than a novel scheduling policy, our metrics have the intent of spotting differences between implementations and are enumerated as follows: (i) strategy decision time, (ii) application makespan, (iii) global scheduler lines of code (LoC) and (iv) number of reusable LoC. The time related metrics have the objective of measuring the overhead incurred by our model both in application makespan and in strategy decision time. The LoC metric serves as an indicator of the code complexity as fewer lines point to less complex segments of code [Nguyen et al. 2007].

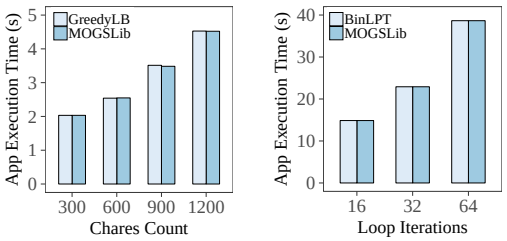
### 3.2. Software and Hardware

In order to compare our model against native implementations, our evaluation contemplates the *Charm++* v6.7 and *OpenMP* v4.0 runtime systems. The *MOGSLib* library, *Charm++* runtime and benchmarks were compiled with `g++` v5.4.0 with the following compilation flags: `-O3 -std=c++14`. Finally, the *libGOMP* library was compiled with its own *makefile* found in its aforementioned repository with the `gcc` compiler without additional flags.

To test the greedy strategy in *Charm++*, we chose the synthetic benchmark contained within the default *Charm++* package, *LB Test*, an iterative application that issues busy wait operations to simulate the workload. The benchmark was executed with different configurations in order to discover a parameter set that displays enough load imbalance to benefit from a global scheduler. To create this scenario, the following *LB Test* configurations were applied: (i) **Iterations**: 150, (ii) **Load balancing calls**: every 40 iterations, (iii) **Minimal task load**: 10 microseconds, (iv) **Maximum task load**: 3000 microseconds. To analyze the schedulers' scalability under different numbers of tasks, we ran this experiment with 300, 600, 900 and 1,200 tasks.

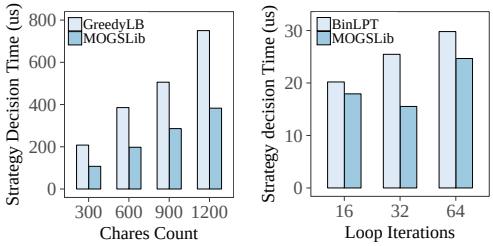
The tests using the *OpenMP* runtime system were executed over a modified version of *libGOMP*, the library responsible for providing the *OpenMP* directives implementations for open-source compilers. The modified version of *libGOMP* contains the required hooks for both MOGSLib and *BinLPT* and can be found in GitHub[3].

[3]Available at: `https://github.com/ECLScheduling/MOGSLib-libgomp-benchmark`




(a) *Charm++*. (b) *OpenMP*.

**Figure 6. Average application execution time.**




(a) *Charm++*. (b) *OpenMP*.

**Figure 7. Average schedule decision time.**

We test the *BinLPT* scheduler with the *SimSched*[4] synthetic benchmark. This application simulates CPU intensive kernels utilizing statistical distributions to generate random classes of workload that are later assigned to loop iterations. The parameters for the *SimSched* benchmark used in this paper were selected in conformity to the *BinLPT* paper. Their objective is to create a use case that better fits the use case of this global scheduler and are configured as follows: (i) **Distribution**: exponential, (ii) **Number of workload classes**: 12, (iii) **Kernel complexity**: quadratic. The necessary modifications to support the *BinLPT* in *OpenMP* system, *SimSched* benchmark details and the parameters to test workload-aware in it are explained in [Penna et al. 2016].

Our experiments were executed on the *Genepi*[5] cluster within the *Grid'5000* distributed environment. Furthermore, our *Charm++* tests were executed over 4 nodes whereas the *OpenMP* tests used only one.

## 4. Results

The results of our experiments in regards to total application execution time are provided in Figure 6(a) for the *Charm++* system and Figure 6(b) for *OpenMP*. Each bar in those figures represents the arithmetic mean of $50$ application runs. In order to further analyze the application execution time, we executed two-tailed *t-student* tests to check if both scheduler versions (native and *MOGSLib*) were originated from a distribution with the same parameters. The confidence interval was set to $5\%$ and $p$-values are displayed in Table 1.

**Table 1. Application execution time parametric tests results.**

| **Charm++ environment** | |
|---|---|
| Task Count | $p$-value |
| 300 | 0.96 |
| 600 | 0.55 |
| 900 | 0.18 |
| 1200 | 0.43 |

| **OpenMP environment** | |
|---|---|
| Loop Iterations | $p$-value |
| 16 | 0.45 |
| 32 | 0.30 |
| 64 | 0.38 |

As the experiments generated $p$-values that surpass the confidence interval of $0.05$, we cannot reject the null hypothesis that both distributions are equal. This conclusion

[4]Available at: `https://github.com/lapesd/libgomp-benchmarks`

[5]*Genepi* complete system specification at: `https://www.grid5000.fr/mediawiki/index.php/Grenoble:Hardware#genepi`

implies that both native and the *MOGSLib* schedulers are able to perform equally on the different tested applications, runtime systems and application size.

### 4.1. Schedule Decision Time Analysis

In order to analyze both implementations in detail, we analyzed the time taken in order to decide the task mapping on each of the aforementioned scenarios through Figures 7(a) and 7(b). The bars depicted in these figures represent the arithmetic mean of the time each strategy took to decide a schedule. Therefore, in *Charm++* experiments, each bar represents $150$ data points ($3$ schedules computed for each of the $50$ runs). Moreover, in *OpenMP* experiments, each bar is composed by $50$ data points, as the scheduler is called before the loop and there is only one loop per application kernel.

The scheduler decision time data depicted in Figures 7(a) and 7(b) presented standard deviations smaller than $1\%$. Furthermore, our model portrayed decision times that were $45\%$ and $18\%$ faster on *Charm++* and *OpenMP* systems, respectively. However, for these tests, the impact of the scheduler decision time is negligible due to its scale (microseconds) in comparison to the application makespan (in seconds). This overhead would be more important in scenarios with more tasks or higher rescheduling frequency. With a small time scale, differences between implementations were bound to happen and their origin is related to different parameters between schedulers. In *Charm++*, generic data structures were used in our model in contrast to the ones used by the *Charm++* scheduler. Moreover, the only difference between the schedulers in *OpenMP* was the compiler used to generate the *MOGSLib* global scheduler. While *libGOMP* is compiled through `gcc`, *MOGSLib* used `g++` to compile and link its scheduler into *OpenMP*.

### 4.2. Complexity Analysis

To better analyze our model in contrast to native implementations, we break our approach in different components that form the global scheduler. Each component's lines of code count is displayed in Table 2, with the last column designated to portray where the component can be reused within other parallel solutions.

**Table 2. MOGSLib components' LoC.**

| **Component** | *BinLPT* | *GreedyLB* | **Reusable on** |
|---|---|---|---|
| Scheduling Policy | 37 | 30 | Runtime Systems |
| Runtime System Adapter | 60 | 22 | Scheduling Policies |
| Concepts | 30 | 40 | Scheduling Policies |

The native versions of *BinLPT* and *GreedyLB* are composed, respectively, by $84$ and $81$ LoC. Our version of those same schedulers are composed by $127$ and $97$ LoC respectively when accounting for the sum of components that assemble the scheduler. However, every component can be reused in at least one scenario as stated in the last column in Table 2. In the scenario where a new scheduler is proposed and the concepts and RTS adapter have been previously developed, the only required implementation is the scheduling policy. This scenario is not uncommon as the concepts can be reused and novel policies are encouraged to use existing system adapters and scheduling concepts. As such, when analyzing solely the size of the scheduling policy code, our model attained up to $63\%$ size reduction in comparison to the native versions.

The segmentation into concepts is advantageous as each concept portrays a single role within the scheduler in contrast to current implementations found in RTSs. Despite resulting in a larger sum of LoC, this approach enables the composition of functionalities implemented by developers of different expertise. That way, developers can rely on reusing concrete concepts rather than re-implementation, leaving the burden of assembling the functionalities to be taken care by libraries such as *MOGSLib*.

## 5. Related Work

Schedulers are a relevant topic in real-time systems due to application diversity which, ultimately, demands specialized policies. Classically, schedulers are kernel components and, as such, developed policies are tied to a specific patch and OS. Furthermore, to enable higher customization and enhance the range of scheduling policies, Asberg [Åsberg et al. 2012] and Mollison [Mollison and Anderson 2013] moved the policies implementations from kernel space to the user space. Through the use of an abstraction layer inside the kernel space, their work showcased policies developed over higher level abstractions with acceptable overhead in hard real-time systems. Ultimately, both proposals used different techniques to decouple the schedulers from the kernel primitives. However, in concordance with our proposal, they also proposed the extraction of the scheduler component into its own module.

In respect to providing modularity to scheduler components, HPC runtime systems like *Charm++*, *OpenMP* and *OpenACC* provide a simple way for decoupling application and scheduler code. Either through annotations, abstractions or language modifications, these systems allow resource management hooks in the applications life-cycle, thus enabling reuse and portability of scheduling policies among applications within the same system. Nonetheless, scheduler implementations are yet limited to a specific abstraction set, contain system-specific code and their algorithms are often scattered through different segments in the RTS. Ultimately, our approach intends to benefit from these RTSs while alleviating the scheduling implementation problems associated with their usage.

Grossman et al. [Grossman et al. 2017] proposed that, through a better description of a component's connections, the composability of parallel libraries can be achieved through modern language facets such as lambda functions and asynchronous calls. Their work is oriented towards the development of a novel runtime system that showcases component connections using lambda functions. Nonetheless, our work shares the use of modern language traits and better description of components' connections. However, we apply meta-programming and ultimately target system-independence rather than proposing a novel system architecture.

Bigot [Bigot et al. 2012] defined parallel solutions as an assembly of components that can be interconnected and swapped to provide performance portability. The work is based on performance portability through modularity and applies driver components to resolve the intricacies of specific systems and technologies. Despite the similarities, our work diverges in the technique applied and context within the parallel solution. Ultimately, their approach presents a component-based model for applications that utilizes a small runtime to link components together, whereas our work presents an attachable scheduler model that can be linked to runtime systems through a library in compilation time.

## 6. Conclusions

In order to ease the development of global schedulers and enable simpler scheduling policy implementations, this paper contributes to the topic by exposing a global scheduler model capable of describing its requirements through meta-programmed scheduler concepts. We discuss the impacts of reusability, modularity and software complexity on parallel components while we present a novel library, *MOGSLib*, as a technical contribution for global scheduler developers. The evaluation of the proposed model is made through synthetic benchmarks executed on *Charm++* and *OpenMP* systems analyzing two distinct workload-aware scheduling policies, *GreedyLB* and *BinLPT*.

Our results (presented in Section 4) displayed that the model incurs in negligible variations in scheduler quality and application makespan. Additionally, at the best case scenario, our approach can reduce the number of LoC needed to develop a new global scheduler by up to 63% when reusing previously implemented scheduling concepts. The possibility of component reusability is beneficial as it enables code replayability without development efforts and less complex software segments. Even in the worst case scenario, where concepts must be implemented from scratch, this approach allows for a better prototyping phase that can adhere to test-oriented development due to each module being responsible for a single role in the system.

In regards to future works, we intend to further study the proposed model, experimenting its adoption into different scheduling policies. Of special interest are strategies that take into account informations about the platform topology, task affinity and memory hierarchy. As more functionalities are required for different policies, we aim to enhance *MOGSLib* with more concrete scheduling concepts and system adapters to provide developers with more tools and options to compose simple and reusable global schedulers.

## ACKNOWLEDGMENT

This work was partially supported by the Brazilian Federal Agency for the Support and Evaluation of Graduate Education (CAPES) and by the Brazilian Council of Technological and Scientific Development (CNPq), project grant 401266/2016-8.

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by INRIA and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

## References

Åsberg, M., Nolte, T., Kato, S., and Rajkumar, R. (2012). Exsched: An external CPU scheduler framework for real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, pages 240–249. IEEE.

Bigot, J., Hou, Z., Pérez, C., and Pichon, V. (2012). A low level component model enabling performance portability of HPC applications. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 701–710. IEEE.

Chapman, B., Jost, G., and Van Der Pas, R. (2008). *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press.

Cruz, E. H., Diener, M., Pilla, L. L., and Navaux, P. O. (2015). An efficient algorithm for communication-based task mapping. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 207–214.

Dongarra, J., Sterling, T., Simon, H., and Strohmaier, E. (2005). High-Performance Computing: Clusters, Constellations, MPPs, and Future Directions. *Computing in Science and Engineering*, 7:51–59.

Grossman, M., Kumar, V., Vrvilo, N., Budimlic, Z., and Sarkar, V. (2017). A pluggable framework for composable HPC scheduling libraries. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 723–732. IEEE.

Heineman, G. T. and Councill, W. T. (2001). Component-based software engineering. *Putting the pieces together, addison-westley*, page 5.

Hoefler, T., Jeannot, E., and Mercier, G. (2014). *An Overview of Topology Mapping Algorithms and Techniques in High-Performance Computing*, pages 73–94. John Wiley & Sons, Inc.

Jeannot, E., Mercier, G., and Tessier, F. (2014). Process placement in multicore clusters: Algorithmic issues and practical techniques. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):993–1002.

Kale, L. V., Bohm, E., Mendes, C. L., Wilmarth, T., and Zheng, G. (2007). Programming petascale applications with Charm++ and AMPI. *Petascale Computing: Algorithms and Applications*, 1:421–441.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer.

Langer, A., Totoni, E., Palekar, U. S., and Kalé, L. V. (2015). Energy-efficient computing for HPC workloads on heterogeneous manycore chips. In *Proceedings of Programming Models and Applications on Multicores and Manycores*. ACM.

Mollison, M. S. and Anderson, J. H. (2013). Bringing theory into practice: A userspace library for multicore real-time scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 283–292. IEEE.

Nguyen, V., Deeds-Rubin, S., Tan, T., and Boehm, B. (2007). A SLOC counting standard. In *Cocomo ii forum*, volume 2007, pages 1–16. Citeseer.

Penna, P. H., Castro, M., Freitas, H. C., Broquedis, F., and Méhaut, J.-F. (2016). Design methodology for workload-aware loop scheduling strategies based on genetic algorithm and simulation. *Concurrency and Computation: Practice and Experience*.

Vlissides, J., Helm, R., Johnson, R., and Gamma, E. (1994). Design patterns: elements of reusable object-oriented software.

Wienke, S., Springer, P., Terboven, C., and an Mey, D. (2012). OpenACC—first experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer.

# Video7: Uma Arquitetura para Armazenamento e Recuperação de Arquivos de Áudio e Vídeo

**Vanderson S. de O. L. Sampaio**[1], **Douglas D. J. de Macedo**[2], **André Britto**[3]

[1]Departamento de Informática e Estatística (INE)
Univ. Federal de Santa Catarina – (UFSC) – Florianópolis, SC – Brasil

[2]Departamento de Ciência da Informação (CIN)
Univ. Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brasil

[3]Departamento de Computação (DCOMP)
Univ. Federal de Sergipe (UFSC) – São Cristóvão, SE – Brasil

vandersons.sampaio@gmail.com, douglas.macedo@ufsc.br, andre@dcomp.ufs.br

***Resumo.*** *O número crescente de áudio e vídeo digital faz surgir a necessidade de ferramentas apropriadas para armazenamentos e gerenciamentos desses tipos de dados, e uma opção para o armazenamento, são os bancos de dados não relacionais (NoSQL). A diversidade de sistemas existentes, provoca o interesse em propor uma arquitetura para gestão desse conteúdo nos diferentes tipos de bancos de dados. Devido a essa necessidade, este trabalho propõe a arquitetura Video7 para armazenamento e recuperação de arquivos de áudio e vídeo de fluxo contínuo armazenados em bases de dados não relacionais. Com base na arquitetura, foi implementada uma ferramenta que utiliza os bancos de dados Apache HBase, Apache Cassandra, Project Voldemort, Redis e MongoDB, e os submete a rotinas estressantes. O objetivo das rotinas de estresse é aferir os tempos das inserções e das consultas, além de suas taxas de transferências em resposta às solicitações a um servidor de mídia. O banco de dados Redis apresenta melhor desempenho nas rotinas submetidas, enquanto o Project Voldemort e o Apache Cassandra apresentam um desempenho inferior aos demais bancos de dados.*

***Abstract.*** *The increasing number of both digital audio and videos, brings up the necessity of appropriate tools for the storagement and management of those kind of data. As options for the storagement, there are the non-relational databases (NoSQL). The diversity of existing systems provokes the interest in proposing an architecture for the management of that content, in different types of databases. Due to that need, this work proposes the Video7 architecture for storing and retrieving both audio and video streaming files stored in non-relational databases. Based on the architecture, it was implemented a tool that makes use of the Apache HBase, Apache Cassandra, Project Voldemort, Redis and MongoDB databases, and is subjected to stressful routines. The purpose of stress routines is to measure insertion and queries times, in addition to their transfer rates in response to requests to a media server. Redis database presents better performance in the submitted routines, while Project Voldemort and Apache Cassandra perform poorly than other databases.*

## 1. Introdução

A todo instante novos dados digitais são gerados na Web. Esses dados possuem uma infinidade de tipos (e.g., numéricos, textuais, imagens) e formatos (e.g., MP3, HTML, PDF). Os dados multimídia (i.e., imagens, vídeos, áudios) representam uma parcela significativa deste montante. Plataformas como YouTube[1] e Vimeo[2], onde os utilizadores são capazes de armazenar, reproduzir e compartilhar seus vídeos, corroboram para esse crescimento [da Costa 2010]. A proliferação do conteúdo multimídia traz consigo a necessidade de melhores métodos para gestão dos dados [Cheng et al. 2008]. Os métodos para gestão dos dados tornam-se viáveis com a utilização de alguns recursos como: sistemas de arquivos distribuídos [de Macedo et al. 2011]; bancos de dados não relacionais (NoSQL) [Li et al. 2012]; abordagens híbridas [de Macedo et al. 2015]; entre outros.

Uma maneira simples de armazenar áudio e vídeo é utilizar servidores de arquivos [Suchomski et al. 2005]. Os servidores de arquivos transferem a responsabilidade do armazenamento dos dados para o sistema operacional. Desta maneira a gestão dos dados, como as políticas de segurança, ocorrem nos diretórios ou nos próprios arquivos, algo pouco positivo, dependendo da aplicação. A disponibilidade de acesso aos dados é um fator de risco. Eventuais falhas no servidor de arquivos podem acarretar falhas na aplicação. Por fim, utilizar servidores de arquivos pode ser custoso, principalmente a falta de ferramentas para interoperabilidade direta com as aplicações (e.g. aplicações Web).

Banco de dados é uma opção para o armazenamento e gerenciamento dos dados multimídias. Porém a variedade de modelos (e.g., SQL, NoSQL, NewSQL) e ferramentas (e.g., PostgreSQL, Redis, MariaDB) de bancos de dados dificulta a proposta de uma solução. [Rebecca and Shanthi 2016, Assis et al. 2017] comparam o desempenho de armazenamento, consulta e atualização de dados multimídia nos modelos de banco de dados relacional (SQL) e não relacional (NoSQL). Estes são apenas dois dos vários estudos que abordam este problema.

Buscando auxiliar na seleção do modelo e ferramentas de bancos de dados este artigo apresenta a arquitetura *Video7*. A arquitetura apresenta flexibilidade tanto para os tipos de dados quanto para os modelos de banco de dados que se pretende utilizar. Por fim, é possível realizar comparações de determinadas métricas de desempenho entre bancos de dados. Para validar a arquitetura são realizados experimentos e inserção e recuperação com os diferentes modelos de bancos de dados NoSQL.

As principais contribuições desse artigo são: i) arquitetura de armazenamento de áudio e vídeo em variados bancos de dados; ii) análise experimental de desempenho dos bancos de dados no processo de inserção e recuperação dos dados; iii) comparação entre diferentes tipos de bancos de dados para dados de áudio e vídeo armazenados. Além dos benefícios de gestão dos dados multimídias, eventuais extensões da arquitetura poderão permitir, de maneira eficiente, a extração de características dos dados.

O restante do artigo está dividido da seguinte maneira. A Seção 2 apresenta os trabalhos relacionados, no sentido de posicionar esta pesquisa aos trabalhos correlacionados. A Seção 3 apresenta a arquitetura Video7, demonstrando seu fluxo de dados e os componentes da mesma. A Seção 4 os cenários de testes da proposta são apresentados. A

[1]http://www.youtube.com
[2]http://www.vimeo.com

Seção 5 os experimentos realizados são expostos e detalhados, divididos pelos processos de inserção e recuperação. Por fim, a Seção 6 apresenta as conclusões e as sugestões de trabalhos futuros deste estudo.

## 2. Trabalhos Relacionados

Na busca por trabalhos relacionados, não foi identificada nenhuma proposta de arquitetura específica para armazenamento de vídeos em bancos de dados não relacionais, desta forma, aqui são apresentados trabalhos que apresentam abordagens similares.

Em [Li et al. 2010] é apresentado o sistema analítico de gerenciamento de vídeos intitulado `HydVideo`. O HydVideo utiliza um pacote de expansão aplicado sobre o banco de dados relacional. A utilização desse pacote permite análises mais eficientes mesmo com a rigidez do modelo relacional. Bancos de dados NoSQL são utilizados como alternativa a rigidez imposta no modelo de dados relacional.

Trabalhos como em [Rebecca and Shanthi 2016, Assis et al. 2017] realizam comparações entre os modelos de dados relacionais e os NoSQL. Em [Rebecca and Shanthi 2016] são realizadas comparações entre os bancos de dados MySQL e MongoDB no armazenamento de imagens médicas. A pesquisa conclui que o banco de dados MongoDB apresenta melhores desempenhos de inserção e de consulta. Em [Assis et al. 2017] dados de imagem e vídeo são experimentados na ferramenta YCSB[3] para os bancos de dados MySQL, Redis, Apache Cassandra e MongoDB. Conclui-se com esses experimentos que bancos de dados NoSQL possuem melhores medidas de tempo em comparação com os bancos de dados SQL.

Os trabalhos que abordam dados multimídia de áudio e vídeo aparentam não se preocupar com a gestão do armazenamento dos dados. As análises identificadas limitam-se a comparar tempos de inserção e recuperação dos dados. As análises são realizadas, muitas vezes, com a utilização de ferramentas que não possuem tal finalidade ou por técnicas pouco sofisticadas.

## 3. Arquitetura Proposta

Esta seção apresenta a arquitetura Video7 através da explanação do fluxo dos dados em uma visão geral e do detalhamento dos componentes utilizados para sua concepção. Arquitetura Video7 foca as aplicações Web, e possui uma capacidade de ampliação dos tipos de arquivos utilizados (e.g., mp4, avi, mjpg) e dos modelos de bancos de dados suportados. O objetivo principal é proporcionar uma arquitetura flexível e extensível que seja capaz de ser estendida para atender novos formatos de dados multimídia, e que seja capaz de agregar novos tipos de bancos de dados ainda não integrados.

O fluxo dos dados é apresentado na Figura 1. A arquitetura separa novas requisições de acordo com a solicitação do usuário. As solicitações podem ser de *Escrita* ou de *Leitura*. As requisições de escrita trazem consigo o arquivo para armazenamento, os metadados deste arquivo (e.g., resolução do vídeo, qualidade do áudio) e a base de dados de destino para persistir os dados. Essas informações são passadas para o *Seletor* que destina os dados e os metadados do áudio/vídeo para o servidor em que o banco de dados está localizado. Nesse servidor, ocorre a operação de leitura/escrita do arquivo.

[3]https://github.com/brianfrankcooper/YCSB/wiki

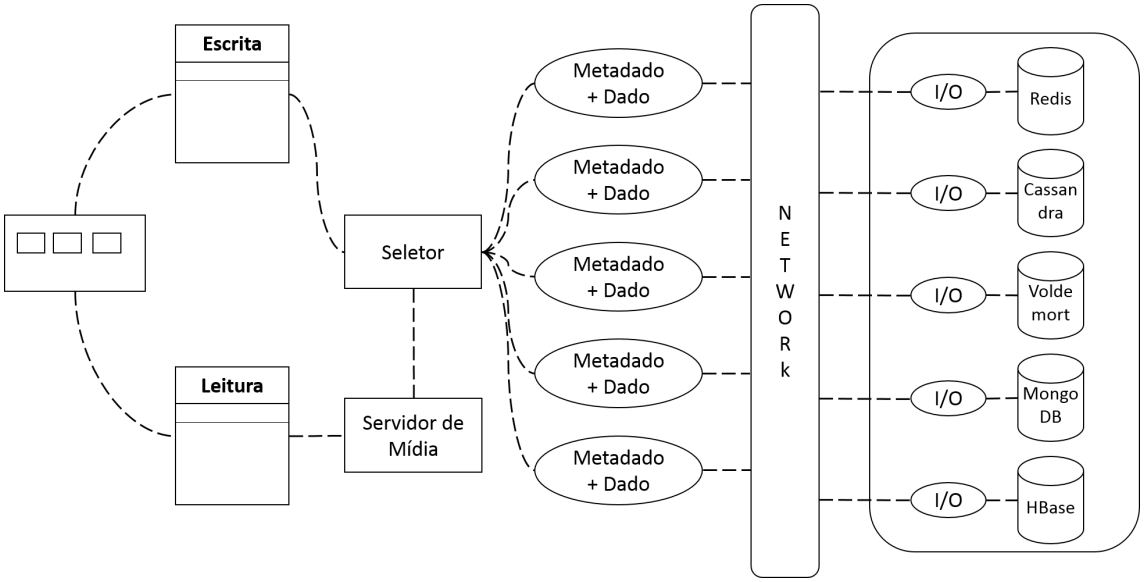



Figura 1: Fluxo de Dados da Arquitetura

As requisições de leitura necessitam do auxílio de um *Servidor de Mídia*. O servidor de mídia é responsável por controlar o fluxo de envio do vídeo com a aplicação. O servidor de mídia se comunica com o *Seletor* da arquitetura à medida que necessita dos dados do arquivo. As requisições de leitura são compostas, por exemplo, pela resolução do vídeo que se deseja recuperar e a base de dados que será consultada.

A Figura 2 exibe a arquitetura proposta. A arquitetura pode ser dividida em sete camadas, sendo elas: Tipos de Usuários, Tipo de Arquivos, Aplicação, Bibliotecas, Arquivos Multimídias, Distribuição dos Dados e Armazenamento. A camada *tipos de usuários* é uma interface onde um usuário poderá solicitar armazenamento de um novo vídeo ou a recuperação de um vídeo existente em alguma base de dados, através do acesso como administrador ou público, respectivamente. A camada *tipos de arquivos* realiza a codificação/decodificação do arquivo que se deseja armazenar/reproduzir. Ainda na camada tipos de arquivos é possível incluir novos tipos e formatos de arquivos.

A camada *aplicação* é responsável por diferenciar a operação desejada. Requisições oriundas da interface administrador são atendidas na classe escrita, e as requisições da interface pública são atendidas na classe leitura. A camada *bibliotecas* disponibiliza uma API (*Application Programming Interface*) que permite a comunicação com o seletor da base de dados ou com o servidor de mídia. Requisições de leitura são destinadas ao servidor de mídia que realiza a comunicação com o seletor. As requisições de escrita se comunicam diretamente com o seletor.

A camada *Arquivos Multimídia* é composta por metadados e dados que contém informações como resolução do vídeo, tempo de duração, situação (e.g., iniciado, parado), dentre outras. O acesso a essa camada ocorre através da classe seletor da camada bibliotecas, sendo o seletor responsável por definir a persistência ou consulta dos dados.

A camada *Distribuição dos Dados* ajusta os dados de acordo com o modelo de cada base de dados. Essa camada também trata das particularidades de cada base de dados e das informações referentes às configurações do servidor que hospeda os bancos de

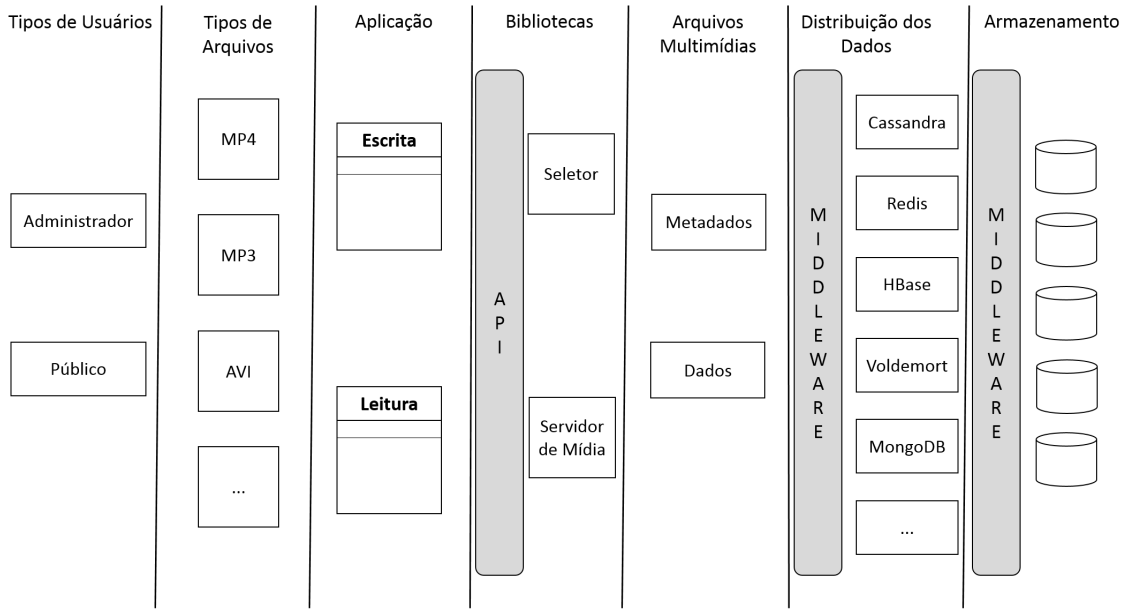



Figura 2: Componentes da Arquitetura Video7

dados. Por fim, na camada *Armazenamento* os dados são persistidos ou consultados no banco de dados selecionado previamente. A existência das camadas distribuição dos dados e armazenamento permite uma flexibilidade à arquitetura, com expansão dos bancos de dados e modelos de bancos de dados utilizados.

## 4. Experimentos

Esta seção descreve os experimentos aplicados para validar a arquitetura proposta. A Seção 4.1 afirma os objetivos dos experimentos. A Seção 4.2 retrata os cenários de testes abordados. A Seção 4.3 detalha as configurações dos experimentos.

### 4.1. Objetivos

Os experimentos objetivam aferir o desempenho de bancos de dados NoSQL no armazenamento de arquivos de áudio e vídeo. A arquitetura Video7 compara as categorias chave-valor, tabular e documento de bancos de dados NoSQL. Os bancos de dados são submetidos à rotina de estresse de inserção e recuperação. A utilização de outros modelos de bancos de dados e armazenamento distribuído não são abordados nesse artigo devido a limitações de espaço do artigo e serão investigados em trabalhos futuros.

### 4.2. Cenários de Testes

Os experimentos avaliam os tempos de inserção e recuperação de vídeos e a taxa de transferência dos arquivos no navegador do usuário em bancos de dados NoSQL. Experimentos preliminares demonstraram que após recuperar $20\%$, em média, do arquivo multimídia a taxa de transferência apresenta constância. São utilizados cinco bancos de dados NoSQL.

O artigo aborda três cenários de testes, são eles: solo, baixo e alto. O cenário solo é um cenário ausente de concorrência. Os cenários baixo e alto simulam ambientes concorridos de até dez e vinte acessos simultâneos, respectivamente. Os valores de acesso simultâneos foram definidos de acordo com o ambiente computacional utilizado para realizar os experimentos.

### 4.3. Configurações do Experimento

Cinco bancos de dados NoSQL de três categorias diferentes são abordados nesses experimentos. A categoria chave-valor é representado pelos bancos de dados Redis e Project Voldemort. O Apache Cassandra e o Apache HBase representam a categoria tabular de banco de dados NoSQL. Por fim, o banco de dados MongoDB é o único representante da categoria de bancos de dados orientado a documentos.

O arquivo de vídeo empregado nas avaliações possui diferentes qualidades de resolução e tamanho físico. As resoluções empregadas foram: $240p$, $360p$, $480p$, $720p$, $1080p$ e $4k$, e os tamanhos físicos dos arquivos foram, em *megabytes*, $13,7$, $19,4$, $38,4$, $65,3$, $112$ e $671$, respectivamente. As medições são repetidas $25$ vezes para cada cenário, banco de dados e resolução de vídeo. Ao final das repetições são calculados as médias e os desvios padrões das medidas de desempenho.

A arquitetura Video7 foi implementada na linguagem de programação Java na sua versão 8. Os testes foram realizados em um ambiente de nuvem pública da *Digital Ocean*. Durante o processo foi utilizado uma máquina virtual com as seguintes configurações: Arquitetura x86; Processador Intel(R) Xeon(R) CPU E5-2630L 2GHz de 4 núcleos; Memória RAM com capacidade de 8GB; Memória secundária SSD com capacidade de 20GB; Sistema Operacional Ubuntu 16.04 x64. A máquina virtual possui o Apache Tomcat 7, um servidor de mídia e os bancos de dados descritos nesta seção.

## 5. Resultados e Discussões

Esta seção foi dividida em três subseções. A Seção 5.1 apresenta os resultados para a operação de inserção. A Seção 5.2 expõe os valores dos tempos e das taxas de transferência calculadas na operação de recuperação. A Seção 5.3 apresenta uma discussão sobre os resultados obtidos. Os valores de tempo e taxa de transferência são calculados em milissegundos e megabytes por segundo, respectivamente. O teste estatístico Kruskal-Wallis confirmou a existência de diferença significativa entre os dados testados em $89\%$ dos casos.

### 5.1. Inserção

Alguns bancos de dados possuem restrições para armazenamento dos dados. O banco de dados Redis não permite armazenar dados com tamanho superior a $512MB$ (quinhentos e doze *megabytes*). Para reverter a limitação do banco de dados Redis foi necessário criar uma lista contendo duas posições para inserir os vídeos em resolução $4k$. O banco de dados MongoDB também limita o tamanho máximo de seus arquivos em $16MB$ (dezesseis *megabytes*). Por fim, o Project Voldemort apresenta limitação quanto a inserções simultâneas, obrigando que o processo seja atômico. Desse modo, o banco de dados Project Voldemort não apresenta valores para os cenários Baixo e Alto.

A Tabela 1 expõe a média e o desvio padrão dos tempos necessário para concluir a inserção em cada cenário por resolução e banco de dados. Para o cenário solo, ausente de concorrência, o banco de dados Redis apresentou os melhores tempos em detrimento ao Project Voldemort. Ao realizar uma análise de acordo com a categoria de cada banco de dados testados, vemos um destaque para o banco de dados Redis na categoria chave-valor. Na categoria Tabular, temos o banco de dados HBase com melhores tempos em

Tabela 1: Medições - Tempo de Inserção (em milissegundos)

| Resoluções | | Cassandra | HBase | MongoDB | Redis | Voldemort |
|---|---|---|---|---|---|---|
| Cenário Solo | 240p | 838.36 ± 191 | 191.12 ± 79 | 172.8 ± 69 | 17.92 ± 4 | 1667.64 ± 99 |
| | 360p | 1390.2 ± 137 | 284.72 ± 125 | 133.2 ± 44 | 49.68 ± 15 | 2104.76 ± 118 |
| | 480p | 3611.56 ± 495 | 545.2 ± 232 | 266.08 ± 104 | 49.6 ± 9 | 4244.52 ± 150 |
| | 720p | 7735.92 ± 649 | 763.44 ± 213 | 539.12 ± 321 | 76.32 ± 12 | 6903.92 ± 238 |
| | 1080p | 11314.72 ± 663 | 1558.76 ± 292 | 920.8 ± 281 | 145.76 ± 47 | 13252.56 ± 221 |
| | 4k | 17557.36 ± 421 | 11669.44 ± 1876 | 8440.8 ± 1141 | 1553.16 ± 351 | 73701.64 ± 2378 |
| Cenário Baixo | 240p | 3857.88 ± 1965 | 1744.85 ± 530 | 605.16 ± 238 | 133.99 ± 78 | - |
| | 360p | 3376.08 ± 1291 | 2255.03 ± 534 | 1098.22 ± 494 | 403.95 ± 137 | - |
| | 480p | 9070.01 ± 1105 | 6006.67 ± 1314 | 3039.03 ± 802 | 767.92 ± 133 | - |
| | 720p | 12281.88 ± 812 | 8140.23 ± 608 | 4736.71 ± 1060 | 1820.70 ± 284 | - |
| | 1080p | 19586.10 ± 1031 | 14575.08 ± 893 | 7637.64 ± 1382 | 3067.41 ± 366 | - |
| | 4k | 70185.34 ± 1096 | 30902.97 ± 1150 | 19025.93 ± 1187 | 13847.93 ± 1476 | - |
| Cenário Alto | 240p | 5766.60 ± 1256 | 2255.28 ± 837 | 2071.93 ± 825 | 429.8 ± 128 | - |
| | 360p | 8508.08 ± 1628 | 3926.99 ± 963 | 2899.74 ± 959 | 723.67 ± 180 | - |
| | 480p | 19124.68 ± 1626 | 6704.81 ± 1564 | 5532.46 ± 1358 | 1688.99 ± 287 | - |
| | 720p | 27422.92 ± 1713 | 10808.42 ± 1549 | 9575.05 ± 1584 | 3437.84 ± 381 | - |
| | 1080p | 39947.65 ± 2289 | 20104.83 ± 2831 | 16446.07 ± 3192 | 12059.69 ± 581 | - |
| | 4k | 93491.09 ± 2817 | 42042.28 ± 2134 | 33442.01 ± 3889 | 19544.42 ± 879 | - |

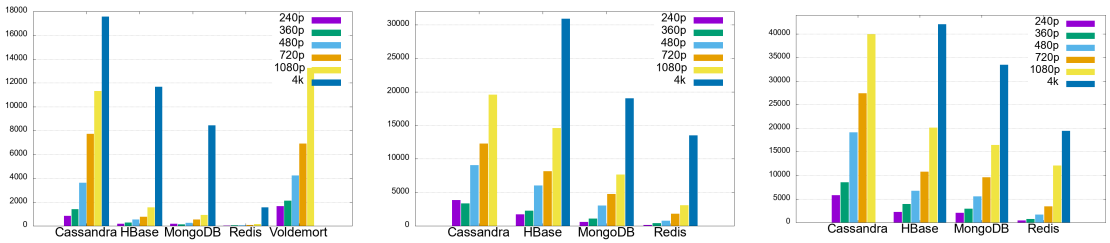
comparação ao Cassandra. E como único representante da categoria documentos, o MongoDB, com os segundos melhores tempos dentre todos os bancos de dados testados.

Para os cenários baixo e alto os experimentos apresentaram a mesma ordem de classificação dos bancos. O banco de dados Redis foi o melhor banco de dados, já o Cassandra apresentou os piores tempos. Ao realizar uma análise por categoria dos bancos de dados testados, temos o Redis como o único representante da categoria chave-valor, uma vez que, o Project Voldemort não registrou nenhum tempo devido a sua limitação. O banco de dados MongoDB foi o único representante da categoria documentos e registrou os segundos melhores tempos dentre todos os bancos de dados testados. Por fim, para a categoria tabular, o HBase apresentou tempos inferiores ao Cassandra.

O gráfico da Figura 3a apresenta os valores dos tempos das inserções no cenário solo. O valor medido com o banco de dados Project Voldemort na resolução em $4k$ não é apresentado devido ao valor elevado ter tornado o gráfico ilegível. Na figura é possível perceber o crescimento dos tempos dos bancos de dados ao alterar a resolução do vídeo. O Redis, obteve tempos próximos a zero em todas as resoluções de vídeo, exceto na resolução em $4k$. É possível identificar um crescimento linear nos bancos de dados HBase e MongoDB, com exceção da resolução de vídeo em $4k$. Por fim, o Cassandra e Project Voldemort apresentaram os maiores tempos do cenário, com uma grande elevação em comparação aos demais bancos de dados testados.

Nos gráficos das Figuras 3b e 3c são apresentados os valores dos tempos de inserção nos cenários baixo e alto, respectivamente. Os valores medidos com o banco de dados Cassandra na resolução em 4k, não são apresentados devido aos valores elevados tornarem os gráficos ilegíveis. O banco de dados Redis obteve os menores tempos medidos em ambos os cenários. Nota-se, na Figura 3b, crescimento linear nos bancos de dados Redis e MongoDB (exceto resolução em $4k$) e diminuição dos tempos médios entre as resoluções em $240p$ e $360p$ no banco de dados Cassandra.

Na Figura 3c percebe-se um crescimento linear entre as resoluções de vídeo em $240p$ à $720p$ e um novo crescimento linear entre as resoluções em $720p$ à $4k$. Os bancos de dados HBase e MongoDB apresentaram valores próximos durante todas as resoluções




(a) Solo (b) Baixo (c) Alto

Figura 3: Tempos de Inserção por Cenário (em milissegundos)

Tabela 2: Medições - Tempo de Recuperação (em milissegundos)

| Resoluções | | Cassandra | HBase | MongoDB | Redis | Voldemort |
|---|---|---|---|---|---|---|
| Cenário Solo | 240p | 1634.28 ± 262 | 797.36 ± 107 | 825.36 ± 88 | 697.52 ± 72 | 1421.16 ± 165 |
| | 360p | 1736.40 ± 175 | 909.80 ± 108 | 1139.96 ± 182 | 799.32 ± 46 | 1805.52 ± 161 |
| | 480p | 2200.44 ± 153 | 1179.96 ± 142 | 1569.00 ± 223 | 1097.80 ± 131 | 2273.08 ± 252 |
| | 720p | 2517.40 ± 335 | 1468.72 ± 115 | 1912.16 ± 162 | 1489.88 ± 198 | 2909.36 ± 158 |
| | 1080p | 10317.80 ± 830 | 2033.48 ± 201 | 2123.84 ± 222 | 2116.84 ± 448 | 14249.56 ± 1396 |
| | 4k | 18235.68 ± 508 | 7913.56 ± 331 | 9392.16 ± 474 | 6320.72 ± 381 | 22042.72 ± 392 |
| Cenário Baixo | 240p | 8125.79 ± 1351 | 3473.44 ± 882 | 4737.07 ± 596 | 2803.32 ± 1067 | 6243.22 ± 1541 |
| | 360p | 9561.01 ± 729 | 4572.15 ± 1053 | 6631.33 ± 1444 | 3424.73 ± 456 | 7739.17 ± 1046 |
| | 480p | 13328.60 ± 788 | 8358.21 ± 820 | 8785.06 ± 913 | 6999.47 ± 541 | 11552.49 ± 1083 |
| | 720p | 18321.38 ± 892 | 11319.11 ± 1118 | 11361.28 ± 972 | 8628.72 ± 701 | 16340.47 ± 950 |
| | 1080p | 23286.53 ± 1647 | 16139.46 ± 1184 | 18709.35 ± 1249 | 12348.08 ± 1374 | 31030.46 ± 978 |
| | 4k | 49518.90 ± 1031 | 30389.57 ± 1318 | 32994.72 ± 1111 | 18224.78 ± 668 | 69884.96 ± 807 |
| Cenário Alto | 240p | 11905.59 ± 732 | 6365.92 ± 703 | 8074.66 ± 704 | 4592.53 ± 444 | 9277.64 ± 669 |
| | 360p | 13517.97 ± 749 | 7414.78 ± 732 | 9119.25 ± 719 | 5060.91 ± 660 | 10319.90 ± 685 |
| | 480p | 18186.66 ± 737 | 12837.69 ± 1195 | 13132.79 ± 799 | 9233.84 ± 582 | 20292.57 ± 675 |
| | 720p | 28708.71 ± 643 | 18299.73 ± 779 | 20386.15 ± 944 | 11831.90 ± 680 | 30952.25 ± 724 |
| | 1080p | 36919.46 ± 1006 | 25700.57 ± 584 | 28393.37 ± 897 | 17862.39 ± 702 | 42676.26 ± 543 |
| | 4k | 54037.14 ± 865 | 38634.51 ± 846 | 40478.34 ± 1213 | 23826.50 ± 728 | 71606.93 ± 1590 |

de vídeo. Por fim, o banco de dados Cassandra apresentou os maiores tempos medidos no cenário, sempre com grande elevação em relação aos demais bancos de dados.

## 5.2. Recuperação

As medições para o processo de recuperação dos vídeos são realizadas no servidor de mídia. O tempo de consulta é calculado através do somatório dos tempos das solicitações das leituras do vídeo ao banco de dados, até que o vídeo seja completamente descarregado. A taxa de transferência é calculada através da razão de $20\%$ do tamanho físico do vídeo pelo tempo necessário para carregá-lo.

### 5.2.1. Tempo de Consulta

A Tabela 2 expõe a média e o desvio padrão dos tempos necessário para concluir a consulta do vídeo em cada cenário por resolução e banco de dados. Para o cenário solo o banco de dados Redis apresentou os melhores tempos de recuperação para as resoluções de vídeo em $240p$, $360p$, $480p$ e $4k$. O banco de dados HBase supera o Redis nas resoluções em $720p$ e $1080p$. Nesse mesmo cenário, o Cassandra necessitou de mais tempo para realizar a recuperação do vídeo na resolução em $240p$. Já para as demais resoluções o banco de dados Project Voldemort registrou os tempos mais elevados.

O banco de dados Redis obteve os menores tempos de recuperação para os cenários baixo e alto, independente da resolução. Em ambos os cenários, os bancos de dados Cassandra e Project Voldemort registraram os tempos mais elevados. O Cassandra superou os tempos do Project Voldemort nas resoluções em $240p$, $360p$, $480p$ e $720p$ para o cenário baixo e nas resoluções e $240p$ e $360p$ para o cenário alto.

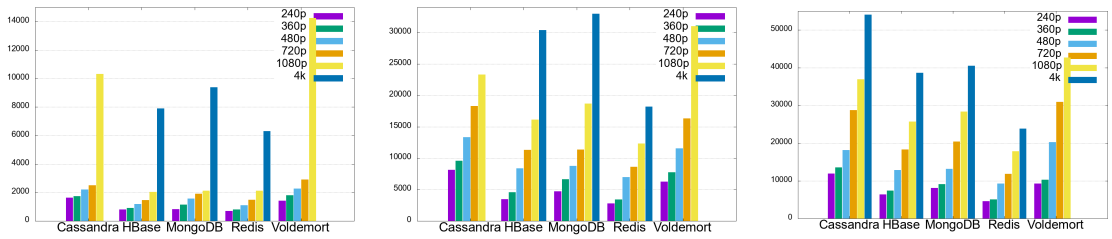
Ao realizar uma análise, de acordo com a categoria de cada banco de dados testados, os valores não sofrem alterações nos diferentes cenários. O banco de dados Redis se destaca na categoria chave-valor. Na categoria tabular, o banco de dados HBase registra os melhores tempos em comparação ao Cassandra. O MongoDB, único representante da categoria documentos, apresenta os terceiros melhores tempos dentre todos os bancos de dados testados.

A Figura 4 apresenta os gráficos dos tempos de recuperação para todos os cenários. Os valores medidos para os bancos de dados Cassandra e Project Voldemort na resolução em $4k$, não são apresentados devido aos valores elevados tornarem os gráficos ilegíveis. A Figura 4a apresenta os valores para o cenário solo. Na imagem, é possível notar o crescimento dos tempos dos bancos de dados ao alterar a resolução do vídeo. Existe uma grande proximidade nos valores medidos entre os bancos de dados Apache HBase, MongoDB e Redis, com uma distorção maior na resolução de vídeo em $4k$. Os bancos de dados Cassandra e Project Voldemort também apresentam uma proximidade nos valores medidos até a resolução de vídeo em $720p$.

Os gráficos das Figuras 4b e 4c apresentam os valores para os cenários baixo e alto, respectivamente. Nas imagens é possível perceber a elevação dos tempos de consulta ao melhorar as resoluções dos vídeos. O gráfico da Figura 4c apresenta valores bem distribuídos para cada resolução de vídeo. De maneira geral, todos os bancos de dados apresentaram crescimento linear em algum momento da medição no cenário, por exemplo, o banco de dados Project Voldemort entre as resoluções de vídeo em $360p$ e $1080p$.

### 5.2.2. Taxas de Transferência

A Tabela 3 retrata a média e o desvio padrão das taxas de transferências encontradas durante as consultas em cada cenário por resolução e banco de dados. Quanto maior a taxa de transferência, melhor o desempenho do banco de dados. O banco de dados Redis apresentou melhores taxas de transferências, para todos as resoluções, no cenário solo. O banco de dados Project Voldemort apresentou os piores valores de taxa de transferência.

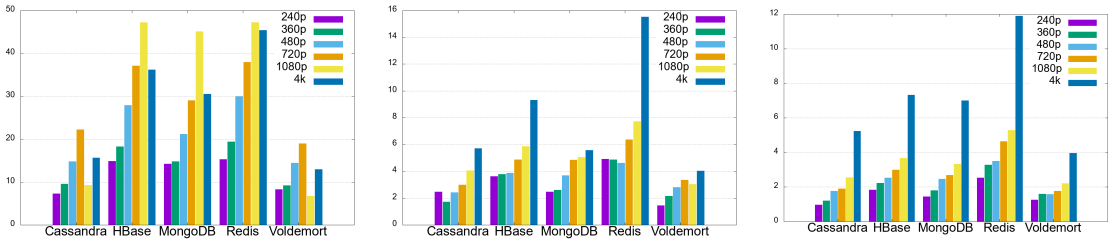



(a) Solo (b) Baixo (c) Alto

Figura 4: Tempos de Recuperação por Cenário (em milissegundos)

Tabela 3: Medições - Taxas de Transferência (em megabytes por segundo)

| Resoluções | | Cassandra | HBase | MongoDB | Redis | Voldemort |
|---|---|---|---|---|---|---|
| Cenário Solo | 240p | 7.2714 ± 1.084 | 14.8685 ± 2.347 | 14.2340 ± 1.621 | 15.3190 ± 2.454 | 8.2870 ± 1.034 |
| | 360p | 9.5344 ± 0.803 | 18.2970 ± 2.168 | 14.8107 ± 2.630 | 19.3872 ± 2.299 | 9.1762 ± 0.961 |
| | 480p | 14.7844 ± 1.115 | 27.9110 ± 4.036 | 21.1897 ± 3.952 | 29.9431 ± 3.896 | 14.4343 ± 1.772 |
| | 720p | 22.2136 ± 2.764 | 37.1111 ± 3.333 | 29.0457 ± 3.195 | 37.9597 ± 7.849 | 18.9779 ± 1.342 |
| | 1080p | 9.2580 ± 0.831 | 47.2063 ± 5.715 | 45.1458 ± 4.807 | 47.2376 ± 12.542 | 6.7314 ± 0.806 |
| | 4k | 15.6812 ± 0.877 | 36.1609 ± 2.134 | 30.4958 ± 2.054 | 45.4011 ± 3.846 | 12.9706 ± 0.736 |
| Cenário Baixo | 240p | 2.4808 ± 0.714 | 3.6174 ± 1.295 | 2.4790 ± 0.390 | 4.9141 ± 2.319 | 1.4610 ± 0.271 |
| | 360p | 1.7150 ± 0.154 | 3.7815 ± 0.998 | 2.5957 ± 0.665 | 4.8628 ± 0.828 | 2.1481 ± 0.330 |
| | 480p | 2.4212 ± 0.198 | 3.8861 ± 0.453 | 3.6967 ± 0.406 | 4.6198 ± 0.419 | 2.8068 ± 0.293 |
| | 720p | 2.9880 ± 0.195 | 4.8807 ± 0.607 | 4.8484 ± 0.531 | 6.3754 ± 0.960 | 3.3545 ± 0.257 |
| | 1080p | 4.0658 ± 0.357 | 5.8679 ± 0.527 | 5.0550 ± 0.393 | 7.7258 ± 0.960 | 3.0384 ± 0.175 |
| | 4k | 5.7088 ± 0.302 | 9.3140 ± 0.577 | 32994.72 ± 0.478 | 15.5225 ± 0.893 | 4.0439 ± 0.202 |
| Cenário Alto | 240p | 0.9735 ± 0.076 | 1.8357 ± 0.218 | 1.4423 ± 0.152 | 2.5379 ± 0.276 | 1.2509 ± 0.106 |
| | 360p | 1.2119 ± 0.091 | 2.2234 ± 0.240 | 1.8017 ± 0.164 | 3.2852 ± 0.481 | 1.5893 ± 0.130 |
| | 480p | 1.7734 ± 0.112 | 2.5295 ± 0.260 | 2.4608 ± 0.190 | 3.5009 ± 0.277 | 1.5885 ± 0.093 |
| | 720p | 1.9069 ± 0.100 | 2.9959 ± 0.198 | 2.6897 ± 0.178 | 4.6392 ± 0.338 | 1.7688 ± 0.095 |
| | 1080p | 2.5574 ± 0.141 | 3.6730 ± 0.196 | 3.3262 ± 0.192 | 5.2901 ± 0.329 | 2.2112 ± 0.111 |
| | 4k | 5.2392 ± 0.263 | 7.3295 ± 0.383 | 6.9986 ± 0.393 | 11.8923 ± 0.709 | 3.9546 ± 0.208 |




(a) Solo (b) Baixo (c) Alto

Figura 5: Taxas de Transferência por Cenário (em megabytes por segundo)

O banco de dados Redis apresenta as melhores taxas de transferência para os cenários baixo e alto, independente da resolução do vídeo. Em ambos os cenários os bancos de dados Project Voldemort e Cassandra registraram os piores valores de taxa de transferência. O Project Voldemort registrou as piores taxas, em comparação ao Cassandra, nas resoluções de vídeo em $240p$, $1080p$ e $4k$ para o cenário baixo e em $480p$, $720p$, $1080p$ e $4k$ para o cenário alto.

Ao analisar as categorias dos bancos de dados testados nos diferentes cenários verifica-se que o banco de dados Redis se destaca na categoria chave-valor. Na categoria tabular, o banco de dados HBase registra as melhores taxas de transferência em comparação ao Cassandra. O MongoDB, único representante da categoria documentos, apresenta as terceiras melhores taxas dentre todos os bancos de dados testados.

A Figura 5 apresenta os gráficos das taxas de transferências para todos cenários. Na Figura 5a é possível separar os bancos de dados Cassandra e Project Voldemort dos bancos de dados Redis, HBase e MongoDB quanto aos valores assumidos. Os bancos de dados Cassandra e Project Voldemort obtiveram as menores taxas do cenário. Os bancos de dados Redis, HBase e MongoDB obtiveram as maiores taxas do cenário, com destaque para a resolução em $1080p$.

Na Figura 5b percebe-se a constância dos valores assumidos pelo banco de dados HBase nas resoluções em $240p$, $360p$ e $480p$. Os valores das taxas de transferências

assumidos no cenário não apresentam grandes variações, com exceção da resolução em $4k$. O banco de dados Redis obteve as maiores taxas em todas as resoluções de vídeo. Por fim, na Figura 5c nota-se a semelhança entre as resoluções de vídeo até $1080p$, destoando apenas a resolução em $4k$.

### 5.3. Discussões

Durante o processo de inserção, o Redis mostrou-se superior em todos os cenários experimentados. A superioridade apresentada por esse banco de dados, ocorreu devido ao seu armazenamento de arquivos em memória RAM. O armazenamento em memória secundária ocorre apenas em momentos de baixa utilização do banco de dados. O processo é executado automaticamente em *background*. O banco de dados Project Voldemort apresentou muitas limitações, dentre as quais está a restrição para inserções simultâneas.

Na inserção os bancos de dados Project Voldemort (cenário solo) e Cassandra (cenários baixo e alto) obtiveram os piores tempos. Os valores medidos para a resolução em $4k$ tornam seu uso contraindicado, principalmente para cenários com acesso simultâneo. Em contrapartida, os bancos de dados HBase e MongoDB apresentaram tempos satisfatórios em todos os cenários testados.

Nós obtivemos tempos de consulta semelhantes entre os bancos de dados, para as diferentes resoluções de vídeo. A semelhança nos tempos de recuperação ocorre devido a utilização de memória principal para armazenamento, característica presente em vários bancos de dados NoSQL. O banco de dados Redis foi protagonista em todos os cenários analisados mas para determinados cenários e resolução o banco de dados HBase apresentou melhores tempos.

O banco de dados Project Voldemort e Cassandra apresentaram tempos elevados, sempre com alternância de ambos na última posição das análises. O Project Voldemort obteve os piores resultados para os vídeos em $4k$, demonstrando que o banco de dados não possui um bom comportamento com grandes valores associados a uma chave.

A análise das taxas de transferências demonstrou não ter relação direta com os tempos totais de recuperação. Os menores tempos das consultas não significam, obrigatoriamente, que o banco de dados irá obter a maior taxa de transferência. Esse fenômeno é explicado, pois os experimentos levaram em consideração o tempo de consulta dos primeiros $20\%$ do tamanho físico do arquivo de vídeo. Para essa análise, o banco de dados Redis obteve as maiores taxas em todos os cenários e resoluções. Os menores valores de taxas de transferências ficaram por conta do Project Voldemort e do Cassandra.

O banco de dados Redis obteve o melhor desempenho ao manipular dados multimídias de áudio e vídeo. Quanto a categoria de banco de dados NoSQL, não é possível eleger a melhor para armazenar vídeos. Por exemplo, na categoria chave-valor o banco de dados Project Voldemort obteve tempos elevados de inserção e recuperação, em contradição com o Redis. A mesma discrepância é verificada na categoria tabular com o HBase obtendo valores satisfatórios para as inserções e recuperação e o Cassandra obtendo valores elevados para as mesmas operações.

## 6. Conclusões e Trabalhos Futuros

Este artigo teve como objetivo propor a arquitetura Video7 para armazenamento e recuperação de dados de fluxo contínuo de áudio e vídeo em bases de dados NoSQL.

Com o intuito de validar essa arquitetura, foram realizadas análises nas operações de inserção e recuperação. Para tal, foi aferido o tempo de conclusão das operações e a taxa de transferência na recuperação dos dados. Essa análise foi realizada para diferentes qualidades de resolução de vídeos em diversas bases de dados NoSQL. Ao fim o banco de dados Redis apresentou os melhores valores, em comparação aos demais bancos de dados, tanto na operação de inserção quanto na recuperação do dado. O banco de dados Project Voldemort, de modo contrário, apresentou os piores valores medidos, além de limitações para aplicação de concorrência na inserção.

A arquitetura contribui de maneira a analisar tipos diferentes de entrada, com a inclusão de dados não estruturados, mas não limitados a tal. Outra vantagem está na escalabilidade das bases de dados em análise, o que permite realizar comparações entre diferentes modelos de bancos de dados (e.g., relacional, não relacional, newSQL, in memory) ou diferentes categorias dentro de um mesmo modelo, como foi apresentado no artigo. Como trabalho futuro será realizado uma análise comparativa entre os diferentes modelos de bancos de dados para diferentes tipos de entradas não estruturadas, não se limitando aos vídeos. A inclusão de métricas mais sofisticadas para os tipos de entradas possíveis também será abordada em trabalhos futuros.

## Referências

Assis, J. O., Souza, V. C. O., Paula, M. M. V., and Cunha, J. B. S. (2017). Performance evaluation of nosql data store for digital media. *12th Iberian Conference on Information Systems and Technologies (CISTI)*.

Cheng, X., Dale, C., and Liu, J. (2008). Statistics and social network of youtube videos. *16th International Workshop on Quality of Service*.

da Costa, J. P. (2010). Youtube vs vimeo: uma análise comparativa de acessibilidade, usabilidade e desejabilidade para os utilizadores de fluxos videomusicais. *Revista de Ciências e Tecnologias de Informação e Comunicação*.

de Macedo, D. D., Capretz, M. A., Prado, T. C., von Wangenheim, A., and Dantas, M. (2011). An improvement of a different approach for medical image storage. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2011 20th IEEE International Workshops on*, pages 140–142. IEEE.

de Macedo, D. D., Von Wangenheim, A., and Dantas, M. A. (2015). A data storage approach for large-scale distributed medical systems. In *Complex, Intelligent, and Software Intensive Systems (CISIS), 2015 Ninth Intl. Conf. on*, pages 486–490. IEEE.

Li, H., Zhang, X., Wang, S., and Du, X. (2010). Towards video management over relational database. *Web Conference (APWEB), 2010 12th International Asia-Pacific*.

Li, T., Liu, Y., Tian, Y., Shen, S., and Mao, W. (2012). A storage solution for massive iot data based on nosql. *2012 IEEE Intl. Conf. on Green Computing and Communications*.

Rebecca, D. and Shanthi, I. (2016). A nosql solution to efficient storage and retrievel of medical images. *International Journal of Scientific & Engineering Research*, 7.

Suchomski, M., Militzer, M., and Meyer-Wegener, K. (2005). Retavic: Using meta-data for real-time video encoding in multimedia servers. *NOSSDAV '05 Proceedings of the intl. workshop on Network and operating systems support for digital audio and video*.

# Impacto de memórias aproximadas na eficiência energética

**Isaías B. Felzmann**[1], **João Fabrício Filho**[1,2], **Rodolfo Azevedo**[1], **Lucas Wanner**[1]

[1]Instituto de Computação - Universidade Estadual de Campinas
Campinas, SP, Brasil

[2]Universidade Tecnológica Federal do Paraná – Câmpus Campo Mourão
Campo Mourão, PR, Brasil

isaias.felzmann@students.ic.unicamp.br

joaof@utfpr.edu.br

{rodolfo,lucas}@ic.unicamp.br

***Abstract.*** *Approximate memories can lower energy consumption at expense of incurring errors in some of the read/write operations. While these errors may be tolerated in some cases, in general, parts of the application must be re-executed to achieve usable results when a large number of errors occur. Frequent re-executions may, in turn, attenuate or negate energy benefits obtained from using approximate memories. In this work, we show the energy impact of memory approximations in applications considering different quality requirements. Five out of nine selected applications showed a positive energy-quality tradeoff. For these applications, our results show up to 30% energy savings at a* $10^{-8}$ *error rate, when a 20% degradation in quality is allowed.*

***Resumo.*** *Memórias aproximadas podem diminuir o consumo de energia ao custo de erros em operações de leitura e escrita. Embora esses erros possam ser tolerados, em geral partes da aplicação devem ser reexecutadas para obter resultados utilizáveis quando muitos erros ocorrem. Entretanto, reexecuções frequentes podem atenuar ou eliminar os benefícios em energia. Neste trabalho, apresentamos o impacto energético de aproximações em memória considerando diferentes aplicações e requisitos de qualidade. Cinco aplicações dentre nove selecionadas apresentaram um equilíbrio positivo entre energia e qualidade. Para essas, demonstramos até 30% de economia de energia para taxas de erro na ordem de* $10^{-8}$*, quando se permite 20% de degradação na qualidade.*

## 1. Introdução

Elementos de memória podem representar até 40% do consumo de energia em sistemas computacionais [Paul et al. 2015,Chang et al. 2017]. O uso de técnicas de ajuste dinâmico de tensão (*Dynamic Voltage-Frequency Scaling - DVFS*) permite que memórias operem em regiões de maior eficiência energética [Calhoun e Chandrakasan 2005]. Porém, a

Este trabalho tem apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES – PROCAD 2966/2014) e da Fundação de Apoio à Pesquisa do Estado de São Paulo (Projeto FAPESP 2017/08015-8)

diminuição da tensão de alimentação resulta em menores margens de ruído estático e dinâmico e, consequentemente, em maior probabilidade de erros de leitura e escrita [Wang e Calhoun 2011].

Tendo em vista que diversas aplicações apresentam alguma resiliência a erros, o paradigma de Computação Aproximada propõe a exploração mais agressiva das margens de tolerância em diversos níveis de projeto, incluindo o ajuste de tensão além da margem aceitável de ruído [Chippa et al. 2014, Gottscho et al. 2014]. Isso permite uma significativa redução na dissipação de potência de um circuito integrado, que é um fator limitante para o desenvolvimento de *hardware* [Esmaeilzadeh et al. 2011, Kugler 2015]. Por outro lado, os erros podem ter impacto maior do que o previsto na qualidade do resultado de qualquer aplicação executada, o que requer algum mecanismo de recuperação e amortiza os ganhos em eficiência energética [Ganapathy et al. 2015, Gupta et al. 2013].

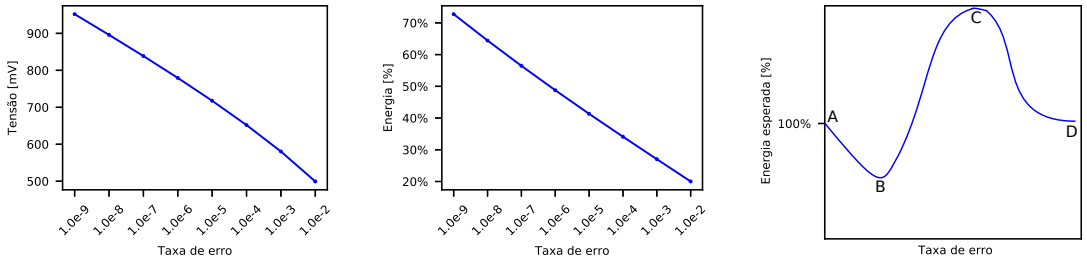
Neste trabalho, utilizamos simulação de arquiteturas computacionais baseada em ArchC [Rigo et al. 2004, Felzmann et al. 2018] para expor diferentes tipos de aplicações a memórias aproximadas. Assim buscamos o ponto de equilíbrio entre o ganho com economia de energia e a perda para recuperação de resultados. Em nosso sistema conceitual, a aplicação possui controle direto sobre parâmetros que definem a probabilidade de erros na leitura e escrita de dados. Além disso, o sistema implementa um mecanismo de controle de qualidade baseado em reexecução de aplicações cujas métricas de qualidade não atingem níveis aceitáveis.

Nossos experimentos demonstram a execução de 9 aplicações que representam atividades comuns em sistemas computacionais. Dessas aplicações, mostramos 5 pontos de inflexão no consumo de energia, considerando as reexecuções para readequação de qualidade. Nossos resultados mostram o perfil energético para cada aplicação e demonstram o equilíbrio entre qualidade e energia, obtendo até 30% de economia de energia para aplicações que exijam um mínimo de 80% de precisão nos resultados.

## 2. Fundamentos e trabalhos relacionados

A Computação Aproximada consiste em um padrão de computação com menor custo energético, comprometendo a precisão dos resultados obtidos [Marwaha e Sharma 2018]. Uma técnica conhecida para exploração de Computação Aproximada é o ajuste de tensão de alimentação além dos limites considerados seguros para manutenção de um resultado preciso, geralmente causando falhas temporais ou de chaveamento dos circuitos [Chippa et al. 2014]. Em elementos de memória, a redução da tensão de alimentação provoca a redução das margens de ruído estático e dinâmico nas células de memória, resultando em uma maior probabilidade de erros de leitura e escrita ou na perda do dado armazenado [Slayman 2011, Gottscho et al. 2014].

O modelo estatístico proposto por Wang e Calhoun [2011] associa, com base em valores iniciais experimentais, a tensão de alimentação com a probabilidade de ocorrência de erros, conforme a Figura 1(a). Considerando que energia possui relação quadrática com a tensão de alimentação, é possível prever o consumo energético após ajuste de tensão, relativo ao consumo de uma memória que garanta uma taxa de erro menor do que $10^{-12}$ (Fig. 1(b)). A análise em nível de circuito, porém, dificulta o estudo do impacto desses erros em memória no nível de aplicação. A alternativa é a modelagem dos erros em níveis de abstração mais altos, na forma de modificações nas palavras de da-




(a) (b) (c)

**Figura 1. Relação entre energia e erros em memórias: (a) Tensão de operação; (b) Energia; (c) Impacto esperado em uma aplicação.**

dos provenientes dos elementos de memória, e a utilização desses modelos em ambientes simulados.

EnerJ [Sampson et al. 2011] propõe um controle explícito para aproximar tipos de dados em linguagem de programação. Esse trabalho mostrou que pequenas aproximações de dados podem inferir em um significativo potencial de economia de energia. Sua sucessão, ACCEPT [Sampson et al. 2015], é um *framework* para programação aproximada guiado por anotações. As relaxações por aproximação desse trabalho demonstraram benefícios para os programas no desempenho e na utilização energética.

No trabalho em [Dorn et al. 2017], os autores exploraram modificações em códigos de programas, visando otimizações para aplicações de *data centers*. Os autores relacionaram a qualidade das saídas de cada código gerado com a economia de energia resultante, mostrando que há menor consumo energético em aplicações com maior aceitabilidade de erro. DrSEUs [Carlisle e George 2018] se volta para memórias, explorando possíveis falhas na cache de processadores e a classificação dos possíveis erros decorrentes dessas falhas, levando em consideração a resiliência da aplicação e a implicação em reexecuções, porém sem considerar as implicações em termos de energia. Rumba [Khudia et al. 2015] revisita a ideia de reexecução com um mecanismo de verificação de partes da aplicação para erros mais altos, ao custo de um monitoramento em tempo de execução.

Neste trabalho, expandimos a cobertura de erros em memória dentro de diferentes tipos de aplicação, sem isolamento de estruturas. Para cada aplicação, apresentamos os limiares de ganho energético com base na qualidade das saídas e probabilidade da necessidade de uma reexecução. Com base nos dados de Wang e Calhoun [2011] e considerando a possibilidade de reexecução, a Figura 1(c) demonstra a tendência esperada no consumo médio de energia de acordo com a taxa de erros de memória. No ponto inicial $A$, a taxa de erros é tão baixa que se compara a uma execução sem erros, com custo energético equivalente. A medida que a taxa aumenta, o consumo energético encaminha-se para o ponto de equilíbrio $B$, de maior economia, caracterizado pela necessidade de poucas reexecuções. A medida que mais reexecuções são necessárias, a energia consumida por execuções precisas adicionais passa a ser dominante, até o ponto máximo $C$, em que muitas execuções aproximadas completam, mas resultam em qualidade abaixo do requerido. Por fim, a ocorrência de falhas de execução começa a causar a conclusão prematura de algumas execuções, reduzindo o trabalho em execuções aproximadas e fazendo com que novamente o custo energético se aproxime ao custo de execuções precisas, no ponto $D$.

## 3. Condução dos experimentos

O modelo de arquitetura que consideramos neste trabalho é baseado em um processador de baixo consumo voltado para dispositivos embarcados. Nesse modelo conceitual, existe a implementação de um conjunto de "estados aproximados" capazes de influenciar diretamente a tensão de alimentação do banco de registradores e da memória de dados, deixando o sistema suscetível a erros de leitura e escrita [Wang e Calhoun 2011]. O estado aproximado atual é definido por meio de um registrador mapeado em memória disponível para a aplicação.

Essa arquitetura foi representada em linguagem ArchC com base em um modelo do processador MIPS [Rigo et al. 2004]. O modelo original foi estendido para a inclusão de aproximações utilizando a linguagem de descrição ADeLe, voltada à modelagem de Computação Aproximada [Felzmann et al. 2018]. Todas as operações de leitura e escrita no banco de registradores e na memória de dados foram substituídas por modelos em software de três tipos de erros:

- *BitFlip*: um dos bits do vetor de dados é sorteado e seu valor é invertido;
- *StuckAt(0)*: um dos bits do vetor de dados é sorteado e seu valor definido como 0;
- *StuckAt(1)*: análogo a *StuckAt(0)*, com o valor do bit definido como 1.

Devido a tais erros serem probabilísticos, qualquer operação de leitura ou escrita tem a mesma probabilidade de ser afetada. Foram selecionadas 9 diferentes aplicações para representar um conjunto de elementos comuns em dispositivos embarcados. Em cada aplicação, o código do *kernel* de execução foi isolado e apenas nele os erros são aplicados. Desse modo, as operações de entrada e saída, características de simulação, são executadas de forma precisa. As aplicações selecionadas, suas respectivas classificações e métricas de qualidade são:

- **Aplicações típicas:** Tipicamente, trabalhos relacionados a Computação Aproximada utilizam algoritmos de processamento multimídia para demonstração de resultados [Mittal 2016]. Assim, selecionamos o algoritmo de compressão de imagens JPEG disponível na suíte AxBench [Yazdanbakhsh et al. 2017] e a computação de Transformada Rápida de Fourier (FFT) do MiBench [Guthaus et al. 2001]. As imagens JPEG computadas pela versão sujeita a erros foram comparadas com resultados precisos utilizando a métrica de Similaridade Estrutural [Wang et al. 2004, Avanaki 2009]. Para FFT foi computado o número de amostras fora de uma margem tolerância da ordem $10^{-9}$ no sinal após reconstrução com a Transformada Inversa.
- **Aplicações *CPU-bound*:** As aplicações Mandelbrot, N-Body e SpectralNorm foram selecionadas dentre as aplicações em [Gouy 2004]. Essas aplicações têm em comum um maior uso do tempo de CPU e um menor uso de memória, portanto potencialmente demonstrando maior resiliência para aproximações em memória. Os *bitmaps* gerados pelas versões sujeitas a erros na aplicação Mandelbrot foram comparados com as saídas geradas pela versão precisa utilizando a métrica de Similaridade Estrutural [Wang et al. 2004, Avanaki 2009]. As qualidades das saídas das execuções sujeitas a erros de N-Body e SpectralNorm foram calculadas pelo complemento do erro relativo médio.
- **Aplicações *Memory-Bound*:** Foram selecionados os algoritmos de Dijkstra, ordenação QSort, compressão de dados bzip e descompressão de dados bunzip

a partir das suítes MiBench [Guthaus et al. 2001] e cBench [Fursin 2008]. Essas aplicações apresentam um maior uso da memória, e, portanto, são mais suscetíveis ao tipo de aproximação de dados explorada neste trabalho. A qualidade das saídas sujeitas a erros da aplicação QSort é medida conforme a fração de elementos iguais aos da ordenação correta. Para implementar a métrica de qualidade da aplicação Dijkstra, a saída foi modelada com base na construção de tabelas de roteamento, na qual cada dado na linha $i$ e coluna $j$ indica o próximo *hop* para chegar ao destino $j$ estando na origem $i$. Dessa forma, a qualidade é a fração de próximos $hops$ corretos na tabela de saída. As aplicações bzip e bunzip foram utilizadas para comprimir e descomprimir arquivos-texto, analisando qualidade pela similaridade entre as cadeias de caracteres do conteúdo dos arquivos.

As execuções de cada aplicação foram repetidas 100 vezes, submetidas a variadas probabilidades de erros de leitura e escrita. As probabilidades foram definidas desde o ponto de ocorrência de 1 erro dentre todas as instruções no *kernel* na aplicação (conforme o número de instruções executado) até $1\%$. Para cada execução, foram analisadas a resiliência da aplicação – que é a probabilidade de ocorrência de uma falha que impeça o término da execução – e a qualidade final do resultado gerado.

Na arquitetura proposta, quando uma determinada execução não atinge um limiar mínimo de qualidade, o resultado deve ser reexecutado em modo preciso, com uma penalidade em energia. Assim, as métricas de qualidade e resiliência foram agregadas para obter a probabilidade de reexecução de uma aplicação em modo preciso e o consumo de energia total, considerando-se a execução aproximada e a precisa subsequente, quando houver. A energia necessária para execução foi derivada pelo método em [Wang e Calhoun 2011].

## 4. Resultados

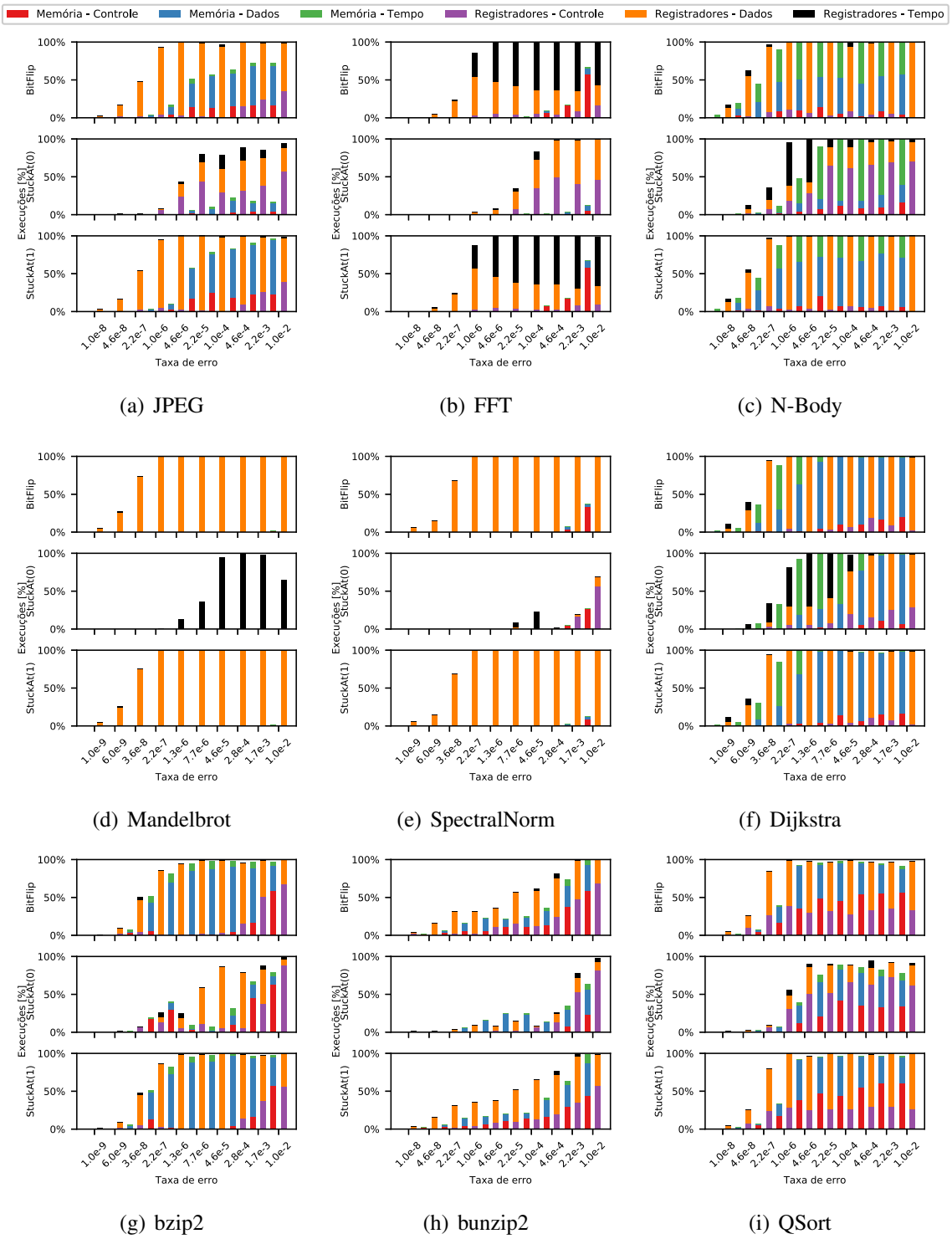
Nossos resultados mostram a análise de resiliência das aplicações para 100 repetições de cada execução. Além disso, foram analisadas as métricas de qualidade apresentadas na Seção 3 e, a partir delas, computada a energia média de uma execução qualquer.

### 4.1. Análise de resiliência

Para análise de resiliência, a ocorrência de uma falha significa que a execução foi interrompida antes da produção de um resultado válido. Tais falhas foram classificadas em três categorias:

- Falhas no fluxo de **controle**: ocorrem quando o endereço de destino de um salto é lido incorretamente, provocando um salto para um endereço inválido.
- Falhas no fluxo de **dados**: ocorrem quando um dado é buscado num endereço de memória inválido (*Segmentation Fault*).
- Falhas de **tempo**: ocorrem quando um resultado válido não é computado em tempo hábil. O tempo máximo que uma execução aproximada pode utilizar foi fixado em 5 vezes o tempo de uma computação precisa.

Os gráficos da Figura 2 demonstram a análise de resiliência das aplicações. De maneira geral, a suscetibilidade a erros no banco de registradores da arquitetura proposta apresenta drásticos efeitos no fluxo da aplicação. De fato, o banco de registradores armazena, além de valores locais, endereços de memória, variáveis de controle de laços



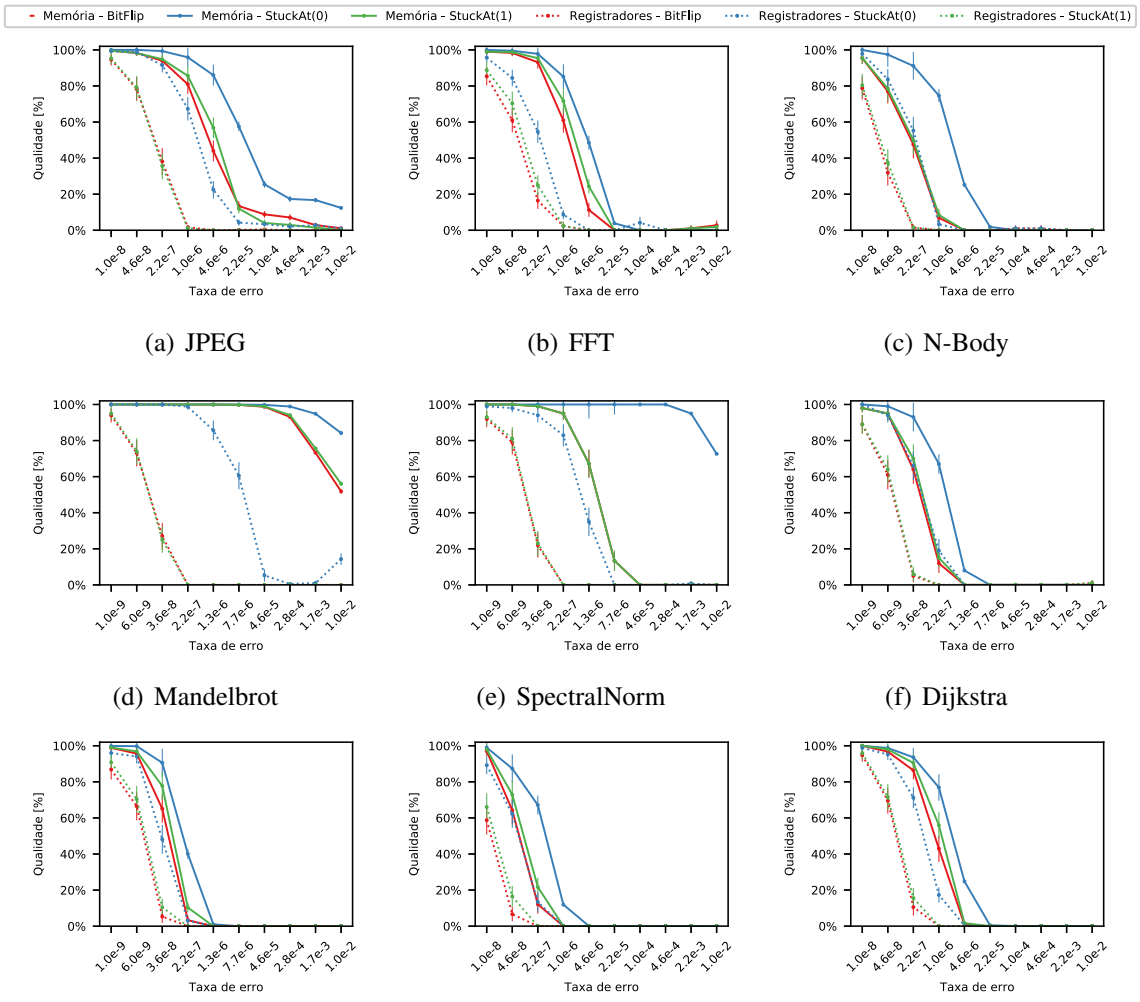

(a) JPEG (b) FFT (c) N-Body

(d) Mandelbrot (e) SpectralNorm (f) Dijkstra

(g) bzip2 (h) bunzip2 (i) QSort

**Figura 2. Análise de resiliência.**

e endereços de retorno de funções ou destino de saltos. Além disso, as falhas do tipo *Tempo*, em que um resultado não foi computado em tempo hábil, concentram-se principalmente em situações em que erros são aplicados aos registradores, demonstrando a baixa resiliência de estruturas de controle a aproximações.




(a) JPEG (b) FFT (c) N-Body

(d) Mandelbrot (e) SpectralNorm (f) Dijkstra

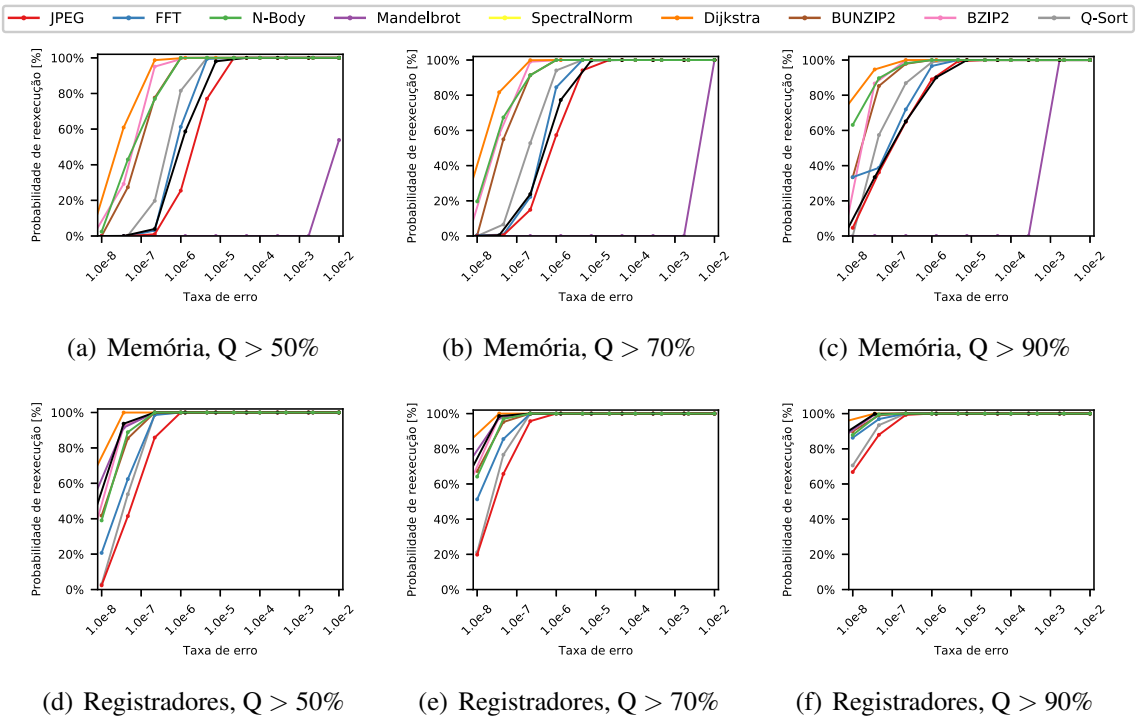
(g) bzip2 (h) bunzip2 (i) QSort

**Figura 3. Qualidade dos resultados.**

Os erros do tipo *StuckAt(0)* mostram ser mais facilmente mascarados pelas aplicações em termos de falhas de execução. Esse tipo de erro, quando afeta endereços de memória, tem a tendência de alterar o dado para um endereço que ainda pertence ao programa executado, possivelmente na mesma página de memória, mitigando a ocorrência de falhas de dados e controle. Por outro lado, esse mesmo comportamento, quando aplicado a endereços de destino de saltos ou variáveis de controle de laços, aumenta o tempo de execução da aplicação, possivelmente criando laços infinitos, aumentando a ocorrência de falhas de tempo, especialmente naquelas baseadas em convergência, como N-Body (Fig. 2(c)) e Dijkstra (Fig. 2(f)).

A resiliência das aplicações é fator limitante do equilíbrio entre qualidade e energia, uma vez que uma falha de execução resulta em computação inútil e, consequentemente, desperdício de energia. Além disso, mesmo quando uma falha não ocorre, problemas de controle podem levar a um maior tempo de execução, com efeito negativo sobre a energia.

A maior parte das falhas decorrentes dos erros inseridos em memória são decor-




(a) Memória, Q > 50% (b) Memória, Q > 70% (c) Memória, Q > 90%

(d) Registradores, Q > 50% (e) Registradores, Q > 70% (f) Registradores, Q > 90%

**Figura 4. Probabilidade de reexecução.**

rentes do acesso a endereços inválidos. Esse tipo de falha pode ser mascarado com base no isolamento de áreas de memória para armazenamento de ponteiros e endereços alvo de saltos. Desta forma, as aplicações demonstrariam uma maior resiliência, já que os erros afetariam apenas os dados propriamente ditos, mas haveria a necessidade de se manter regiões de memória sempre em um modo não sujeito a falhas, o que impacta o custo energético das operações.

## 4.2. Qualidade dos resultados

Os gráficos da Figura 3 mostram a qualidade final média do resultado relativa a uma execução precisa da mesma aplicação. Para computação da qualidade, foi utilizada a média das métricas descritas na Seção 3 para 100 execuções, com um intervalo de confiança de 95%. As execuções que resultam em falha de execução foram computadas com qualidade zero, pois um resultado utilizável não foi obtido.

As falhas de execução são fatores dominantes na medida final de qualidade. O isolamento de estruturas de controle pode evitar falhas de execução, direcionando o impacto dos erros ao resultado final. A análise dos pontos iniciais dos gráficos da Figura 3 evidencia que o detrimento de qualidade é menos abrupto em execuções que completam com sucesso, o que indica uma possibilidade de maior eficiência energética.

Outro efeito da baixa resiliência das aplicações é o maior impacto na qualidade causado pelas aproximações em registradores. Ainda assim, desconsiderando-se as falhas, a aproximação de registradores provoca maior degradação de qualidade, pois como as operações ocorrem em maior número, são mais afetadas por erros. Isso indica uma desvantagem da técnica de aproximação quando aplicada a registradores.

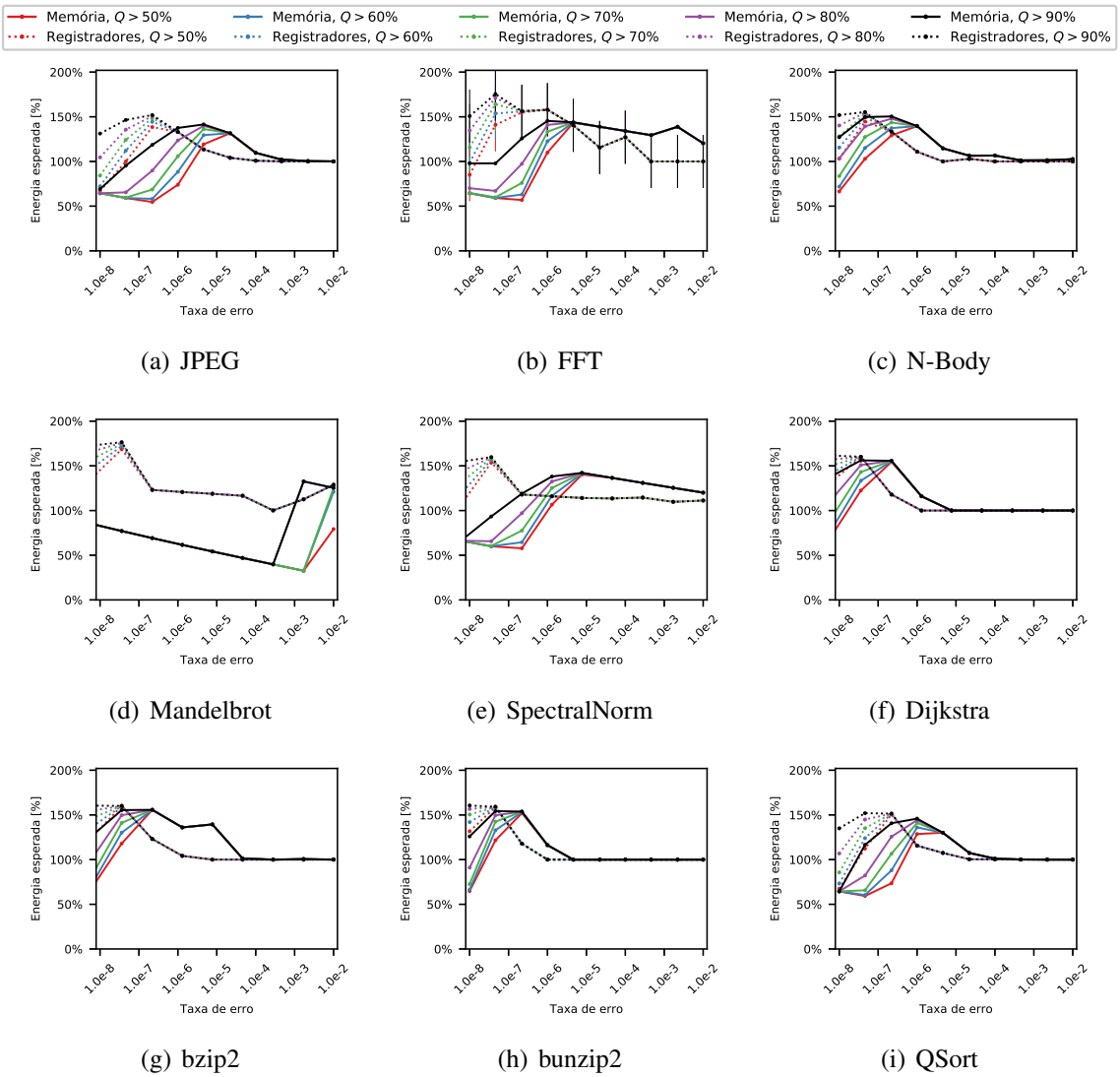



**Figura 5. Equilíbrio entre qualidade e energia.**

## 4.3. Energia

Embora o uso de estruturas de memória aproximadas ofereça um ganho em economia de potência no sistema, a ocorrência de falhas de execução e a degradação da qualidade do resultado podem requerer a reexecução de determinadas instâncias de uma aplicação. Quando uma reexecução em modo preciso é necessária, existe um impacto no consumo de energia. Com base na qualidade do resultado esperada para cada aplicação (Seção 4.2), calculamos a probabilidade de uma determinada execução resultar em qualidade abaixo do requerido e ter de ser reexecutada em modo preciso. A Figura 4 mostra essa probabilidade quando deseja-se uma métrica de qualidade maior que 50%, 70% e 90%, com erros aplicados a memória e registradores.

Uma reexecução em modo preciso resulta em uma penalidade em energia fixa para cada aplicação. Por outro lado, a energia de uma execução em modo aproximado depende do tempo de execução até a obtenção de um resultado ou a ocorrência de uma falha. Assim, a energia relativa esperada foi computada de acordo com a média do número

de instruções executadas pela versão aproximada da aplicação e do número de instruções do modo preciso.

A Figura 5 relaciona a taxa de erros na execução de cada aplicação com o consumo médio esperado de energia, com um intervalo de confiança de 95%. De modo geral, todas as aplicações demonstram alguma economia de energia com a flexibilização do requisito de qualidade, quando considerados erros aplicados em memória. Para erros em registradores, por outro lado, a maior parte dos resultados sequer demonstra alguma economia de energia, prejudicada principalmente pela baixa resiliência das aplicações.

De modo particular, a inclusão de falhas em registradores para a aplicação FFT ocasiona uma grande variação no número de instruções executadas em modo aproximado entre diferentes repetições. Um comportamento semelhante também pode ser observado analisando-se o grande número de falhas de tempo para essa aplicação (Figura 2(b)). A estrutura de laços aninhados do algoritmo da Transformada de Fourier é bastante desfavorável a erros que afetem as estruturas de controle, como os aplicados a registradores. Para as demais aplicações, por outro lado, a variação no número de instruções em modo aproximado, embora cause variações na energia de cada execução, é facilmente mitigada perante várias repetições.

Os resultados para o consumo de energia demonstram um comportamento semelhante ao esperado (Figura 1(c)), em que o consumo diminui com o aumento da taxa de erros até o ponto em que há um maior detrimento de qualidade, tornando a reexecução um fator dominante. Esse ponto de inflexão no consumo de energia – ponto *B* na Figura 1(c) – é visível para 5 das 9 aplicações testadas. As demais aplicações apresentam uma inclinação ascendente para os pontos iniciais, indicando que o ponto de equilíbrio existe, mesmo que menos acentuado, em uma taxa de erros menor. Taxas menores, por outro lado, caracterizam uma menor economia de energia, atenuando os efeitos da exploração de aproximações.

## 5. Conclusão

Apresentamos um estudo do efeito de erros em memória, caracteristicamente decorrentes da exploração de técnicas de Computação Aproximada, na execução de aplicações. Nosso estudo demonstra um ponto de equilíbrio entre a qualidade final do resultado e o custo energético da utilização de memórias em um sistema computacional. A economia de energia, embora mais acentuada em aplicações inerentemente mais resilientes a falhas, está também presente em aplicações não tradicionalmente associadas com Computação Aproximada. Assim, uma maior permissividade de erros em memória demonstrou-se vantajosa, evidenciando uma boa aplicabilidade das técnicas para o aumento da eficiência energética.

Apesar das vantagens, algumas aplicações só começam a demonstrar economia de energia para requisitos de qualidade mais flexíveis. A flexibilização da qualidade, dependendo da aplicação, pode ser indesejável, de modo que é necessário o desenvolvimento de técnicas que melhorem a resiliência das execuções. Nossos experimentos mostram que as falhas de execução decorrentes de erros em memória, como saltos para endereços inválidos, são um fator determinante da qualidade final do resultado. Para melhorar a resiliência, seria possível isolar alguns dados, como variáveis intimamente ligadas com controle e endereços de memória, em regiões não sujeitas a falhas.

O isolamento de áreas de memória para armazenamento de dados que não admitem erros, aliado à exploração de aproximações em um conjunto de dados extenso, tem o potencial de deslocar o ponto de equilíbrio entre qualidade e energia para regiões energeticamente mais eficientes. Assim, é importante o desenvolvimento de ferramentas voltadas à identificação de regiões de dados verdadeiramente essenciais, para minimizar a ocorrência de falhas nas execuções. Como trabalhos futuros, planejamos o estudo da resiliência de diversas aplicações, isolando regiões de dados essenciais, por meio de modelos de programação ou arquitetura, com o objetivo de avaliar o equilíbrio entre qualidade e energia, como base para o desenvolvimento de uma ferramenta automatizada.

## Referências

Avanaki, A. N. (2009). Exact histogram specification optimized for structural similarity. *Optical Review*. DOI: 10.1007/s10043-009-0119-z.

Calhoun, B. H. e Chandrakasan, A. (2005). Analyzing static noise margin for sub-threshold SRAM in 65nm CMOS. In *ESSCIRC*. DOI: 10.1109/ESSCIR.2005.1541635.

Carlisle, E. e George, A. D. (2018). Cache fault injection with DrSEUs. In *IEEE Aerospace Conference*. DOI: 10.1109/AERO.2018.8396690.

Chang, K. K., Yauglikcci, A. G., Ghose, S., Agrawal, A., Chatterjee, N., Kashyap, A., Lee, D., O'Connor, M., Hassan, H., e Mutlu, O. (2017). Understanding reduced-voltage operation in modern dram devices: Experimental characterization, analysis, and mechanisms. In *POMACS*. DOI: 10.1145/3084447.

Chippa, V. K., Mohapatra, D., Roy, K., Chakradhar, S. T., e Raghunathan, A. (2014). Scalable Effort Hardware Design. *TVLSI*. DOI: 10.1109/TVLSI.2013.2276759.

Dorn, J., Lacomis, J., Weimer, W., e Forrest, S. (2017). Automatically Exploring Tradeoffs Between Software Output Fidelity and Energy Costs. *IEEE TSE*. DOI: 10.1109/TSE.2017.2775634.

Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., e Burger, D. (2011). Dark silicon and the end of multicore scaling. In *ISCA*. DOI: 10.1145/2000064.2000108.

Felzmann, I. B., Susin, M. M., Duenha, L., Azevedo, R. J., e Wanner, L. F. (2018). ADeLe: Rapid Architectural Simulation for Approximate Hardware. In *SBAC-PAD*.

Fursin, G. (2008). Collective Benchmark. `ctuning.org/cbench/`.

Ganapathy, S., Karakonstantis, G., Teman, A., e Burg, A. (2015). Mitigating the Impact of Faults in Unreliable Memories for Error-resilient Applications. In *DAC*. DOI: 10.1145/2744769.2744871.

Gottscho, M., BanaiyanMofrad, A., Dutt, N., Nicolau, A., e Gupta, P. (2014). Power / capacity scaling: Energy savings with simple fault-tolerant caches. In *DAC*. DOI: 10.1145/2593069.2593184.

Gouy, I. (2004?). The Computer Language Benchmarks Game. `benchmarksgame-team.pages.debian.net/benchmarksgame/`.

Gupta, P., Agarwal, Y., Dolecek, L., Dutt, N., Gupta, R. K., Kumar, R., Mitra, S., Nicolau, A., Rosing, T. S., Srivastava, M. B., Swanson, S., e Sylvester, D. (2013). Underdesig-

ned and opportunistic computing in presence of hardware variability. *IEEE TCAD*. DOI: 10.1109/TCAD.2012.2223467.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., e Brown, R. B. (2001). MiBench: A free, commercially representative embedded benchmark suite. In *IEEE WWC*. DOI 10.1109/WWC.2001.990739.

Khudia, D. S., Zamirai, B., Samadi, M., e Mahlke, S. (2015). Rumba: An online quality management system for approximate computing. In *ISCA*. DOI: 10.1145/2749469.2750371.

Kugler, L. (2015). Is "Good Enough" Computing Good Enough? *CACM*. DOI: 10.1145/2742482.

Marwaha, D. e Sharma, A. (2018). A review on approximate computing and some of the associated techniques for energy reduction in IOT. In *ICISC*. DOI: 10.1109/ICISC.2018.8399087.

Mittal, S. (2016). A survey of techniques for approximate computing. *CSUR*. DOI: 10.1145/2893356.

Paul, I., Huang, W., Arora, M., e Yalamanchili, S. (2015). Harmonia: Balancing Compute and Memory Power in High-performance GPUs. In *ISCA*. DOI: 10.1145/2749469.2750404.

Rigo, S., Araujo, G., Bartholomeu, M., e Azevedo, R. (2004). ArchC: a systemC-based architecture description language. In *SBAC-PAD*. DOI: 10.1109/SBAC-PAD.2004.8.

Sampson, A., Baixo, A., Ransford, B., Moreau, T., Yip, J., Ceze, L., e Oskin, M. (2015). ACCEPT: A Programmer-Guided Compiler Frameworkfor Practical Approximate Computing. Technical report, University of Washington, UW-CSE.

Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., e Grossman, D. (2011). EnerJ: Approximate Data Types for Safe and General Low-power Computation. In *PLDI*. DOI: 10.1145/1993498.1993518.

Slayman, C. (2011). Soft error trends and mitigation techniques in memory devices. In *RAMS*. DOI: 10.1109/RAMS.2011.5754515.

Wang, J. e Calhoun, B. H. (2011). Minimum Supply Voltage and Yield Estimation for Large SRAMs Under Parametric Variations. *TVLSI*. DOI: 10.1109/TVLSI.2010.2071890.

Wang, Z., Bovik, A. C., Sheikh, H. R., e Simoncelli, E. P. (2004). Image quality assessment: from error visibility to structural similarity. *IEEE TIP*. DOI: 10.1109/TIP.2003.819861.

Yazdanbakhsh, A., Mahajan, D., Esmaeilzadeh, H., e Lotfi-Kamran, P. (2017). AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE D&T*. DOI: 10.1109/MDAT.2016.2630270.

# Gerador Parametrizável de Aceleradores para *K-means* em FPGA e GPU*

**Jeronimo Penha[1,2], Lucas Bragança[1], Kristtopher Coelho[1], Michael Canesche[1], Jansen Silva[1], Giovanni Comarela[1], José Augusto M. Nacif[1], Ricardo Ferreira[1]**

[1]Depto de Informática – Universidade Federal Viçosa, Brasil
[2] Depto de Informática – CEFET-MG, Campus Leopoldina, Brasil

***Resumo.*** *O algoritmo K-means é um método utilizado para aprendizado não supervisionado no agrupamento de dados. Este trabalho apresenta um gerador de código de domínio específico para o K-means capaz de gerar código para GPUs e FPGAs. Para aumentar a eficiência, o código é parametrizável e especializado para GPUs da Nvidia e para a plataforma HARP v.2 da Intel/Altera. Entretanto, o gerador é modular e pode ser estendido para outras plataformas de FPGA e GPU. Outra contribuição deste trabalho é simplificação do uso de FPGAs de alto desempenho para programadores, pois não requer nenhum conhecimento de hardware por parte do usuário para prover um acelerador de alto desempenho no nível de software. O gerador também simplifica a programação para GPU. Para os experimentos realizados, em comparação com o tempo de execução em uma CPU Intel XEON, a solução proposta em GPU acelerou em até 55 vezes e o FPGA obteve uma aceleração de até 13,8 vezes. Em relação à eficiência energética, as execuções em FPGA foram até 10 vezes mais eficientes que as GPUs avaliadas. Os resultados foram validados com duas GPUs: K40 e 1080ti.*

## 1. Introdução

Atualmente, as GPUs (*Graphics Processing Unit*) e os FPGAs (*Field-Programmable Gate Array*) são duas tendências em aceleradores para desempenho com eficiência energética em comparação aos processadores de uso geral [Neshatpour et al. 2015]. Neste contexto, a eficiência pode ser maior ao trabalhar com domínios específicos, conforme destacado por David Patterson e John Hennessy na aula do prêmio Turing de 2017 [ACM 2018]. Apesar dos FPGAs oferecerem maior potencial de eficiência energética, o maior desafio ainda é a sua programação que exige conhecimentos específicos de hardware, mesmo com os ambientes de alto nível como OpenCL [Tang and Khalid 2016].

Uma demanda atual é a extração de conhecimento e identificação de grupos de grandes volumes de dados. Este artigo aborda a implementação do algoritmo *K-means* [Forgy 1965] de aprendizado não supervisionado nas plataformas de GPU e FPGA. O algoritmo *K-means* possui potencial para a paralelização de operações com reúso de dados, que é uma característica apropriada para as arquiteturas alvo. A primeira contribuição deste trabalho é a apresentação de um gerador de código parametrizável para GPU para

*Financiamento: FAPEMIG, Fundação CEFETMINAS, Intel, NVIDIA, CNPq. Synopsys pela doação das licenças de software. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

a execução do *K-means*, que otimiza o reúso dos dados. A segunda contribuição é a geração automática e transparente de hardware e software para execução do algoritmo na nova plataforma CPU-FPGA HARP v.2 de alto desempenho da Intel.

A estrutura do artigo é descrita, a seguir. A Seção 2 apresenta brevemente o algoritmo *K-Means*, a Seção 3 descreve a plataforma HARP v.2 e a Seção 4 apresenta a arquitetura desenvolvida para execução em FPGA. Na Seção 5, o gerador para GPU é apresentado. Os experimentos e resultados são discutidos na Seção 6, os trabalhos relacionados são discutidos na Seção 7, e por fim, a Seção 8 apresenta as principais conclusões.

## 2. *K-means*

O *K-means* é um algoritmo de aprendizado não supervisionado capaz de particionar um conjunto de pontos em $k$ grupos. O algoritmo é utilizado em larga escala em aplicações de mineração de dados [Choi and So 2014]. A cada iteração, todos os pontos são avaliados. Várias iterações são executadas até que convirja. Este processo exige esforço computacional para grandes volumes de dados [Abdelrahman 2016].

A Figura 1 apresenta um exemplo com pontos de duas dimensões, $x$ e $y$, dos quais deseja-se classificar em dois grupos, i.e., $k = 2$. Cada grupo terá um centroide $C_i$ e cada ponto é classificado no grupo do centroide mais próximo. A Figura 1(a) mostra o ponto $p$ a ser classificado no grupo $G_0$ por estar mais perto do centroide $C_0$. A etapa de classificação é executada para todos os pontos. A próxima etapa é o cálculo da nova posição dos centroides. Essa nova posição é a que minimiza a média do quadrado das distâncias de todos os pontos do grupo. A Figura 1(b) mostra o reposicionamento dos centroides após a execução de uma iteração. Também é destacado o reposicionamento do centroide $C_0$ e os pontos classificados nos grupos $G_0$ e $G_1$ na etapa anterior. O algoritmo repete as duas etapas (classificação e reposicionamento) até que convirja ou atinja um limite de iterações.

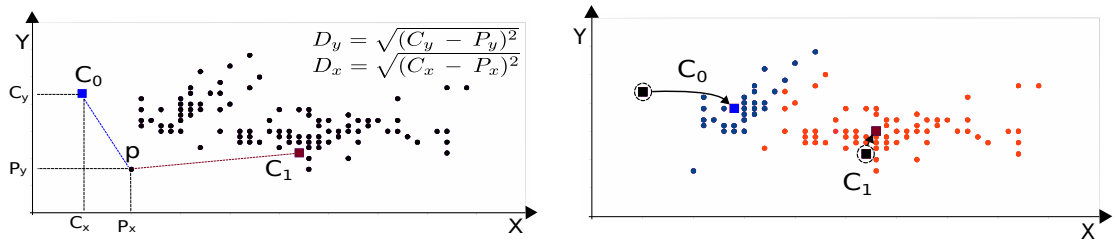



**Figura 1. Exemplo de agrupamento pelo *K-means*.**

O algoritmo pode ser descrito com o paradigma de *map-reduce*. A primeira etapa de classificação, pode ser definida por um mapeamento seguido de duas reduções. A Figura 2(a) mostra o fluxo de operações para o exemplo com duas dimensões ($n = 2$) e dois centroides ($k = 2$). O mapeamento é responsável pelo cálculo da distância em cada dimensão. A métrica de distância utilizada no *K-means* é a distância euclidiana, ou seja, através do cálculo de raiz quadrada do quadrado da diferença para cada dimensão. Para a dimensão $x$ em relação ao centroide $i$, será $D_{i_x} = \sqrt{(P_x - C_{i_x})^2}$. Esta operação básica é executada para cada ponto e todas as suas dimensões em relação a todos os centroides. Como é uma métrica de distância, uma simplificação pode ser

feita com a retirada do cálculo da raiz quadrada com o intuito de reduzir a complexidade [Zaki and Meira Jr 2014]. O segundo passo é uma redução para somar as distâncias de todas as dimensões do ponto em relação ao centroide. No exemplo, para os centroides $C_0$ e $C_1$, o cálculo será $D_0 = D_{0_x} + D_{0_y}$ e $D_1 = D_{1_x} + D_{1_y}$, respectivamente. O terceiro passo é a redução para encontrar o centroide mais próximo de $p$, ou seja, $Min(D_0, D_1)$. A função $Min$ retorna o número do centroide mais próximo de $p$. Esta é a fase de classificação do *K-means*. Formalmente, para $k$ centroides e $n$ dimensões, o ponto será classificado como $Min(D_0, \ldots, D_{n-1})$ onde a distância para o centroide $C_i$ será $D_i = \sum_{j=0}^{n-1}(p_j - C_{i_j})^2$. Assim, tem-se $kn$ subtrações e multiplicações durante o mapeamento, a redução de soma totaliza $k(n-1)$ adições e a redução de mínimo com $k-1$ comparações. A cada iteração para $m$ pontos, tem-se $m[2kn + k(n-1) + k - 1]$ operações. A primeira fase do algoritmo é apropriada para execução em GPU e FPGA que são capazes de executar todas as operações em paralelo, dependendo do código, da arquitetura e dos valores de $m, k$ e $n$. Outro aspecto favorável às GPUs e aos FPGAs é o reúso do dados que é definido pelo número de operações realizadas para cada dado lido da memória, que é igual $\frac{m[2kn+k(n-1)+k-1]}{mn} = 3k - \frac{1}{n}$, ou seja, quanto maior for o valor de $k$, o cenário se torna mais favorável às GPUs e aos FPGAs.

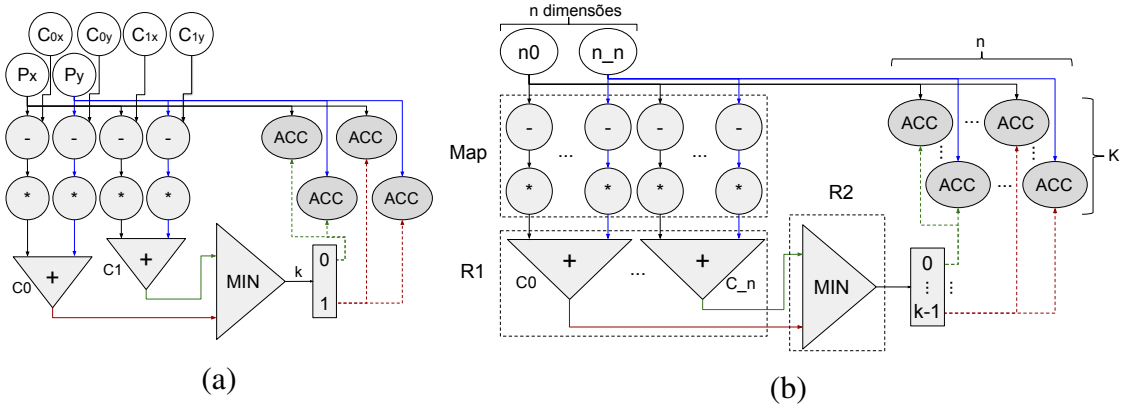



**Figura 2. (a) Grafo de operações com $k = 2$ e $n = 2$; (b) Grafo Genérico.**

A etapa de reposicionamento é responsável pelo cálculo dos novos valores para cada uma das dimensões dos centroides. Para cada ponto classificado pelo centroide $C_i$, os valores de cada dimensão são acumulados. Por isto há vários acumuladores ($Acc$) na Figura 2. Se $t$ pontos forem classificados para o centroide $C_i$, a nova posição na dimensão $j$ para o centroide $C_i$ será calculado por $C_{i_j} = \frac{\sum_{j=1}^{t} p_j}{t}$.

## 3. Plataforma de Alto Desempenho com FPGA HARP v.2

Recentemente, várias plataformas para computação de alto desempenho com FPGAs foram apresentadas por grandes empresas como Intel-Altera [Gupta 2016], Amazon [Amazon 2018] e Microsoft [Caulfield 2016]. A plataforma utilizada neste trabalho é o HARP v.2 da Intel-Altera, que possui um FPGA fortemente acoplado através da memória compartilhada com um processador Xeon E5-2600 v4. A Intel provê interfaces em *software* e *hardware* que mantêm a coerência de dados entre o processador e o FPGA.

A Figura 3(a) ilustra os principais pontos da arquitetura da plataforma HARP v.2. A Intel provê a API OPAE (*Open Programmable Acceleration Engine*) que traz os objetos

necessários para a comunicação do aplicativo com os aceleradores desenvolvidos para a plataforma. Entretanto, é necessário que o usuário possua conhecimentos específicos em FPGA e da documentação do OPAE. A ligação física entre memória e o acelerador é feita através do barramento Intel QPI (*Quick Path Interconnect*) mais dois barramentos PCI-e, nos quais são possíveis taxas de transmissão de dados na ordem de 16GB/s. O FPGA mantém uma *cache* interna de 64 Kbytes. A coerência com as memórias é feita de forma transparente. A CPU e o FPGA trabalham de forma assíncrona e independente.

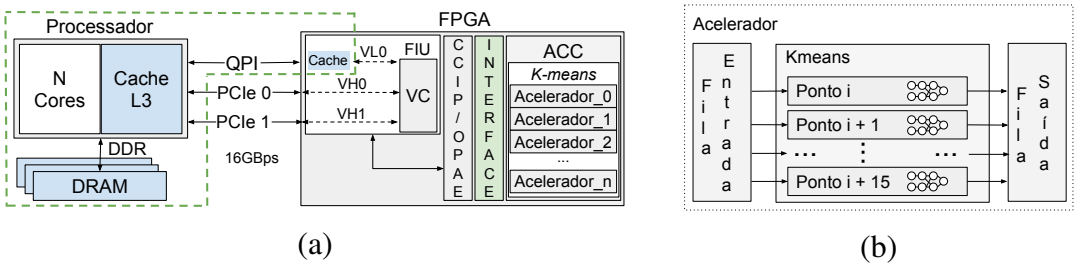



**Figura 3. (a) Plataforma HARP v.2 da Intel (b) Arquitetura do acelerador.**

Diferente da arquitetura de uma GPU que possui memória dedicada, o FPGA do HARP v.2 terá sempre que ler dados das memórias conectadas a CPU (*cache* L3 e RAM) através dos barramentos PCI e QPI, uma vez que a cache implementada dentro do FPGA é muito pequena, limitada a 64 Kbytes.

## 4. Arquitetura do Acelerador *K-means* no FPGA

A arquitetura do gerador é o mapeamento direto do grafo de fluxo de dados do algoritmo de forma a explorar o potencial de paralelismo espacial e temporal (*pipeline*). Além do grafo, as interfaces de entrada e saída são adicionadas, ilustradas na Figura 3(b), que fazem o acoplamento da plataforma de hardware com o código em execução no processador implementado para plataforma HARP v.2 da Intel. A interface pode ser estendida para outras plataformas. As interfaces são baseadas na unidade de hardware apresentada em [Bragança et al. 2018]. Como no HARP v.2, a granularidade da comunicação é limitada a uma linha de cache de 64 bytes, caso um ponto necessite menos de 64 Bytes, a arquitetura explora o paralelismo espacial com replicações, o que possibilita o processamento de vários pontos em paralelo para maximização da vazão de dados. Ademais, para mitigar a latência da comunicação, a interface possibilita o uso de filas e mais de uma instância do acelerador com execuções simultâneas.

Toda a arquitetura é gerada automaticamente de forma parametrizada em função do número de dimensões do conjunto de dados e do número de centroides. O gerador foi desenvolvido em Python com auxílio da biblioteca Verilogen [Takamaeda-Yamazaki 2015]. O hardware é gerado em Verilog e sintetizado com a ferramenta Intel Quartus para FPGAs. O gerador encapsula toda a complexidade de interface, sincronismo, compilação e síntese para FPGAs e a comunicação do hardware/software. O usuário precisa apenas instanciar os dados a serem processados e fazer a chamada ao acelerador a partir de software.

## 5. Gerador de código para GPU

Nesta seção é apresentado o gerador de código para o algoritmo *K-means* para GPU Nvidia em CUDA. O gerador é parametrizável em função do número de dimensões

$n$ e do número de centroide $k$. Cada *thread* é responsável pela classificação de um ponto e executa os seguintes passos: o cálculo da distância em relação a todos os centroides, a redução de soma e a redução para a determinação centroide mais próximo. Para evitar divergência com condicionais em GPU, foram usadas condicionais simples. A etapa de reposicionamento também é realizada na GPU.

```
__global__ void kmeans(int*in,
int *c, int *nC,
int *total, const int n) {
  int i;
  i = (blockIdx.x * blockDim.x +
       threadIdx.x) * DIM;
  // leitura do Ponto
  int pd0 = in[i + 0];
  int pd1 = in[i + 1];
  // Map para distancia
  int k0d0 = pd0 - c[0]; // C0_0
  int k0d1 = pd1 - c[1]; // C0_1
  int k1d0 = pd0 - c[2]; // C1_0
  int k1d1 = pd1 - c[3]; // C1_1
  //Quadrado Distancia
  k0d0 *= k0d0; k0d1 *= k0d1;
  k1d0 *= k1d0; k1d1 *= k1d1;
  // Reducao de Soma para Distancia
  k0d0=  k0d0+k0d1;// Distancia de C0
  k1d0=  k1d0+k1d1;// Distancia de C1
  // Reducao de Minimo
  int minId;
  minId = (k1d0 < k0d0) ? 1 : 0;
  // Inclusao do Ponto para
     Reposicionamento
  atomicAdd(&(nC[DIM*minId+0]),pd0);
  atomicAdd(&(nC[DIM*minId+1]),pd1);
  atomicAdd(&(total[minId]), 1);
}
```

**Figura 4. Código simplificado para *K-means* gerado para GPU com $k = 2$ e $n = 2$ realizando a redução de reposicionamento com operações atômicas.**

Ao iniciar o cálculo concorrente de milhares de *threads*, a GPU mascara a latência da leitura na memória e das dependências de dados no nível de instruções de todos os passos da etapa de classificação, incluindo o mapeamento e as duas reduções. O gargalo é a etapa final de reposicionamento que envolve a redução com um acumulador para cada dimensão de cada centroide. Duas versões foram propostas para a fase de reposicionamento dos centroides. Uma baseada nas operações atômicas que possuem suporte em hardware nas novas GPUs (a partir da geração Kepler) e outra baseada em memória compartilhada, com segmentação a partir de blocos e bancos distintos de memória compartilhada em função dos valores de $n$ e $k$. A Figura 4 ilustra o código gerado para o exemplo com duas dimensões e dois centroides na versão mais simples baseada em operações atômicas.

## 6. Experimentos e Resultados

Para avaliar os geradores propostos foi usada a base de dados *US Census 1990* [Dheeru and Karra Taniskidou 2017]. Esta base de dados também foi usada em outros trabalhos com FPGA e GPU [Li et al. 2016, Kaplan et al. 2018, Abdelrahman 2016]. O tamanho da palavra utilizado para cada dimensão foi de 16 bits e a base tem 2 milhões de pontos. Para avaliar a variação do desempenho em função do número de centroides e do número de dimensões, utilizamos combinações valores de $n$ e $k$ variando na faixa 2, 4, 8, 16 e 32.

Os resultados foram comparados com a execução em uma CPU Intel Xeon E5-2630 v3 @ 2.40GHz com uma implementação do *K-means* em C, semelhante ao código gerado para GPU, compilado com GCC 4.8.4, com a opção -O3, executando como uma *thread*. O tempo de execução foi medido com a utilização de *chrono::high_resolution_clock*. Além disso, três outras implementações foram utilizadas como referência. A primeira

é a implementação popular para execução do *K-means* encontrada no pacote *Scikit-learn* [Pedregosa et al. 2011]. A segunda é a implementação em OpenMP com 16 *threads* disponibilizada pelo pacote Rodinia, versão 3.1 [Che et al. 2009, Rodinia 2016]. A terceira é a implementação em GPU/CUDA também do pacote Rodinia, versão 3.1, executada na GPU K40. O código CPU, usado como referência, foi em média 6,5 vezes mais rápido que a implementação *Scikit-learn*. A versão CPU foi em média 3,42 e 6,83 vezes mais lenta que as implementações Rodinia em OpenMP e GPU/CUDA, respectivamente. A plataforma Intel HARP v.2 possui um FPGA Arria 10 modelo 10AX115U3F45E2SGE3. Duas GPUs foram utilizadas, uma Nvidia Tesla K40c com 12GB de memória RAM e uma frequência de relógio de 745 MHz e uma Nvidia Pascal 1080ti com 11 GB e uma frequência de relógio de 1.584 GHz. Todos os experimentos foram conduzidos com 10 repetições e são apresentados os resultados médios.

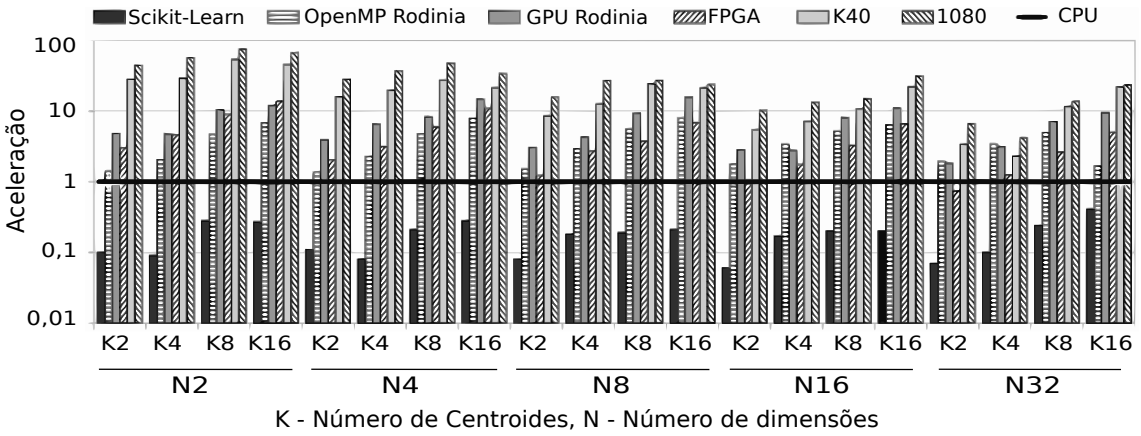



**Figura 5. Acelerações das GPUs K40 e 1080 e do FPGA normalizadas em relação ao tempo de execução com um *thread* em CPU. Escala logarítmica.**

A Figura 5 mostra os resultados do ganho de aceleração agrupados pelo número de dimensões. Pode-se perceber que o ganho de aceleração das GPUs e do FPGA em relação a CPU para conjunto de pontos com a mesma quantidade de dimensões aumenta em função de $k$, o motivo é o reúso dos dados de $3k - \frac{1}{n}$ operações por byte. Para cada grupo avaliado, as três primeiras barras são os resultados do *scikit-learn*, Rodinia OpenMP e GPU. As três últimas barras são os resultados dos geradores de código propostos para o *K-means*: FPGA, K40 e 1080ti. Os tempos de execução foram normalizados em relação ao tempo de execução da CPU, usado como referência. Quanto maior o valor, melhor é o resultado. A linha em negrito com o valor 1 simboliza o tempo de execução na CPU. As GPUs K40 e 1080ti chegam a ser até 54 e 67 vezes mais rápida que a CPU. O FPGA foi até 13,85 mais rápido. Pode-se observar que a implementação padrão *scikit-learn* é, em média, 63 vezes mais lenta que o FPGA, 245,5 e 250 mais lenta que as GPUs K40 e 1080ti, respectivamente. Em relação a implementação Rodinia para GPU, os geradores para a K40 e a 1080ti foram, em média, de 3 e 4 vezes mais rápidos, respectivamente. Em relação a versão Rodinia OpenMP com 16 *threads*, os geradores com a K40 e a 1080ti foram, em média, de 6 e 8 vezes mais rápidos.

O maior limitador da plataforma Intel HARP v.2 é a leitura de dados. Como o FPGA não tem memória local, todas as leituras são feitas da memória da CPU a cada iteração. Além disso, para $k = 2$ com 32 dimensões, o conjunto de dados é 16 vezes maior

que para 2 dimensões. Na GPU, os dados são transferidos uma vez pelo barramento PCI e permanecem na memória global da GPU até o fim de todas as iterações. A memória global das GPUs tem uma vazão de 200-400 GB/s, o que é significativamente maior que transferência CPU-FPGA, que fica em torno de 16 GB/s. O tempo de execução inclui todas as transferências de dados. É importante ressaltar que o HARP v.2 é um protótipo que pode não refletir o desempenho de sistemas futuros da Intel. Recém lançado em maio de 2018, o processador Intel Xeon 6138P traz integrado um FPGA Arria 10 [Intel 2018] e pode atingir taxas de transferência de 160 GB/s, 10 vezes mais que o HARP v.2.

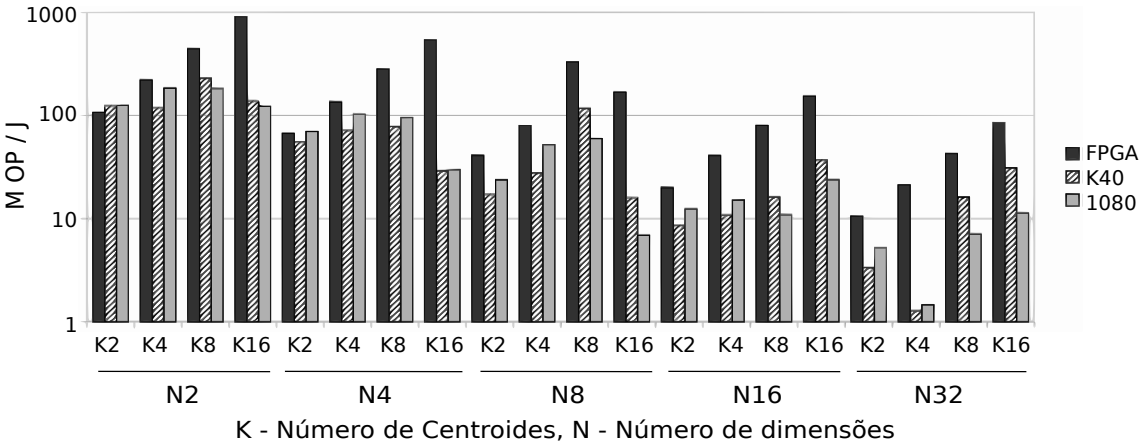



**Figura 6. Eficiência energética para a execução dos *kernels* em GPU e em FPGA.**

O potencial de eficiência energética do FPGA é outro ponto importante. A plataforma HARP v.2 tem um consumo médio de 22,351 Watts aferidos durante as execuções com a ferramenta disponibilizada pela API OPAE da Intel. Como referência para comparação foi utilizada uma estimativa de potência aferida para GPU K40 disponibilizada em [Ferro et al. 2017] de 134 Watts. A GPU 1080ti tem o consumo de 300 Watts. Com base nestes dados, a Figura 6 apresenta a eficiência energética como a quantidade de milhões de operações (MOPs) por Joule obtidos em cada experimento para uma iteração nas GPUs e no FPGA. Para um mesmo valor de $n$, quanto maior o reúso, mais eficiente é a solução. Pode-se observar que a solução com FPGA é promissora, mesmo sendo um protótipo da Intel com limitações de leitura de memória. Este fato também corrobora o interesse das grandes empresas nas soluções com FPGA em nuvem, como por exemplo a Microsoft [Caulfield 2016] e a Amazon [Amazon 2018].

Como já mencionado, o HARP v.2 é ainda um protótipo. Ao realizar a chamada do acelerador, a sobrecarga de tempo da API Intel domina o tempo de execução. A Figura 7 exibe o tempo de execução da chamada do *kernel* do acelerador para cálculo de uma iteração no FPGA incluindo a transferência dos dados, mas não considera o tempo gasto pela API de software da Intel para a realização da inicialização entre as chamadas sucessivas do acelerador (entre as iterações). Em média, o tempo da execução, incluindo a transferência de dados, é de apenas 30% do tempo da iteração. A sobrecarga das APIs é aproximadamente de 70% do tempo da iteração. Para GPU, a sobrecarga das chamadas é pequena, onde o tempo da execução e da transferência é da ordem de 95% ou mais. Ao desconsiderar a sobrecarga da API, o FPGA foi mais rápido que a GPU K40 em três casos, apesar de ter uma taxa de leitura de memória 13,75 vezes menor e uma frequência

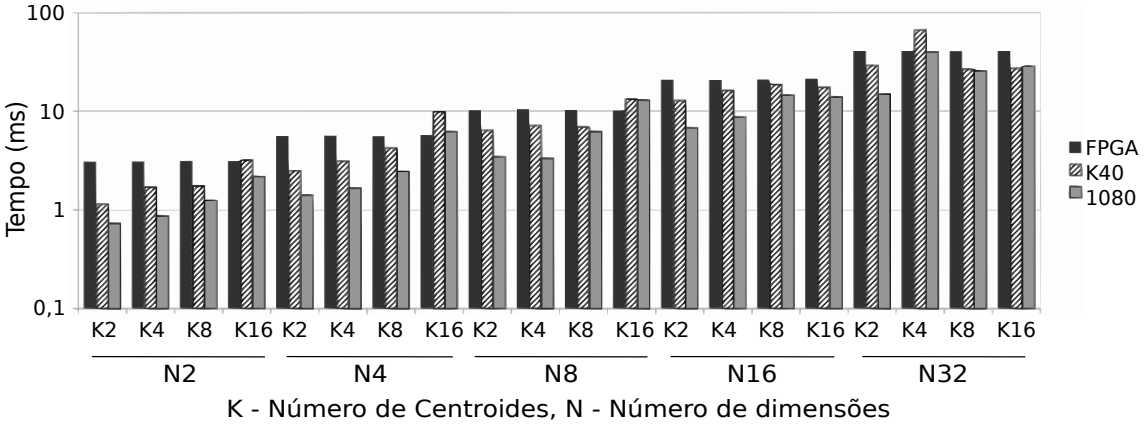



**Figura 7. Tempos de execução da chamada dos aceleradores.**

de relógio de 200 MHz em comparação com os 745 MHz da K40. Já a GPU 1080ti é mais rápida pois possui uma frequência de relógio de 1,58 GHz e uma taxa de leitura de memória de 485 GB/s.

Vale ressaltar uma diferença da Figura 7, na qual o tempo não varia em função de $k$ para um dado $n$, do comportamento dos resultados anteriores. Nas Figuras 5 e 6, a eficiência energética e a aceleração aumentam em função de $k$. Na Figura 5, a medida é relativa ao tempo da CPU, portanto apesar do tempo de execução do FPGA não variar para mesmo $n$, o tempo da CPU varia, aumentando em função de $k$ pois não tira proveito do paralelismo de calcular todas as distâncias ao mesmo tempo. No caso da Figura 6, como número de operações aumenta em função de $k$, a eficiência aumenta, uma vez que o tempo de execução não se altera para um dado $n$. Ao dobrar o valor de $n$ dobra-se a quantidade de dados. Por exemplo, para 2 milhões de pontos e $n = 2$, tem-se 8 MB de dados e para $n = 4$ teremos 16 MB. Outra observação é a redução da quantidade de cópias do algoritmo por linha de cache com o aumento da quantidade de dimensões. A linha de *cache* do HARP v.2 é de 64 Bytes. Com $n = 2$, cada ponto consome 4 bytes e pode-se instanciar 16 cópias que processam 16 pontos em paralelo. Para $n = 4$, cada ponto terá 8 bytes e tem-se 8 cópias do processo por linha de *cache*.

Ao analisar o aumento de $k$ para um $n$ fixo, pode-se observar que com mais centroides, há um maior reúso dos dados. Para $k = 2$ e $n = 2$, a Figura 8(a) mostra que existem 11 operações por ponto e com $k = 4$, na Figura 8(b), são 23 operações por ponto. Desse modo, o mesmo ponto será comparado com 4 centroides simultaneamente, o que explora o paralelismo e aumenta o reúso.

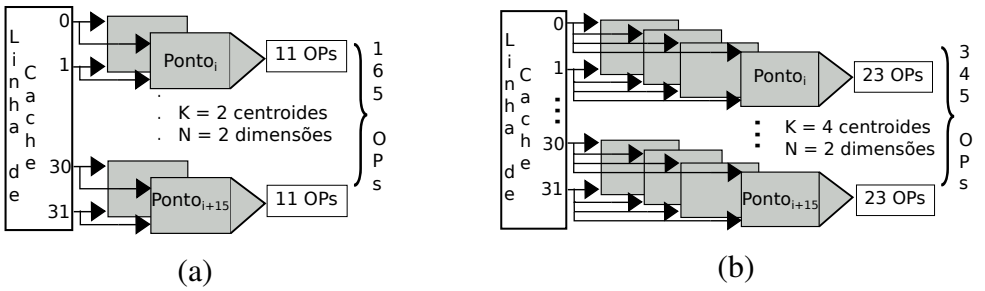



**Figura 8. Impacto do valor de $k$: (a) 165 Operações por 64 bytes; (b) 345 ops.**

A Figura 9 apresenta a quantidade de recursos do FPGA gastos em função dos valores de $k$ e $n$, que é um aspecto favorável aos FPGAs. São apresentadas apenas a quantidade de ALMs (blocos lógicos) e de DSPs (unidades lógica-aritméticas), pois o número de módulos de memória não varia, uma vez que são utilizados apenas pela interface com a CPU. As ALMs são usadas para construir a interface, controle, operações de soma e redução de mínimo. Em média, 18-20% das ALMs são usadas pela interface da Intel e apenas 2-6% das ALMs são gastas pelo gerador para implementar o controle e as operações de soma e redução. O maior valor de $k$ é apenas limitado pela quantidade de DSPs disponíveis no FPGA utilizado. Entretanto, existe uma perspectiva favorável aos FPGAs, pois o novo FPGA Stratix X da Intel possui 6760 DSPs [Stratix 2018] em comparação com os 1518 do Arria 10 do HARP v.2. Além do aumento da quantidade de DSP, a flexibilidade do FPGA permite que futuras extensões do gerador façam uso de aritmética dedicada com ALMs (que são abundantes) para o cálculo do quadrado das distâncias ao invés dos DSPs como multiplicadores. Portanto, com vazão de dados, mais DSPs, o desempenho do FPGA para o *K-means* pode aumentar significativamente.

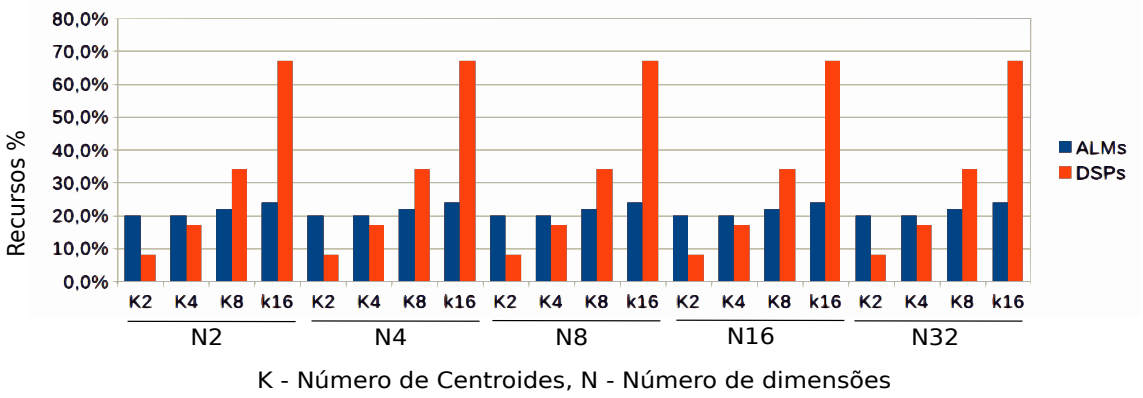



**Figura 9. Recursos do FPGA utilizados pelos aceleradores.**

## 7. Trabalhos Relacionados

Esta seção apresenta um estudo das implementações do *K-means* em FPGAs e GPUs. Um gerador parametrizável em função do número de dimensões e centroides para FPGA foi proposto em [Amaricai 2017]. Entretanto, o artigo não apresenta resultados de execução, apenas estimativas e simulação. É importante ressaltar que uma descrição que pode ser simulada em FPGA, não necessariamente irá executar corretamente. Além disso, o tempo de sincronização, chamada, leitura e escrita de dados pode não estar incluso. Uma estimativa para execução do *K-means* em FPGA é apresentada em [Neshatpour et al. 2016], também sem resultados concretos com execução. Outra implementação em uma plataforma CPU/FPGA Xilinx é apresentada em [Neshatpour et al. 2015]. A aceleração é estimada em função do potencial de aceleração do FPGA e da CPU, inclusive a comunicação, o que mostra um potencial de aceleração de até 150x para o *K-means*, porém o sistema não foi avaliado em tempo de execução. Uma abordagem em alto nível para FPGA com a utilização da linguagem OpenCL é apresentada em [Tang and Khalid 2016], na qual a implementação utiliza uma função para a classificação e outra para o reposicionamento dos centroides. A classificação utiliza a distância de Manhattan no lugar da distância Euclidiana. Esta

proposta apresenta melhora de desempenho apenas para valores grandes de $k$, maiores que 128. Em [Maciel et al. 2017], um acelerador para FPGA foi implementado, onde os parâmetros $k, n$ e número de iterações podem ser alterados em tempo de execução. Porém o formato dos dados é fixo (64 bits), o valor máximo está limitado a apenas 16K pontos e foi apenas prototipado em uma placa de baixa custo com cálculo paralelo de apenas 2 pontos por vez, o que restringe seu uso em problemas reais. Para $k = 8$ e $n = 4$, o desempenho é de apenas 6M operações/s a 50 MHz, o gerador proposto neste trabalho executa 9,7G operações/s à 200 MHz, ou seja, 250 vezes mais rápido que o acelerador proposto por [Maciel et al. 2017] e não está limitado a 16K entradas.

Como não existe uma padronização, é difícil realizar uma comparação entre as implementações. A comparação será feita através de bases de dados de mesmo tamanho e considerando o tempo de uma iteração no qual o número de operações realizadas é equivalente. A arquitetura FPGA proposta em [Li et al. 2016] é baseada no paradigma *Map-Reduce* onde a avaliação foi feita com *benchmarks* de apenas 2 e 4 dimensões para $k = 4$. Ao comparar com os resultados apresentados na Seção 6, o gerador proposto neste trabalho apresenta tempos de execução duas vezes mais rápidos que resultados de [Li et al. 2016] para $n = 2$ e $4$. Ademais, o gerador é genérico e não foi dedicado a dois valores de $n$. Uma arquitetura com elementos de processamento para o *K-means* em FPGA foi proposta em [Lee et al. 2017], que é pelo menos duas vezes mais lenta que o gerador proposto neste trabalho para $k = 3$ e $n = 10$. O trabalho apresentado em [Abdelrahman 2016] é o mais próximo da abordagem proposta em termos da plataforma de FPGA. Os resultados também são avaliados em tempo de execução. O FPGA é o protótipo da HARP v.1, já descontinuado. O FPGA é usado para calcular a etapa de classificação e a CPU calcula o reposicionamento. Apenas $k = 8, n = 2$ e $k = 4, n = 4$ são avaliados para os quais a abordagem proposta neste trabalho em FPGA é 20 vezes ou mais rápida que o resultado apresentado em [Abdelrahman 2016]. Uma biblioteca para GPU é apresentada em [Bhimani et al. 2015], porém os resultados são pelo menos 10 vezes mais lentos da solução em FPGA e na GPU K40 apresentada nesse trabalho. Em relação a implementação proposta para GPU 1080ti, os resultados do trabalho proposto em [Bhimani et al. 2015] foram 50 vezes mais lentos. O maior problema é que parte da classificação e reposicionamento são implementados na CPU. Uma comparação entre as soluções com CPU, GPU e FPGA foi apresentada em [Huang 2017], onde a solução com FPGA foi a mais rápida executando na Nuvem da Amazon [Amazon 2018]. A solução apresentada por este trabalho em FPGA é genérica com resultados equivalentes em desempenho considerando $k = 8$ e $n = 5$ aos apresentados para uma abordagem específica em [Huang 2017]. Como já apresentado na Figura 5, a implementação Rodinia para GPU [Rodinia 2016] foi de 3 à 4 vezes mais lenta que o gerador proposto neste trabalho.

## 8. Conclusão

Este trabalho apresenta um gerador parametrizável do algoritmo *K-means* para FPGAs e GPUs. O desempenho é maximizado ao se especializar em um domínio específico e explorar o paralelismo espacial e temporal. Apesar de ser específico, existe uma demanda por agrupamento de dados, onde o *K-means* e suas variações têm muitas aplicações. Os resultados experimentais mostraram ganho em tempo de execução em comparação a CPU de até 54,55 e 13,85 vezes para as GPUs k40 e 1080ti e a plataforma

HARP v.2, respectivamente. O gerador executou com uma frequência de 200 MHz no FPGA, mas pode ser ajustado para executar a 400 MHz, que dobrará seu desempenho. Apesar do desempenho do FPGA ter sido limitado pelas baixas taxas de transmissão de 16 GB/s na leitura de memória do protótipo HARP v.2 da Intel, existe um potencial para os novas plataformas FPGAs da Intel e da Xilinx onde o desempenho da transferência de dados deve melhorar significativamente.

Como trabalhos futuros, há dois aspectos importantes do algoritmo *K-means* que podem ser explorados: a determinação da quantidade de centroides e a seleção de um subconjunto relevante de atributos. Essas duas funcionalidades podem ser adicionadas de forma eficiente às implementações aqui propostas, tanto em GPU quanto FPGA, uma vez aumentarão ainda mais o reúso de dados. Outra possibilidade é a computação aproximada que vem sendo explorada em vários algoritmos de aprendizado de máquina. Pode-se esperar que a flexibilidade do FPGAs tenha um nicho melhor neste domínio que as GPUs e as CPUs. Para a nova geração *Volta* de GPUs da Nvidia [Jouppi et al. 2017], existe ainda um potencial a ser explorado pelo *K-means* com os operadores TPU que fazem multiplicação e acúmulo de matrizes com estruturas sistólicas. Com relação às CPUs, o *K-means* pode explorar paralelismo a nível dos múltiplos núcleos e também o paralelismo no nível de instrução com as extensões vetoriais AVX.

## Referências


Abdelrahman, T. S. (2016). Accelerating K-Means Clustering on a Tightly-Coupled Processor-FPGA Heterogeneous System. In *IEEE International Conference ASAP*.

ACM (2018). In *ACM Award 2017* `https://amturing.acm.org/`.

Amaricai, A. (2017). Design Trade-offs in Configurable FPGA Architectures for K-Means Clustering. *Studies in Informatics and Control*, 26(1):43–48.

Amazon (2018). In *Elastic Compute Cloud - Amazon EC2 - AWS http://aws.amazon.com/ec2/*.

Bhimani, J., Leeser, M., and Mi, N. (2015). Accelerating K-Means Clustering with Parallel Implementations and GPU Computing. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*.

Bragança, L., Alves, F., Penha, J. C., Coimbra, G., Ferreira, R., and Nacif, J. A. M. (2018). Simplifying HW/SW Integration to Deploy Multiple Accelerators for CPU-FPGA Heterogeneous Platforms. In *IEEE SAMOS*.

Caulfield, A. M. e. a. (2016). A Cloud-Scale Scceleration Architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE Int. Symposium on Workload Characterization*.

Choi, Y.-M. and So, H. K.-H. (2014). Map-Reduce Processing of K-Means Algorithm with FPGA-Accelerated Computer Cluster. In *IEEE International Conference ASAP*.

Dheeru, D. and Karra Taniskidou, E. (2017). In *UCI Machine Learning Repository http://archive.ics.uci.edu/ml*.

Ferro, M. et al. (2017). Analysis of GPU Power Consumption Using Internal Sensors. In *Workshop em Desempenho de Sistemas Computacionais e de Comunicacao*.

Forgy, E. W. (1965). Cluster Analysis of Multivariate Data: Efficiency Versus Interpretability of Classifications. *Biometrics*, 21:768–769.

Gupta, P. (2016). Accelerating Datacenter Workloads. In *Field Programmable Logic and Applications, Keynote - Slides available at www.fpl2016.org*.

Huang, K. (2017). K-Means Parallelism on FPGA. Master's thesis, Northeastern University, Boston, USA.

Intel (2018). In *Introducing the Intel Xeon Scalable processor with integrated Arria 10 FPGA https://itpeernetwork.intel.com/intel-processors-fpga-better-together/.*

Jouppi, N. P., , et al. (2017). In-Datacenter Performance Analysis of a Tensor Processing Unit. In *International Symposium on Computer Architecture (ISCA)*.

Kaplan, R., Yavits, L., and Ginosar, R. (2018). Prins: Processing-in-Storage Acceleration of Machine Learning. *IEEE Transactions on Nanotechnology*.

Lee, D., Althoff, A., Richmond, D., and Kastner, R. (2017). A Streaming Clustering Approach Using a Heterogeneous System for Big Data Analysis. In *IEEE/ACM ICCAD*.

Li, Z., Jin, J., and Wang, L. (2016). High-Performance K-Means Implementation Based on a Simplified Map-Reduce Architecture. *arXiv preprint arXiv:1610.05601*.

Maciel, L. A., Souza, M. A., and de Freitas, H. C. (2017). Projeto e Avaliação de uma Arquitetura do Algoritmo de Clusterização k-means em vhdl e fpga. *WSCAD*.

Neshatpour, K. et al. (2015). Energy-Efficient Acceleration of Big Data Analytics Applications Using FPGA. In *IEEE International Conference on Big Data*.

Neshatpour, K. et al. (2016). Big Biomedical Image Processing Hardware Acceleration: A Case Study for K-Means and Image Filtering. In *Int. Symposium on Circuits and Systems*.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, et al. (2011). Scikit-Learn: Machine Learning in Python. *Journal of machine learning research*, 12.

Rodinia (2016). In *Version 3.1* `http://lava.cs.virginia.edu/Rodinia/download_links.htm`.

Stratix (2018). In *Device Overview https://www.altera.com/products/fpga/stratix-series/.*

Takamaeda-Yamazaki, S. (2015). Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL. In *Applied Reconfigurable Computing*.

Tang, Q. Y. and Khalid, M. A. (2016). Acceleration of K-Means Algorithm Using Altera SDK for OpenCL. *ACM Transactions on Reconfigurable Technology and Systems*, 10(1):6.

Zaki, M. J. and Meira Jr, W. (2014). *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press.

,

# Análise da Utilização de Simuladores para Estimar o Consumo Energético em Ambientes de Cloud Computing

**Vinícius Meyer, Rafael Krindges,**
**Tiago C. Ferreto, Cesar A. F. De Rose, Fabiano Hessel**

[1]Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS, Brasil

{vinicius.meyer.001,rafael.krindges}@acad.pucrs.br,

{tiago.ferreto, cesar.derose, fabiano.hessel}@pucrs.br

***Resumo.*** *A computação em nuvem é uma das tecnologias mais populares por conta dos vários benefícios proporcionados para empresas de TI no mundo. No entanto, diante das crescentes solicitações dos usuários por serviços de computação, os provedores de nuvem demandam um consumo de energia cada vez maior. Para avaliar o consumo energético de grandes centros de dados, este artigo explora, através da utilização de simuladores de ambientes de computação em nuvem, diversas configurações de recursos e como estas impactam no consumo total de energia. Os resultados mostraram que os simuladores conseguem estimar custos de energia para diversos tipos de infraestruturas. Ainda, é apresentada uma avaliação dos recursos que cada simulador utiliza do hardware no tempo de execução.*

## 1. Introdução

A tecnologia da computação em nuvem é cada vez mais aprimorada, devido à flexibilidade da utilização de recursos de computação sem a necessidade de infraestruturas próprias. O uso dessa tecnologia gera, para o usuário uma maior segurança, confiabilidade e disponibilidade de recursos sob demanda [Makaratzis et al. 2018]. Diversas organizações têm migrado seus serviços de *datacenters* próprios para *cloud computing* por conta destes benefícios. Entretanto, para os provedores destes serviços, um fator importante a ser analisado é o consumo energético, uma vez que a quantidade de energia consumida nos grandes centros de dados está aumentando significativamente [Makaratzis et al. 2018]. Estes centros de dados em larga escala têm enormes orçamentos de energia que devem ser avaliados [Dayarathna et al. 2016]. No estudo realizado por [Guyon et al. 2017], afirma-se que cerca de 0,5% do consumo mundial de energia refere-se a *datacenters* em nuvem, embora se espere que esse valor quadruplique em 2020.

Para avaliar diferentes estratégias de utilização de recursos e quantidade de energia consumida, ferramentas de simulação de sistemas distribuídos [Casanova 2001] [Kliazovich et al. 2010] [Calheiros et al. 2011] [Tighe et al. 2012] [Nuñez et al. 2012] foram propostas durante os últimos anos, fornecendo plataformas para realizar experimentações. A simulação economiza as despesas da experimentação em infraestruturas reais de nuvem, uma vez que testar o problema em recursos reais pode se tornar extremamente complexo. Assim, os simuladores de nuvens fornecem simplicidade, repetibilidade de experimentos e reduzem os custos de experimentos [Dayarathna et al. 2016].

Neste artigo é apresentada uma comparação da utilização de modelos de energia em simuladores para avaliar o consumo energético de ambientes de computação em

nuvem. Diversas configurações de recursos foram criadas para explorar o que de fato impacta mais para o aumento do consumo energético. Os resultados aqui obtidos foram comparados com um artigo considerado relevante para a área e para este estudo. Ainda, os mesmos cenários foram testados utilizando o simulador de sistemas distribuídos SimGrid com o intuito de verificar se o mesmo demonstra um comportamento similar em relação ao consumo de energia. Durante a instalação e utilização dos simuladores avaliados foi percebido que alguns apresentaram algumas limitações. Assim, também avaliamos como estes simuladores se comportam em relação ao uso dos recursos do computador em que estão sendo executados.

O restante deste documento está organizado da seguinte maneira. A Seção 2 introduz os simuladores *open source* para computação em nuvem disponíveis na literatura. Na Seção 3 estão expostos os trabalhos relacionados relevantes ao estudo desenvolvido. A Seção 4 expõem a relação entre o consumo de energia em diversas configurações de centro de dados realizada com a utilização de simuladores. Esta seção também explora o uso dos recursos que os simuladores fazem sobre o *hardware* em que estão sendo executados. Por fim, a Seção 5 apresenta a conclusão deste estudo e direções para trabalhos futuros.

## 2. Simuladores para Computação em Nuvem

Esta seção tem como objetivo apresentar os simuladores de computação em nuvem abordados neste estudo.

### 2.1. Simuladores para Computação em Nuvem

Após pesquisa realizada sobre simuladores para ambiente de nuvem computacional, relacionamos alguns considerados mais importantes. Os critérios de seleção foram: (i) implementar modelo de consumo de energia, (ii) ser *open source* e estar disponível para *downloads*/testes e (iii) número de citações que o artigo original do simulador obteve. Com base nestes critérios foram selecionados os seguintes *middlewares*:

- **CloudSim**[1] - É a ferramenta mais utilizada para a modelagem e simulação de ambientes de computação em nuvem. Se trata de um simulador de eventos discretos que fornece um mecanismo de virtualização com recursos abrangentes para modelar o gerenciamento da utilização de máquinas virtuais em grandes infraestruturas. Não só inclui políticas de provisionamento de máquinas virtuais (VMs) nas máquinas físicas mas, também, programação dos recursos dos *hosts* entre máquinas virtuais, possibilidade de escalonamento de tarefas entre máquinas virtuais e modelagem de custos envolvidos nas operações realizadas [Calheiros et al. 2011].
- **DCSim**[2] - Simulador que possibilita modelar VMs replicadas que compartilham carga de trabalho de entrada sobre uma infraestrutura multicamadas. Fornece métricas para avaliar violações de *Service Level Agreement* (SLA) e de desempenho. Um ponto importante é que o DCSim é um simulador de eventos discretos para a modelagem de *datacenters* virtualizados que opera como Infraestrutura como Serviço (IaaS), provendo subsídios para locatários com foco em cargas de

[1] http://cloudbus.org/cloudsim/
[2] https://github.com/digs-uwo/dcsim

trabalho interativas, como aplicativos *web*. Neste simulador, um *datacenter* consiste em um conjunto de *hosts* interconectados (máquinas físicas), regidos por um conjunto de Políticas de Gerenciamento [Tighe et al. 2012].

- **GreenCloud**[3] - Sistema de simulação construído sobre o simulador de rede NS2[4] e que tem como principal objetivo simular a troca de pacotes entre os processos em execução em ambiente de nuvem computacional. Tendo em vista que este sistema foi construído sobre o NS2, o mesmo implementa um modelo de referência do protocolo TCP/IP completo que garante a integração de diferentes protocolos de comunicação, que fornece um melhor controle de grãos e não está presente em qualquer ambiente de simulação de computação em nuvem. Ele também fornece uma modelagem detalhada e análise da energia consumida pelos elementos dos servidores de rede, roteadores e as ligações entre eles [Kliazovich et al. 2010].
- **iCanCloud**[5] - Uma plataforma de simulação destinada a modelar e simular sistemas de computação em nuvem, voltada para os usuários que lidam de perto com esses tipos de sistemas. Foi implementada sobre o simulador OMNET[6]. O simulador iCanCloud foi criado para fornecer um conjunto de recursos avançados: realizar grandes experimentos; fornecer um *hypervisor* global flexível e, totalmente personalizável para integrar qualquer política de intermediação na nuvem; reproduzir os tipos de instâncias fornecidos pela infraestrutura de nuvem fornecida e, fornecer uma interface gráfica amigável para configurar e iniciar simulações. Uma característica do iCanCloud é o suporte de execuções de simulações paralelas, onde um experimento pode ser executado abrangendo várias máquinas distribuídas [Nuñez et al. 2012].

## 3. Trabalhos Relacionados

Grandes centros de dados são considerados infraestruturas críticas, demandando um alto consumo de energia. Assim, modelos de energia são importantes para otimizar e projetar as operações com eficiência energética. No estudo realizado por [Dayarathna et al. 2016], foram pesquisadas técnicas utilizadas para a modelagem do consumo de energia para *data centers* e seus componentes subjacentes em mais de 200 modelos. Ainda, o trabalho é divido em modelos de energia centrados em *software* e centrados em *hardware*. Dentro deste escopo foram investigados modelos para sistemas operacionais, máquinas virtuais e aplicações. Já, [Li et al. 2017] selecionaram, avaliaram e sintetizaram sistematicamente 76 estudos considerados relevantes, racionalizando e organizando mais de 30 modelos encontrados com notações unificadas. Para ajudar na predição de modelos de energia de trabalhos futuros, os autores desconstruíram o ambiente de execução e implementação em tempo de execução de aplicativos em nuvem e identificaram que existem fatores ambientais (tais como velocidade do disco, frequência da memória principal, números de núcleos de processadores, entre outros) e fatores de carga de trabalho (como tamanho da tarefa, número de tarefas, entre outros) que influenciam no consumo de energia.

Levando em consideração que mecanismos de pesquisa *web* são sistemas complexos que operam em grandes *clusters* de processadores, criados para gerenciar requisições

[3]https://greencloud.gforge.uni.lu/
[4]http://www.isi.edu/nsnam/ns/
[5]http://icancloud.org/
[6]https://www.omnetpp.org/

de consultas altamente dinâmicas e imprevisíveis, [Orellana et al. 2017] apresentam um método para estimar o consumo de energia de processadores para consultas *web* em ambientes simulados. Como os cálculos realizados neste ambiente apresentam padrões bem definidos, é possível prever o desempenho de maneira precisa. O método proposto é baseado no modelo Multi BSP para capturar os custos de *hardware* e inclui um modelo de consumo de energia que considera parâmetros de operações de custo dominantes e consultas de usuários. O trabalho feito por [O'brien et al. 2017] sintetiza esses esforços de pesquisa em modelos preditivos de energia, com ênfase principal na arquitetura de nós. Por meio dessa pesquisa, também destacam as deficiências desses modelos para prever de forma correta e abrangente os consumos de energia, levando em conta a natureza hierárquica e heterogênea dos sistemas de computação de alto desempenho altamente integrados.

Já [Alshammari et al. 2018] realizaram um estudo que aplica uma abordagem para avaliar qualquer simulador de nuvem por meio de um método de validação comparativa em centro de dados. Os autores estenderam trabalhos encontrados na literatura sobre avaliação de simuladores de computação em nuvem, aplicando um método para avaliar o CloudSim em outras ferramentas. Também apresentam o nível de precisão das plataformas de simulação GreenCloud e Mininet aplicando uma análise de sensibilidade para os resultados no desempenho real e simulado. Como conclusões afirmam que o GreenCloud não prevê o consumo de energia para um micro centro de dados com precisão. Por outro lado, Mininet mostra precisão razoável na modelagem do desempenho da rede. [Makaratzis et al. 2018] realizaram um estudo sobre simuladores de nuvens, com o intuito de examinar os diferentes modelos de energia que são usados para os componentes de *hardware* que constituem um centro de dados em nuvem. O foco é dado nos modelos de energia que foram propostos para a previsão do consumo de energia dos componentes do *datacenter*, como CPU, memória, armazenamento e rede. Os experimentos foram realizados utilizando diversos simuladores *open source* com o objetivo de comparar os diferentes modelos de energia utilizados por eles.

## 4. Relação entre simuladores

Com o intuito de avaliar o consumo de energia gerado por grandes centros de dados, realizamos uma pesquisa na literatura sobre os simuladores disponíveis de ambientes de computação em nuvem. Após a pesquisa decidiu-se utilizar, conforme critérios já citados, os quatro simuladores a seguir: CloudSim (versão 3.0.3), DCSim (versão lançada em 2014 - única até o momento), GreenCloud (versão 2.1.2) e iCanCloud (versão 1.0).

### 4.1. Cenários e Testes

Para explorar os modelos de energia que cada plataforma de simulação apresentada possui, diversos experimentos foram conduzidos. Como o objetivo é simular aplicações que fazem uso intensivo de CPU, utilizamos como base as configurações definidas no trabalho realizado por [Makaratzis et al. 2018]. A utilização das mesmas configurações utilizadas por outro artigo provê uma base de comparação para observar se os experimentos aqui realizados estão em conformidade com trabalhos de relevância relacionados ao estudo. Assim, o período de tempo em todas as simulações foi de uma hora em cada execução. As aplicações utilizadas contaram com 9.000.00 Milhões de Instruções. O comprimento de processamento é uma maneira de inferir o tempo necessário para executar um trabalho em um recurso de nuvem (cujo poder de computação é medido em Milhões de Instruções

por Segundo, MIPS). O tamanho de entrada e saída de cada fluxo de trabalho é 5MB e 30MB, respectivamente. Estes valores referem-se ao tamanho dos arquivos de entrada e saída gerados pela aplicação. A forma de alocação de máquinas virtuais (VMs) sobre as máquinas físicas (*hosts*) utilizada foi *First-Fit-Approach*, isso significa que para cada *host* criado, apenas uma VM é alocada nele. Logo o número de VMs é igual ao número de *hosts*. As características dos *hosts* são: CPU com 2,66 GHz e 2 núcleo de processamento, 4 GB de memória RAM, 1000 GB de disco e velocidade de rede de 1 GB/s. As características utilizadas nas VMs são: vCPU de 2,66 GHz e 1 núcleo de processamento, 4 GB de memória RAM, 10GB de disco e velocidade de rede de 100 MB/s. Todos os computadores simulados se comunicam entre si. Para avaliar o consumo de energia foram definidas cinco configurações com diferentes quantidades de recursos, como segue: (i) 50 VMs (*hosts*), (ii) 100 VMs (*hosts*), (iii) 200 VMs (*hosts*),(iv) 500 VMs (*hosts*) e (v) 1000 VMs (*hosts*).

Para a realização dos experimentos, foi utilizado um servidor Dell *PowerEdge* R720 disponível no Laboratório de Alto Desempenho[7] (LAD) da PUCRS, com as seguintes configurações: 2 processadores Intel Xeon E5-2650 de 2,6 Ghz com 8 núcleos físicos e 16 núcleos virtuais para cada processador, totalizando 32 núcleos virtuais e 128 GB de memória RAM. O Sistema Operacional é o Ubuntu Server versão 16.04. Todos os códigos fontes dos testes e arquivos de configurações necessários para a realização deste estudo estão disponíveis em um repositório do GitHub[8].

## 4.2. Consumo Energético

O consumo de energia em cada simulador está apresentado na Figura 1 para cada cenário em kWh. Pode-se observar que o consumo de energia estimado em todas as estruturas de simulação está próximo do mesmo nível para todos os valores de *hosts* e VMs. A mesma tendência foi notada também ao usar diferentes características de *hosts*/VMs e entradas de simulação, ou seja, todas as simulações produziram estimativas semelhantes. Todos os resultados neste estudo estão próximos dos resultados que [Makaratzis et al. 2018] apresentaram, reforçando que todos os experimentos foram conduzidos de maneira correta.

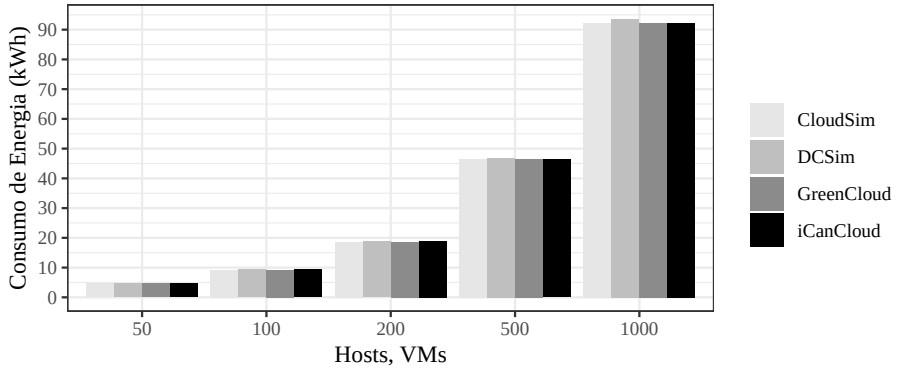



**Figura 1. Estimativas de consumo energético gerado pelos simuladores nas diferentes configurações de recursos.**

[7]www.pucrs.br/ideia/laboratorios/lad/

[8]http://github.com/viniciusmeyer/wscad2018

O simulador CloudSim usa vários modelos de energia para processadores em servidores, mas não considera o impacto de outros componentes no consumo de energia do *datacenter*, assim como o DCSim. Já o GreenCloud, considera o impacto energético da CPU dos servidores, rede, memória e armazenamento, além de calcular o consumo de energia dos *switches* de rede. O iCanCloud considera o impacto energético de vários componentes do *datacenter* e fornece modelos de energia para os processadores dos servidores, memória, armazenamento, rede e fonte de alimentação.

### 4.3. Modelo de energia do SimGrid

Após a realização da comparação de estimativas de consumo de energia com simuladores de nuvem computacional, surgiu a hipótese de comparar o trabalho realizado por [Makaratzis et al. 2018] e os resultados obtidos no nosso estudo com uma importante ferramenta de simulação de ambientes distribuídos, o SimGrid[9].

O projeto do SimGrid foi iniciado em 1999 para permitir o estudo de escalonamento de algoritmos em plataformas heterogêneas [Casanova et al. 2008]. No decorrer dos anos, novas funcionalidades foram adicionadas ao sistema, sendo uma dessas a possibilidade de simular o consumo de energia [Heinrich et al. 2017]. Tal funcionalidade foi validada por seus desenvolvedores na plataforma Grid'5000 [Balouek et al. 2012]. O modelo de energia utilizado no SimGrid é descrito pela fórmula:

$$P_{i,f,w}(u) = P_{i,f}^{static} + P_{i,f,w}^{dynamic} \times u \tag{1}$$

Onde:

- *i* é uma máquina (servidor);
- $f$ é a frequência de operação;
- *w* é o trabalho computacional realizado;
- *u* é o percentual utilizado.

Para a simulação foi utilizado a versão 3.19 compilada em Java do aplicativo, juntamente com os exemplos disponíveis no GitHub[10] do mesmo.

Foram realizadas as simulações para 50, 100, 200, 500 e 1000 *hosts*/VMs, respectivamente. A aplicação forneceu os dados em Joules, sendo os mesmos convertidos para kWh utilizando a fórmula:

$$E_{kWh} = E_J \times 2,777778 \times 10^{-7} \tag{2}$$

A Tabela 1 apresenta os valores aferidos no SimGrid juntamente com os valores obtidos com os demais simuladores testados. As configurações de ambiente e aplicações são as mesmas utilizadas nos demais simuladores. É possível perceber que o modelo de energia fornecido pelo SimGrid, nas simulações aqui realizadas, está muito próximo dos resultados dos demais. Podemos afirmar que todos os simuladores testados neste estudo, com o uso das mesmas configurações, possuem o mesmo comportamento de consumo energético.

[9]http://simgrid.gforge.inria.fr/

[10]https://github.com/simgrid/simgrid/tree/master/examples/java

**Tabela 1. Comparação entre os resultados obtidos juntamente com SimGrid**

| Hosts, VMs | CloudSim | DCSim | GreenCloud | iCanCloud | SimGrid |
|---|---|---|---|---|---|
| **50** | 4,67 | 4,68 | 4,61 | 4,60 | 4,65 |
| **100** | 9,30 | 9,37 | 9,22 | 9,31 | 9,33 |
| **200** | 18,68 | 18,74 | 18,56 | 18,95 | 18,72 |
| **500** | 46,57 | 46,85 | 46,61 | 46,32 | 46,84 |
| **1000** | 92,23 | 93,70 | 92,15 | 92,30 | 93,73 |

Consumo estimado em kWh

### 4.4. Análise de Desempenho

Após a realização dos testes com todos os simuladores percebeu-se que o CloudSim, o DCSim e o SimGrid, em todas as configurações de simulação, executaram em menos de 10 segundos. Entretanto, os tempos de execução de simulação utilizando o GreenCloud e o iCanCloud foram consideravelmente superiores, com tempos de execução maiores que 18 horas. Sendo assim, resolvemos realizar uma análise mais detalhada da utilização dos recursos que cada simulador faz no *hardware* em que está sendo executado. A Tabela 2 apresenta os tempos em segundos de cada simulação, para os mesmos cenários.

**Tabela 2. Tempos de Simulação**

| Hosts, VMs | CloudSim | DCSim | GreenCloud | iCanCloud | SimGrid |
|---|---|---|---|---|---|
| **50** | 0,58 | 0,34 | 1049 | 9 | 0,22 |
| **100** | 0,76 | 0,40 | 1663 | 25 | 0,21 |
| **200** | 1,02 | 0,53 | 3785 | 66 | 0,29 |
| **500** | 2,29 | 0,83 | 17701 | 371 | 0,46 |
| **1000** | 5,55 | 1,10 | 65865 | 1622 | 0,75 |

Tempo de simulação em segundos

Cabe ressaltar que o resultado do consumo de energia não é alterado, uma vez que os parâmetros da simulação não são modificados. O desempenho da simulação em si sofre pequenas alterações no tempo de execução dependendo diretamente do computador em que se está rodando. Estes valores são os resultados de uma média simples de 10 execuções em cada cenário de teste.

#### 4.4.1. Recursos Utilizados pelos Simuladores

Com o objetivo de observar os recursos utilizados pelos simuladores, inserimos no servidor um *script* que tem a função de capturar e armazenar o percentual de uso da CPU e a quantidade de memória que o simulador utiliza. Esses recursos são os mais afetados em todos os casos e por isso foram escolhidos para serem analisados.

A Figura 2 apresenta a utilização da memória e da CPU pelo CloudSim. Foi observado que o CloudSim é implementado em Java e se aproveita da arquitetura *multicore* do *hardware* com o uso de *threads*. Sendo assim, o percentual do uso de CPU é uma média da utilização total da aplicação sobre os 32 núcleos do servidor. A escala de tempo está

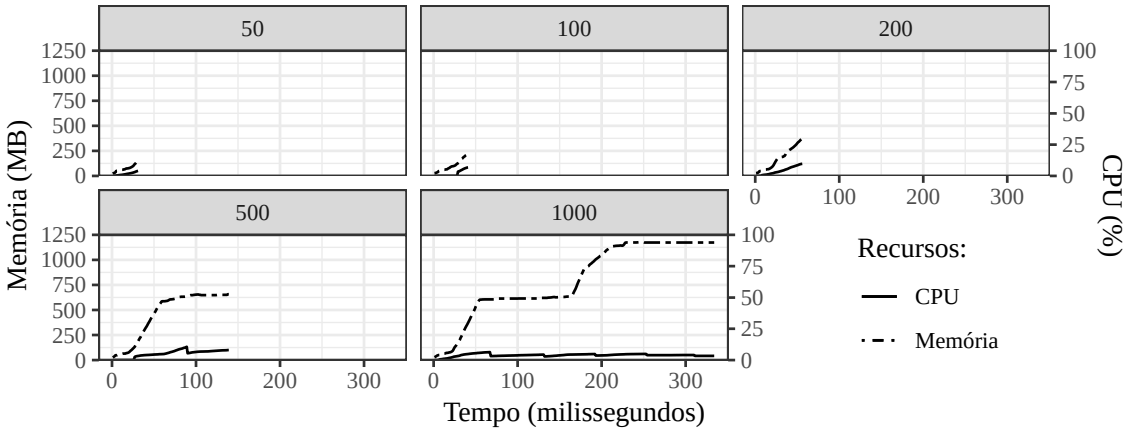



**Figura 2. Utilização de memória *versus* CPU durante as simulações executadas no CloudSim**

representada em milissegundos e a memória em Megabytes. Apesar do número de VMs aumentar em cada execução, o tempo de execução não aumenta de maneira proporcional. Já o uso de memória segue um padrão. O aumento de memória no tempo se dá em função do acréscimo de VMs na simulação.

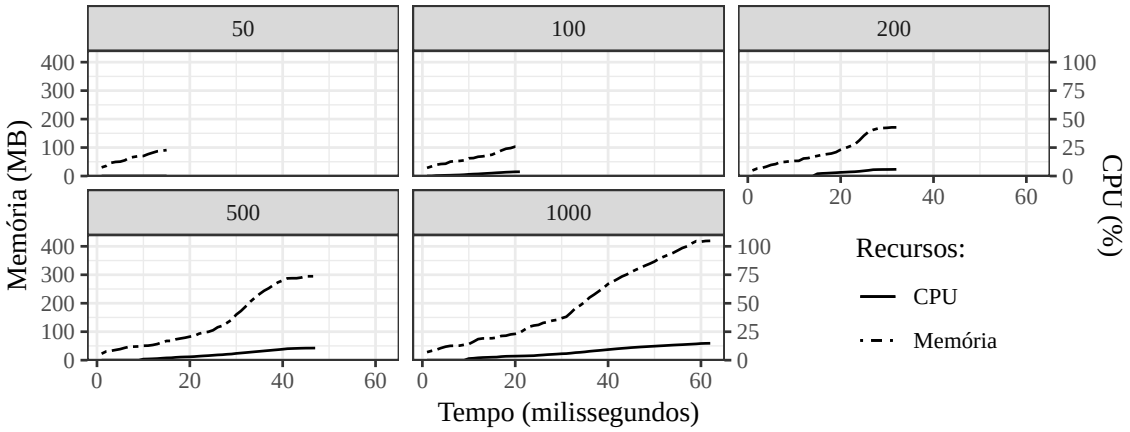



**Figura 3. Utilização de memória *versus* CPU durante as simulações executadas no DCSim**

A Figura 3 mostra os resultados obtidos com a utilização do DCSim. Por se tratar de execuções relativamente curtas, as mesmas escalas para tempo e memória são empregadas. O DCSim também foi construído com Java e faz uso de *threads*, logo para expressar a utilização da CPU, a mesma média sobre todos os processadores foi calculada. Basicamente, a utilização de memória aumenta de acordo com o acréscimo de recursos no ambiente simulado. Este simulador demonstrou o melhor desempenho no tempo de execução e utilização dos recursos.

Com a utilização do GreenCloud o que chamou atenção foi o tempo excessivo de execução. Com 50 VMs a execução durou 17 minutos enquanto que com 1000 VMs subiu para 18 horas e 28 minutos. Isto representa um tempo 64 vezes maior, para uma aumento de 20 vezes de acréscimo de recursos no ambiente de simulação. Observando o uso da

memória RAM na Figura 4, podemos perceber que o crescimento também é proporcional ao aumento de recursos criados dentro do ambiente de simulação. Também podemos afirmar que em todas as execuções ocorre o crescimento deste recurso e em determinado momento, próximo ao fim da execução da simulação, a utilização da memória reduz para aproximadamente 11 MB. Observando os processos que a simulação gera no *hardware* foi constatado que enquanto a memória está próximo de 11 MB, o simulador está em fase de geração de relatórios (ou *traces*) dos resultados. Assim, utiliza uma fração menor e constante de memória RAM quando comparada com o restante da execução.

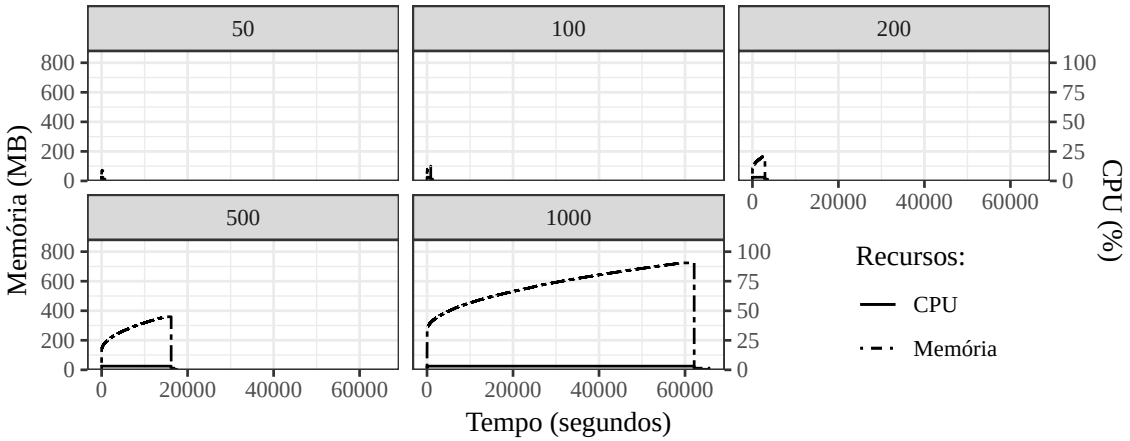



**Figura 4. Utilização de memória *versus* CPU durante as simulações executadas no GreenCloud**

Ainda, a mesma Figura 4 mostra o uso de memória em relação ao uso de CPU no tempo de execução. Outro ponto observado foi que o GreenCloud foi construído sobre NS2 com utilização de linguagem C. Em sua implementação não são exploradas as técnicas de paralelismo de memória compartilhada e apenas um núcleo do servidor é utilizado, e este é o principal motivo do GreenCloud ter um dos maiores tempos de execução. Em todos os casos, é possível observar que enquanto o uso de memória aumenta, o processamento é utilizado em apenas um núcleo, o qual chega em 100%. Durante a geração dos relatório o uso deste núcleo de CPU cai para aproximadamente 15% do seu total. É interessante notar na figura que, para 50 VMs, a maior parte do tempo de execução do simulador se dá para geração de relatórios. Para 100 VMs praticamente 50% do tempo de execução serviu para gerar os resultados da execução. Já para 1000 VMs, o tempo de geração de *traces* é de aproximadamente 4,5% do total da execução, o que resultou em 50 minutos. O que conclui que quanto maior o tempo de execução, o tempo de geração de resultados é proporcionalmente menor.

Em relação ao iCanCloud, notou-se que o mesmo faz uso intenso de memória enquanto está executando. Na execução de 50 VMs, o tempo de execução foi de 9 segundos e a utilização da memória RAM chegou a 294 MB ao seu término. Com 100 VMs, 200 VMs, 500 VMs e 1000 VMs utilizou 494 MB, 850 MB, 2,16 GB e 4,7 GB, respectivamente. A Figura 5 apresenta estes resultados em função do tempo. De 50 para 100 VMs a carga de VMs aumentou 2 vezes enquanto que o uso de memória aumentou 1,6 vezes. De 50 para 200 a carga de VMs aumentou 4 vezes enquanto que a carga de memória aumentou 2,8 vezes. De 50 para 1000 VMs a carga aumentou 20 vezes enquanto que o uso

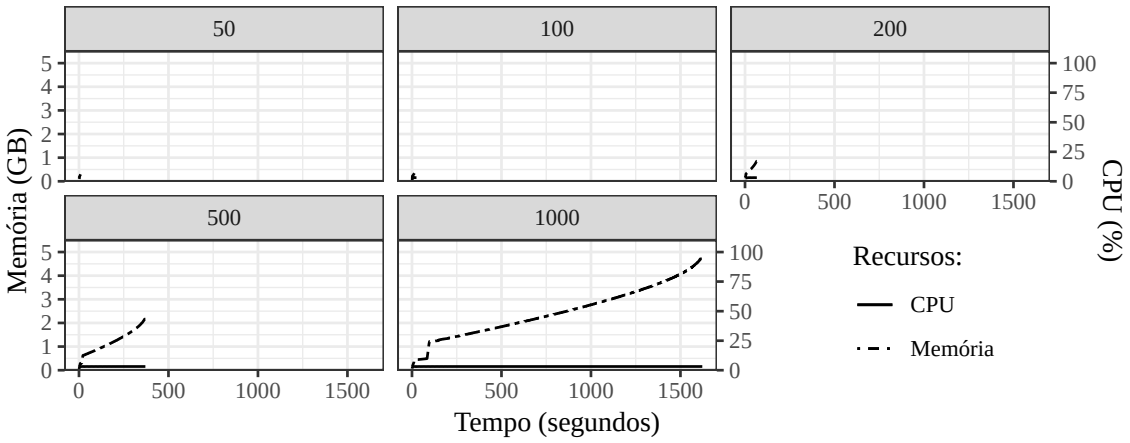



**Figura 5. Utilização de memória *versus* CPU durante as simulações executadas no iCanCloud**

de memória aumentou 16 vezes. Isso também mostra uma certa proporcionalidade entre o número de recursos criados no simulador e a memória do computador que executa esta simulação. Quanto mais recursos o *datacenter* simulado utiliza, mais memória o computador que executa o simulador aloca para a aplicação. Essa é uma avaliação interessante caso haja a necessidade de criar um centro de dados gigante. Pode-se ter uma ideia de quanto de memória RAM irá ser preciso para rodar a simulação. Nesta figura nota-se que enquanto o simulador está em execução apenas um núcleo do CPU está sempre em 100% do seu uso. Isto ocorre pois o comportamento do iCanCloud segue o mesmo do GreenCloud em relação à exploração de técnicas de paralelismo.

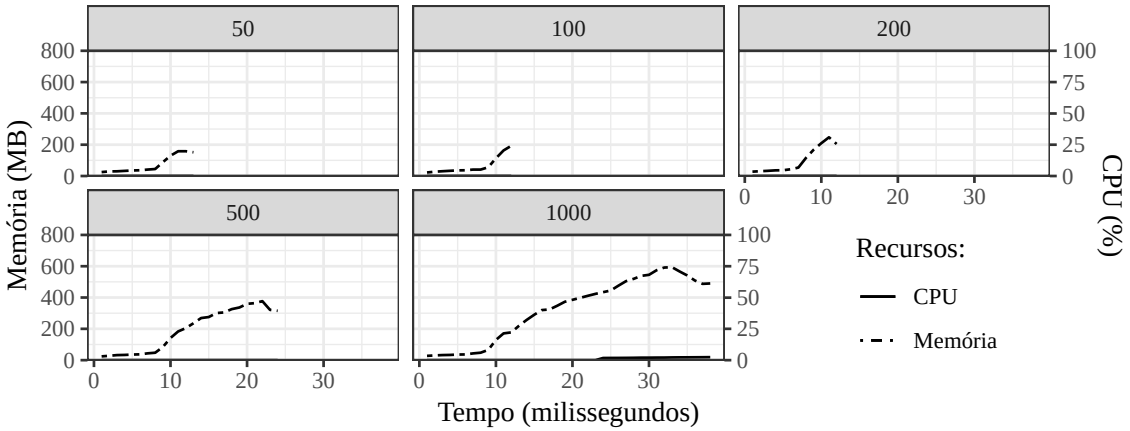



**Figura 6. Utilização de memória *versus* CPU durante as simulações executadas no SimGrid**

A Figura 6 apresenta a utilização de CPU em relação ao uso de memória realizado pelo SimGrid durante o tempo de execução das simulações. Um fato que chamou atenção é que para executar todos os testes, o SimGrid teve o menor consumo de CPU de todos os simuladores. Observando o comportamento da execução, notou-se que ele explora o paralelismo e utiliza diversos núcleos do CPU, de maneira mais eficiente, nos testes aqui realizados. O aumento do uso de memória se dá pelo crescimento de recursos na

simulação. De maneira geral pode-se afirmar que o SimGrid fez menos utilização de memória RAM, quando comparado com os demais simuladores.

## 5. Conclusão e Trabalhos Futuros

Neste estudo, foram examinadas as plataformas de simulação de nuvem de código aberto mais populares, com foco nos modelos e métricas propostos para simular o comportamento das infraestruturas em nuvem. A pesquisa apresenta as estimativas de consumo de energia que diversas configurações de centros de dados utilizam.

A partir da experimentação com os *framework* CloudSim, DCSim, GreenCloud e iCanCloud, pode-se observar que todos os simuladores fornecem a mesma tendência do consumo de energia em servidores à medida que *hosts* e VMs variam, onde as aproximações do consumo total de energia estavam no mesmo nível. Em um segundo momento, foram executadas as mesmas simulações no SimGrid, no qual os valores aferidos seguiram a mesma tendência dos demais simuladores.

Após observar que alguns simuladores resultaram em tempos de execução considerados excessivos (mais de 18 horas), foi analisado em detalhes a utilização dos recursos pelos simuladores sobre o *hardware*. De modo geral, pode-se afirmar que o aumento do uso de memória em todos os testes ocorreu com o aumento dos recursos simulados. Ou seja, quanto mais *hosts* e VMs, mais memória o simulador necessita para ser executado. Também, o CloudSim, DCSim e SimGrid exploram técnicas de paralelismo e atingem desempenhos elevados quando comparados com GreenCloud e iCanCloud, os quais utilizam apenas um núcleo de processamento durante praticamente todo o tempo de execução. Ainda, o SimGrid foi o simulador que obteve melhor desempenho em tempo de execução e uso de CPU. Já no quesito utilização de memória RAM, o simulador com a menor utilização foi o DCSim.

Levando em consideração que o GreenCloud e iCanCloud utilizaram apenas um núcleo de processamento do servidor, como trabalho futuro pretendemos estudar e analisar a possibilidade de paralelizar alguma parte do código fonte, visando melhorar o desempenho. Ainda, percebemos que nenhum simulador testado fornece características como refrigeração e/ou espaço físico em que o *datacenter* se encontra. Então, pretende-se incluir essas características em ambientes simulados e comparar com ambientes físicos para avaliar sua validade.

## Referências

[Alshammari et al. 2018] Alshammari, D., Singer, J., and Storer, T. (2018). Performance evaluation of cloud computing simulation tools. In *2018 IEEE 3rd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pages 522–526.

[Balouek et al. 2012] Balouek, D., Amarie, A. C., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., et al. (2012). Adding virtualization capabilities to the grid'5000 testbed. In *International Conference on Cloud Computing and Services Science*, pages 3–20. Springer.

[Calheiros et al. 2011] Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., and Buyya, R. (2011). Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.*, 41(1):23–50.

[Casanova 2001] Casanova, H. (2001). Simgrid: a toolkit for the simulation of application scheduling. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 430–437.

[Casanova et al. 2008] Casanova, H., Legrand, A., and Quinson, M. (2008). Simgrid: A generic framework for large-scale distributed experiments. In *Tenth International Conference on Computer Modeling and Simulation (uksim 2008)*, pages 126–131.

[Dayarathna et al. 2016] Dayarathna, M., Wen, Y., and Fan, R. (2016). Data center energy consumption modeling: A survey. *IEEE Communications Surveys Tutorials*, 18(1):732–794.

[Guyon et al. 2017] Guyon, D., Orgerie, A. C., Morin, C., and Agarwal, D. (2017). How much energy can green hpc cloud users save? In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 416–420.

[Heinrich et al. 2017] Heinrich, F. C., Cornebize, T., Degomme, A., Legrand, A., Carpen-Amarie, A., Hunold, S., Orgerie, A. C., and Quinson, M. (2017). Predicting the energy-consumption of mpi applications at scale using only a single node. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 92–102.

[Kliazovich et al. 2010] Kliazovich, D., Bouvry, P., Audzevich, Y., and Khan, S. U. (2010). Greencloud: A packet-level simulator of energy-aware cloud computing data centers. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, pages 1–5.

[Li et al. 2017] Li, Z., Tesfatsion, S., Bastani, S., Ali-Eldin, A., Elmroth, E., Kihl, M., and Ranjan, R. (2017). A survey on modeling energy consumption of cloud applications: Deconstruction, state of the art, and trade-off debates. *IEEE Transactions on Sustainable Computing*, 2(3):255–274.

[Makaratzis et al. 2018] Makaratzis, A. T., Giannoutakis, K. M., and Tzovaras, D. (2018). Energy modeling in cloud simulation frameworks. *Future Generation Computer Systems*, 79:715 – 725.

[Nuñez et al. 2012] Nuñez, A., Vázquez-Poletti, J. L., Caminero, A. C., Castañé, G. G., Carretero, J., and Llorente, I. M. (2012). icancloud: A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing*, 10:185–209.

[O'brien et al. 2017] O'brien, K., Pietri, I., Reddy, R., Lastovetsky, A., and Sakellariou, R. (2017). A survey of power and energy predictive models in hpc systems and applications. *ACM Comput. Surv.*, 50(3):37:1–37:38.

[Orellana et al. 2017] Orellana, J., Bonacic, C., Marín, M., and Gil-Costa, V. (2017). Energysim: An energy consumption simulator for web search engine processors. In *Proceedings of the Summer Simulation Multi-Conference*, SummerSim '17, pages 18:1–18:12, San Diego, CA, USA. Society for Computer Simulation International.

[Tighe et al. 2012] Tighe, M., Keller, G., Bauer, M., and Lutfiyya, H. (2012). Dcsim: A data centre simulation tool for evaluating dynamic virtualized resource management. In *2012 8th international conference on network and service management (cnsm) and 2012 workshop on systems virtualization management (svm)*, pages 385–392.

# Power Consumption of Parallel Programming Interfaces in Multicore Architectures: A Case Study. *

**Adriano Marques Garcia**[1], **Claudio Schepke**[1], **Alessandro Gonçalves Girardi**[1], **Sherlon Almeida da Silva**[1]

[1]Laboratório de Estudos Avançados em Computação (LEA)
Universidade Federal do Pampa (UNIPAMPA) – Alegrete – RS – Brazil

adriano1mg@gmail.com,
{claudio.schepke, alessandro.girardi}@unipampa.edu.br,
sherlonalmeidadasilva@gmail.com

***Abstract.*** *This paper evaluates the power consumption of different parallel programming interfaces (PPI) in a multicore architecture. These PPIs are: PThreads, OpenMP, MPI-1 and MPI-2 (spawn). We measure the total energy and execution time of 11 applications in a single architecture, varying the number of threads/processes. The goal is to show that these applications can be used as a parallel benchmark to evaluate the power consumption of different PPIs. The results show that PThreads has the lowest power consumption among the interfaces, consuming less than the sequential version for memory-bound applications.*

## 1. Introduction

Nowadays, many countries are limiting the use of existing supercomputers because of their high energy consumption [Bourzac, Katherine 2017]. This shows that energy consumption is currently a concern in many different computer systems. The increasing of the applications complexity and data size has required extensive research into both computational performance and energy efficiency. The popularization of Green500, which lists computers from the TOP500 list of supercomputers in terms of energy efficiency, shows that reducing energy consumption is one of the directions of high-performance computing [Bourzac, Katherine 2017]. So the challenge should not only be to increase performance, but also to consume less energy.

The performance increase is reached with even faster multiple parallel processors. Parallel computing aims to use multiple processors to execute different parts of the same program simultaneously [Rauber and Rünger 2010]. However, processors should be able to exchange information at a certain point in execution time. While tasks parallelism makes it possible to increase the performance, the need for communication among them and the more extensive use of the processor can lead to an increase in power consumption.

The parallelism can be explored with different Parallel Programming Interfaces (PPIs), each one having specific peculiarities in terms of synchronization and communication. In addition, the performance gain may vary according to processor architecture and hierarchical memory organization, communication model of each PPI, and also with the complexity and other characteristics of the application.

*Supported by CAPES

Although parallelism allows performance gains, this can lead to higher power consumption. This power consumption grows mainly according to the amount of processors that are used in parallel and the volume of communication among them. On the other hand, the reduction in execution time allowed by the parallelization causes the decrease in the total energy consumption in some cases. Parallel benchmarks can be used to define which parallelization strategy compensates for the increase in power consumption in a particular architecture. However, there is not a benchmark that offers a good set of applications, fully parallelized, using multiple PPIs and different models of communication by tasks. The most commonly used parallel benchmarks have only partial parallel sets using more than one PPI.

To fill this gap, this work studies a set of 13 applications developed with the purpose of evaluating the performance and energy consumption in multi-core architectures. These applications were developed and classified according to different criteria in previous studies [Lorenzon et al. 2015b, Lorenzon et al. 2015a, Lorenzon 2014, Garcia 2016]. These studies have shown that these applications have characteristics that are distinct enough to represent different scenarios. The objective of this work is to show the impact of these distinct characteristics on the power consumption of different applications and also the impact of the implementations using different PPIs.

The remainder of this work is organized as follows. In the section 2 we present the PPIs in which the applications are implemented. The related works are discussed in section 3, where we compare our work with similar benchmarks. The section 4 presents the set of applications and the techniques used to parallelize them, bringing more details about the historic of classifications. Section 5 shows how our experiments were structured and brings some information to a better understanding of the results. The section 6 discusses the results and, finally, section 7 draws the final considerations and future works.

## 2. Parallel Programming Interfaces

There are several forms of parallelism that can be applied into a program, such as: data parallelism, shared memory, exchange of messages, and operations in remote memory. These models differ in several aspects, such as whether the available memory is locally shared or geographically distributed, and volume of communication [Gropp et al. 1999]. In this work, the set of applications were implemented using two communication models with the four PPIs: PThreads, OpenMP, MPI-1 and MPI-2.

The OpenMP pattern consists of a series of compiler directives, function libraries, and a set of environment variables that influence the execution of parallel programs [Rauber and Rünger 2010]. These directives are inserted into the sequential code and the parallel code is generated by the compiler from them. This interface operates on the basis of the thread fork-join execution model.

Different from OpenMP, in POSIX Threads (PThreads) the parallelism is explicit through library functions.That is, the programmer is responsible for managing *threads*, workload distribution, and execution control [Butenhof 1997]. PThreads comprises some subroutines that can be classified into four main groups: thread management, mutexes, condition and synchronization variables.

MPI-1 standard API specifies point-to-point and collective communications operations, among other characteristics. In a program developed using MPI-1 all processes are

statically created at the start of the execution. This means that the number of processes remains unchanged during program execution. At the start of the program, an initialization function of the execution environment MPI is executed by each process. This function is `MPI_Init()`. A process MPI is terminated by calling the function `MPI_Finalize()`. Each process is identified by a rank.

Applications deployed with MPI-2 can begin the execution with a single process. Then, the primitive `MPI_Comm_spawn()` can be used for the creation of processes dynamically. A process of an MPI application, which will be called by the parent, invokes this primitive. This invocation causes a new process, called child, to be created, which does not need to be identical to the parent. After creating a child process, it will belong to an intra-communicator and the communication between parent and child will occur through this communicator. In the child process, the execution of the function `MPI_Comm_get_parent()` is responsible for returning the intercom that links it to the parent. In the parent process, the intercom that binds the child is returned in the execution of the function `MPI_Comm_spawn()`.

## 3. Related Work

There are several benchmarks developed to serve different purposes. Through a bibliographic study, we searched for benchmarks that have similar purposes and the same target architectures of the benchmark proposed in this work. Therefore, we have considered benchmarks that provide a set of parallel applications for embedded or general-purpose multi-core architectures. In this way, we identify the following benchmarks: ALPBench, PARSEC, ParMiBench, SPEC, Linpack, NAS and Adept Project.

### 3.1. Similar Benchmarks

ALPBench consists of a set of parallelized complex media applications gathered from various sources and modified to expose thread-level and data-level parallelism. It consists of 5 applications parallelized with PThreads. This benchmark is focused on general-purpose processors and has open source license.

PARSEC (Princeton Application Repository for Shared-Memory Computers) is an open source benchmark suite. It consists of 13 applications, some parallelized using OpenMP, or PThreads or Intel TBB. The suite focuses on emerging workloads and was designed to contain a diverse selection of applications that are representative of next-generation shared-memory programs for chip-multiprocessors.

ParMiBench is an open source benchmark that specifically serves to measure performance on embedded systems that have more than one processor. This benchmark organizes its applications into four categories and domains: industrial control and automotive systems, networks, office devices and security. Its set consists of 7 parallel applications implemented using PThreads.

SPEC is a closed source benchmark, but offers academic licenses. This benchmark is intended for general purpose architectures, but is subdivided into several groups with specific target architectures, and can be used for several purposes, such as: Java servers, file systems, high-performance systems, CPU tests, and others. We consider the following groups of SPEC: SPEC MPI2007 and SPEC OMP201. SPEC MPI2007 is a set of 18 applications deployed in MPI focused on testing high-performance computers.

SPEC OMP2012 uses 14 scientific applications implemented in OpenMP, offering optional energy consumption metrics based on SPEC Power.

HPL consists of a software package that solves arithmetic dual floating-point precision random linear systems in high-performance architectures. It runs a testing and timing program to quantify the accuracy of the solution obtained, as well as the time it took to compute. HPL is open-source and consist of 7 applications that form a collection of subroutines in Fortran, mostly CPU-Bound. Parallel implementations use MPI. HPL is the benchmark that makes up the so-called *High-Performance Computing Benchmark Challenge*, which is a list of the 500 fastest high-performance computers in the world.

The NAS Parallel Benchmarks is a small set of open source programs that serve to evaluate the performance of parallel supercomputers. The benchmark is derived from physical applications of fluid dynamics and consists of four cores and three pseudo-applications. It is an open source benchmark and the applications are implemented with MPI and OpenMP. Some applications are also implemented in HPF, UPC, Java, Titanium, TBB etc.

The Adept Benchmark is used to measure the performance and energy consumption of parallel architectures. Its code is open and is divided into 4 sets: Nano, Micro, Kernel and Application. The Micro suite, for example, consists of 12 sequential and parallel applications with OpenMP, focusing on specific aspects of the system, such as process management, caching, and others. On the other hand, the Kernel set has 10 applications implemented sequentially and parallel with OpenMP, MPI and one of them in UPC (Unified Parallel C).

### 3.2. Comparison of Benchmarks

The benchmark addressed in this work consist of 11 applications implemented in C and their complexities range from $O(n)$ to $O(n^3)$. All applications are parallelized in 4 PPIs: PThreads, OpenMP, MPI-1 and MPI-2. These PPIs are the target of this work because they are the most widespread in the academic field and also because they are supported by most multi-core architectures, both embedded and general purpose. Therefore, the purpose of this benchmark is to provide the user with a tool to evaluate the performance and energy consumption of different PPIs in multi-core architectures.

We analyze the main characteristics of the related parallel benchmarks and compare to the benchmark we propose in this work in Table 1. In relation to the benchmarks, some use only one PPI while others use more than one. However, some of those who use more than one PPI do not have the whole set of applications paralleled by all PPIs. They implement parts of the set with one PPI and other parts with another PPI.

SPEC, for example, has 32 parallel applications divided into two different sets. One of them using OpenMP and the other one using MPI, so it has no set implemented with two or more PPIs. NAS, on the other hand, uses several other PPIs. However, most of application are not implemented in all PPIs. Many of them are not supported by any multicore architecture (ARM Architecture). Three of the benchmarks use PThreads, five of them use OpenMP, and four use MPI. ALPBench also uses Intel TBB and Adept uses UPC.

Thus, even if some of these benchmarks implement three different PPIs, none of

**Table 1. Comparison of our benchmark with the similar ones**

| Rating criteria | ALPBench | PARSEC | ParMiBench | SPEC | HPL | NAS | Adept | Our benchmark |
|---|---|---|---|---|---|---|---|---|
| Number of applications | **5** | **13** | **7** | **32** | **7** | **7** | **10-12** | **11** |
| Number of PPIs | **1** | **3** | **1** | **2** | **1** | **2** | **3** | **4** |
| Number of communication models | **1** | **1** | **2** | **1** | **1** | **2** | **2** | **2** |
| Set of applications implemented in multiple PPIs | | | | | | **X** | | **X** |
| Open-source | **X** | **X** | **X** | | **X** | **X** | **X** | **X** |

them allow an efficient comparison of these PPIs and different communication models (message passing or shared memory). Also they do not exploit the parallelism with dynamic process creation that MPI-2 offers. In this way, we do not find any other benchmark that uses different PPIs, different communication models and a completely parallelized set of applications. The exception is the NAS, but it only offers two PPIs. Therefore, none of them meets the objective of comparing parallel programming interfaces, which is the main objective of the benchmark we are proposing in this work.

## 4. Benchmark Applications

This section presents the 11 applications of the benchmark. They were developed with the purpose of establishing a relation between performance and energy consumption in embedded systems and general purpose architectures [Lorenzon et al. 2014]. Late, the proposal to use this set of applications to form a parallel benchmark was given by [Garcia and Schepke 2018]. All the applications used in this work have detailed descriptions in [Garcia 2016], where the authors provide the algorithms, a detailed description of each application and also the mathematical equations that were used in some implementations. Below is a list of these applications with a brief description of each.

- **Gram-Schmidt**- The Gram-Schmidt process is a method for orthonormalising a set of vectors in an inner product space.
- **Matrix Multiplication** - This algorithm multiplies the lines of a matrix $A$ by the columns of a matrix $B$.
- **Dot Product** - The dot product is an algebraic operation that multiplies two equal-length sequences of numbers.
- **Odd-Even Sort** - It is a comparison sort algorithm related to bubble sort.
- **Dijkstra** - It finds a minimal cost path between nodes in a graph with non-negative edges.
- **Discrete Fourier Transform** - The discrete Fourier transform (DFT) converts a finite sequence of equally-spaced samples of a function into an equivalent-length sequence of equally-spaced samples of the discrete-time Fourier transform (DTFT), which is a complex-valued function of frequency.

- **Jacobi Method** - The Jacobi method is an algorithm for determining the solutions of a diagonally dominant system of linear equations.
- **Harmonic Sums** - The Harmonic Sums or Harmonic Series is a finite series that calculates the sum of arbitrary precision after the decimal point.
- **PI Calculation** - It applies the Gregory-Leibniz formula to find $\pi$.
- **Numerical Integration** - This algorithm integrates an $f(x)$ function in a given interval, using approximation techniques to define an area.
- **Turing ring** - It is a space system in which predators and prey interact in the same place. The system simulates the iteration and evolution of preys and predators through the use of differential equations.

These algorithms are used in the most diverse computing areas. Four of them are directly related to linear algebra. However, some other areas are also represented, some of them are: molecular dynamics, electromagnetism, digital signal processing, image processing, mathematical optimization, among others.

### 4.1. Parallelizing the Applications

Parallelize a sequential program can be done in several ways. However, inappropriate techniques can negatively impact the performance of an application. To minimize this problem, all parallel implementations in this work were based on statements from [Foster 1995, Butenhof 1997, Gropp et al. 1999, Rauber and Rünger 2010]. [Rauber and Rünger 2010] propose that the parallelization must be done in a systematic way, according to them, there are three fundamental steps for the parallelization of a sequential application, which are: computation decomposition; assigning tasks to processes/threads; mapping processes/threads into physical processing units.

The decomposition of the computation and assignment of tasks to processes/threads occurred explicitly in the parallelization with PThreads and MPI 1 and 2 in order to obtain the best workload balancing. Also message exchange functions among processes were included, as well as the dynamic creation of processes in MPI-2. For Parallelization with OpenMP, parallel loops with thin and coarse granularity were used. According to [Foster 1995, Rauber and Rünger 2010], this technique is most appropriate for parallelizing applications that perform iterative calculations and traverse contiguous data structures (e.g. matrix, vector, etc.). For each data structure, a specific parallelization model was adopted.

## 5. Methodology

The results presented in section 6 are the average of 30 executions disregarding the extreme values. This number of executions was established as indicated in [Hunold and Carpen-Amarie 2016]. In this study, the authors perform experiments that show that the minimum number of executions in order to obtain statistically acceptable results in MPI is 30. Following the indications of this study, the results in MPI-1 and MPI-2 showed a standard deviation below 0.5 in the worst cases. OpenMP and PThreads showed a standard deviation below 0.1 in all cases. During the experiments, the computer remained locked to ensure that other applications did not interfere in the results.

The toolkit Intel® Performance Counter Monitor (PCM) 2.0 was used to measure energy consumption. It has a tool to monitor the power states of the processor and DRAM

**Table 2. Details about the applications**

| Data Structures | Problem Size | Acronym | Application | Complexity |
|---|---|---|---|---|
| Unstructured data | 1 billion | NI | Numerical Integration | $O(n)$ |
| | 4 billion | PI | PI Calculation | |
| | 15 billion | DP | Dot Product | |
| Vector | 100000 | HA | Harmonic Sums | $O(n * d)$ |
| | 150000 | OE | Odd-Even Sort | $O(n^2)$ |
| | 32768 | DFT | Discrete Fourier Transf. | |
| Matrix | 2048×2048 | TR | Turing Ring | $O(m * n)$ |
| | | DJ | Dijkstra | $O(n^3)$ |
| | | JA | Jacobi Method | |
| | | MM | Matrix Multiplication | |
| | | GS | Gram-Schmidt | |

memory. For the runtime, the time at the beginning and at the end of the main function of each application was measured and the difference of these values was used. We used these data to calculate power consumption, according to Equation 1. Where $W$ is the power in Watts, $J$ is the total energy in Joules and $s$ is the execution time in seconds.

$$W = \frac{J}{s} \tag{1}$$

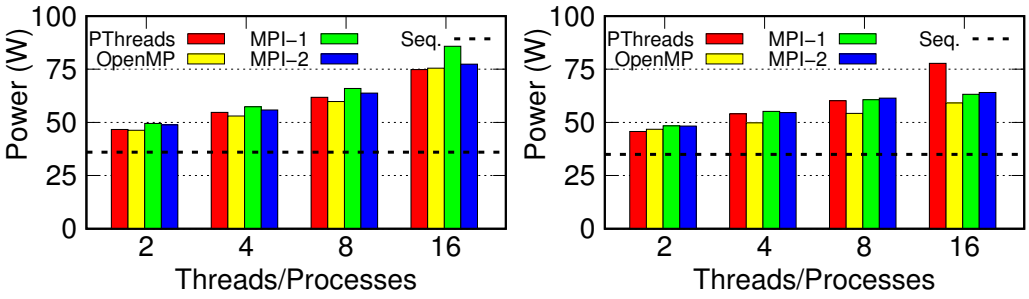
The Table 2 shows the size of the inputs used for each application, the acronym used to identify each application in the following section, and their complexities.

## 6. Results

Next experiments were carried out on a computer equipped with 2 Intel® Xeon® E5-2650 v3 processor. Each processor has 10 physical cores and 10 virtual cores operating at the standard 2.3 GHz frequency and a turbo frequency of 3 GHz. Its memory system consists of three levels of cache: a 32 KB cache L1 and a 256 KB cache L2 for each core. Level L3 has a 25 MB cache for each processor using Smart Cache technology. The main memory (RAM) is 128 GB in size and DDR3 technology. The operating system is Linux version 4.4.0-128 using Intel® ICC 18.0.1 compiler with default optimization flags.

In Figure 1 and Figure 2 bars show the power consumption in watts for each PPI. Each chart displays the results by applications individually. These results refer to running using 2, 4, 8, and 16 parallel threads/processes for each case. In addition, a dashed horizontal line represents the sequential result of the respective application. In Figure 1 are presented the results for the CPU-bound applications. These results show that the power consumption of MPI-1 and MPI-2 is slightly higher when using 2 and 4 parallel tasks. However, for 8 and 16 MPI-1 and MPI-2 processes, they have a power consumption equal to or lower than PPIs that use threads.

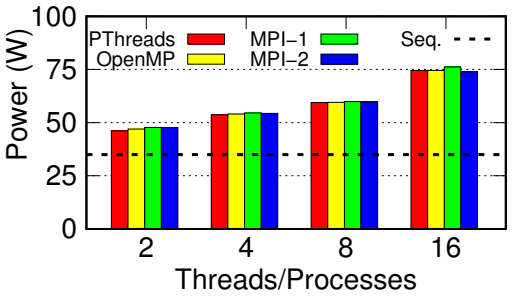
The DFT application shows a different PThreads behavior in relation to the other PPIs when the number of parallel tasks increases. With 2 threads PThreads follows the



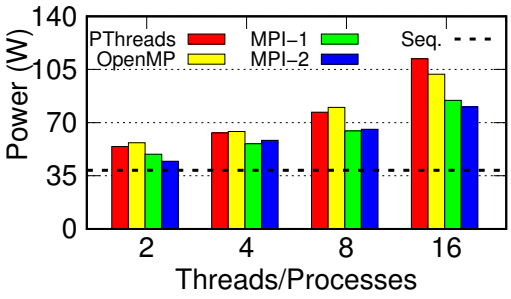

**(a)** Dijkstra - DJ



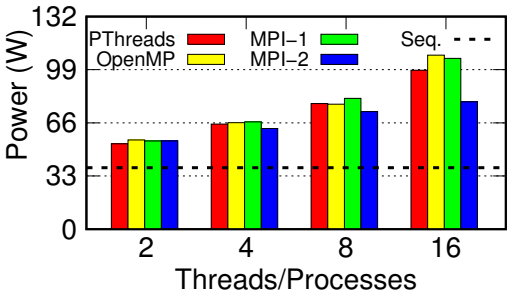

**(b)** Disc. Fourier Transf. - DFT



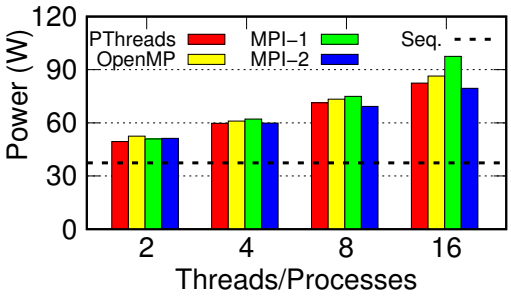

**(c)** Harmonic Sums - HA



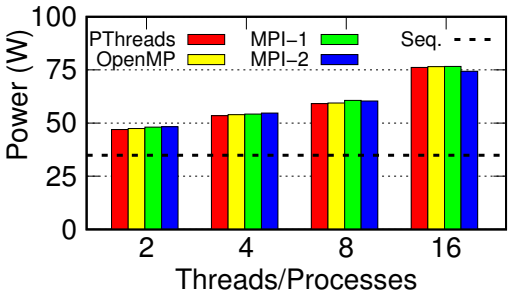

**(d)** Matrix Multiplication - MM




**(e)** Numeric Integration - NI




**(f)** Dot Product - DP




**(g)** Pi Calculation - PI

**Figure 1. Power consumption for CPU-bound applications.**

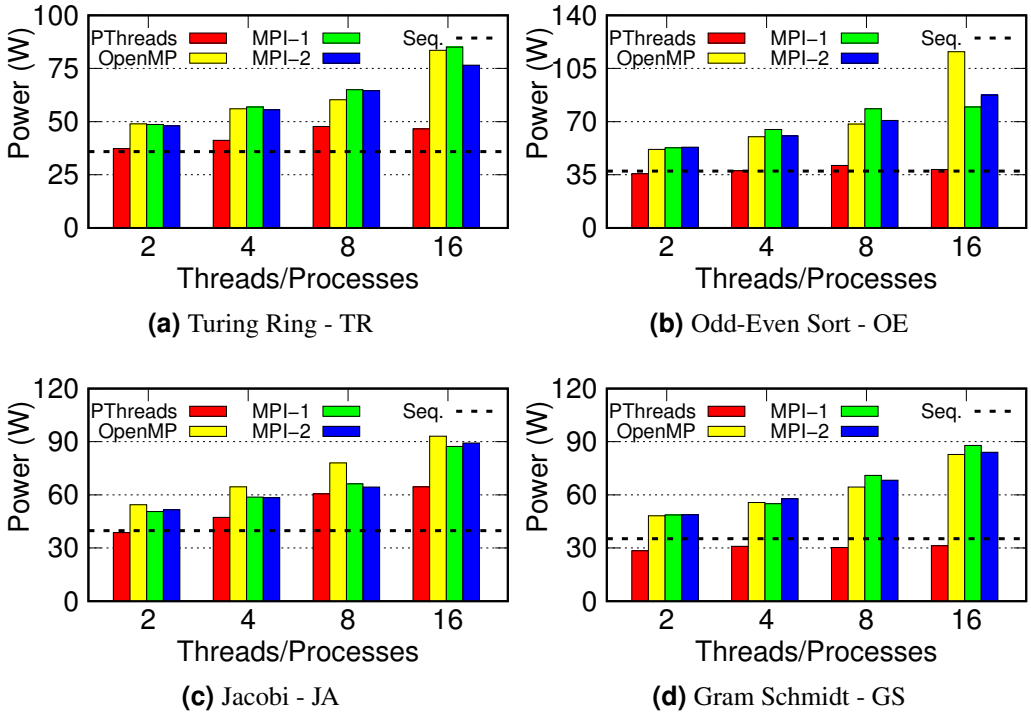
pattern that is seen in most CPU-Bound applications. However, using more threads, this consumption exceeds the consumption of other PPIs incrementally. The other three PPIs follow the pattern by increasing the number of threads, where OpenMP consumes less power than MPI-1 and MPI-2, have similar results for this application.

Comparing MM with DFT shows similar behavior of PThreads in both applications. The consumption of this PPI is lower than OpenMP with 2 threads but grows at a higher rate than the other PPIs as the number of threads increases. The difference in this application is that OpenMP also uses more power than both MPI PPIs, but the rate of increase is not as high as PThreads, which consumes twice as much power with 16 threads as when running with 2 threads. In addition, using MPI-2 with 2 processes this application was the one that most approached the consumption of the sequential version among the CPU-Bound applications.

The low power consumption by PThreads in memory-bound applications does not represent that the total power consumed was lower in this PPI. As seen by [Garcia et al. 2018] the runtime and energy consumption is higher than OpenMP. This low power consumption meant that the application consumed less energy over time, but that time was higher than OpenMP. This means that PThreads has a lower overhead caused by parallelization over OpenMP. In fact, for all memory-bound applications OpenMP uses about 2 and 3 times more memory than PThreads with 8 and 16 threads respectively. On the other hand, PThreads have approximately 10 times more cache misses than other PPIs in memory-bound applications. In this way the execution of PThreads takes more time but the use of hardware in this period is less intense in relation to the other PPIs, which implies in a lower consumption of energy over time. This increase in runtime can be caused by busy waiting for PThreads.

In the memory-bound applications (Figure 2), OE with OpenMP using 16 threads reached the higher power consumption for these cases. What we have concluded, is that the average workload initially set, is not large enough for all cases. Besides that, as seen by [Garcia et al. 2018] OpenMP achieves good performance but the energy consumption keeps the same when scaling to 16 threads, and this leads to an increase in the power consumption. OE is a memory-bound application, so the overhead of communication/synchronization among threads begins to impact negatively earlier in these cases. With MPI-1 and MPI-2 the results obtained are very similar to the results of CPU-bound applications. The growth of the power consumption as the increase in the number of parallel processes follows the same pattern previously observed. What is perceived is that MPI-2 has a lower consumption than MPI-1 in most cases for both CPU and memory-bound. This small difference may possibly be caused by dynamic process creation. This causes processes to be created later in MPI-2.

Our initial hypothesis was that a higher use of the processor and memory system should cause an increase in energy consumption in proportion to the number of parallel threads/processes. But in addition, reducing the execution time of each application should reduce its consumption in proportion to the performance achieved over the sequential application. But what happens in practice is that the energy consumption does not decrease in the same proportion as the execution time, as investigated by [Garcia et al. 2018]. In this way, this generates an increase in energy consumption. This is because there are other factors that impact on energy consumption, such as the need for communication among




**(a)** Turing Ring - TR

**(b)** Odd-Even Sort - OE

**(c)** Jacobi - JA

**(d)** Gram Schmidt - GS

**Figure 2. Power consumption for memory-bound applications.**

tasks and the increase in complexity of control structures that the OS has to deal with.

Another observed factor is that PThreads access less the memory system during synchronization. This means that for memory-bound programs parallelized using PThreads, this more robust processor we used is a good choice, since it provides considerable performance improvements at the same price in the energy consumption. For CPU-bound programs, the power consumption for each PPI are very similar. As the applications uses more CPU, the impact of particular characteristics of each communication model on the memory system is reduced.

The difference in these PPIs can be explained in the context of threads and processes. Threads are often referred as a lighter type of process for the system, while processes are heavier. A thread shares with other threads its code area, data, and operating system resources. Because of this sharing, the operating system needs to deal with less scheduling costs and thread creation, when compared to context switching by processes. All of these factors impact on performance and consequently on power consumption.

## 7. Conclusions and Future Work

In this paper, we present a set of applications that can be used as a benchmark. The main purpose of this benchmark is to analyze power consumption and performance of different parallel programming interfaces in a multi-core architecture. We first compared our studied benchmark with the main parallel benchmarks that are currently used for the same purpose. This comparison showed that there is no benchmark that meets one of our goals: to offer a simpler way to compare PPIs. In addition, we did a study of the history of the applications, where we showed that there were already authors using them for the

same purpose. This fact meant that there was no other benchmark that would effectively meet this demand. So it was necessary to create one from scratch.

Our experimental results showed that the power consumption have increasing rate proportional to the number of threads/processes used in parallel. An important thing to note is that the way each application uses memory causes a high impact on power consumption. These memory access features of each application and PPI should be far better investigated to trace a relationship between the increase in the number of parallel tasks and energy consumption in an upcoming work. With this more detailed analysis we will be able to observe possible problems and points of improvement in the codes.

In future works, we intend to verify how the distribution of threads/processes to different cores and processors affects our experiments. We should also repeat the experiments using another compiler, such as gcc without optimization flags. Next step is to check the scalability of our applications, so we will increase the number of threads/processes by varying the size of the workload and execute in a distributed system. In addition, we have two new applications (Histogram Similarity and Game of Life) that are already implemented and we are making final adjustments to include them in the benchmark. We also consider including more PPIs such as Intel TBB, UPC or Intel Cilk. Finally, to make the benchmark available, we will check the percentage of floating point operations, integer, loads, store, etc. In this way we have data defining each application independently of the architecture used.

## References

Bourzac, Katherine (2017). *Supercomputing poised for a massive speed boost*. Springer Nature International Journal of Science.

Butenhof, D. R. (1997). *Programming with POSIX threads*. Addison-Wesley Professional.

Foster, I. (1995). *Designing and building parallel programs*. Addison Wesley Publishing Company.

Garcia, A. M. (2016). Classificação de um benchmark paralelo para arquiteturas multinúcleo.

Garcia, A. M. and Schepke, C. (2018). Uma proposta de benchmark paralelo para arquiteturas multicore. *XVIII Escola Regional de Alto Desempenho*, pages 285–289.

Garcia, A. M., Schepke, C., Girardi, A. G., and Silva, S. A. (2018). A new parallel benchmark for performance evaluation and energy consumption. *13th International Meeting on High Performance Computing for Computational Science (VECPAR)*.

Gropp, W., Lusk, E., and Thakur, R. (1999). *Using MPI-2: Advanced features of the message-passing interface*. MIT press.

Hunold, S. and Carpen-Amarie, A. (2016). Reproducible mpi benchmarking is still not as easy as you think. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3617–3630.

Lorenzon, A. F. (2014). Avaliação do desempenho e consumo energético de diferentes interfaces de programação paralela em sistemas embarcados e de propósito geral. Master's thesis, Universidade Federal do Rio Grande do Sul.

Lorenzon, A. F., Cera, M. C., and Beck, A. C. S. (2014). Performance and Energy Evaluation of Different Multi-Threading Interfaces in Embedded and General Purpose Systems. *Journal of Signal Processing Systems*, 80(3):295–307.

Lorenzon, A. F., Cera, M. C., and Beck, A. C. S. (2015a). Optimized use of parallel programming interfaces in multithreaded embedded architectures. In *VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on*, pages 410–415. IEEE.

Lorenzon, A. F., Sartor, A. L., Cera, M. C., and Beck, A. C. S. (2015b). The Influence of Parallel Programming Interfaces on Multicore Embedded Systems. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 2, pages 617–625. IEEE.

Rauber, T. and Rünger, G. (2010). *Parallel programming: For multicore and cluster systems*. Springer Science & Business Media.

# Arquitetura Samsara: Explorando Ciência de Situação no Gerenciamento de Nuvens Computacionais

**Vilnei Neves[1], Mauricio Pilla[1], Adenauer Yamin[1], Laércio Pilla[2]**

[1]Programa de Pós-Graduação em Computação (PPGC)
Universidade Federal de Pelotas (UFPel)
96010-610 – Pelotas – RS – Brazil

{vilnei.neves,pilla,adenauer}@inf.ufpel.edu.br

[2]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
Grenoble, France

laercio.lima@inria.fr

***Resumo.*** *As questões relacionadas ao consumo de energia e de sua eficiência nos ambientes computacionais em larga escala têm emergido como principais pontos no desenvolvimento de sistemas computacionais modernos. Este artigo apresenta a concepção da arquitetura Samsara, a qual tem como objetivo o gerenciamento de nuvens computacionais com eficiência energética. A arquitetura foi desenvolvida explorando estratégias de Ciência de Situação, operando de forma autonômica e com intervenção humana mínima, importantes aspectos em grandes centros de processamento de dados. Samsara foi implementada como um dos módulos da plataforma de nuvem OpenStack e considera a capacidade máxima de alocação de cada máquina física, realizando nas mesmas a consolidação das cargas de trabalho. Nas avaliações realizadas com cargas sintéticas foram atingidas reduções de até 12,3% no consumo de energia na infraestrutura computacional gerenciada, demonstrando o potencial de Samsara para a operação de nuvens computacionais.*

## 1. Introdução

A Computação em Nuvem tem conquistado um significativo espaço entre as tecnologias que dão suporte ao atual cenário da computação distribuída e paralela. Entretanto, o consumo de energia dos recursos computacionais subutilizados, principalmente em ambientes de computação em larga escala, pode representar um desperdício substancial de energia elétrica. Soma-se a isso uma crescente preocupação em promover redução nas emissões de dióxido de carbono, o que faz com que a busca por eficiência energética seja considerada essencial para garantir que o crescimento da Computação em Nuvem ocorra de maneira sustentável [Mastelic et al. 2015].

Contudo, a Computação em Nuvem é exposta a cargas de trabalho heterogêneas que refletem a diversidade de aplicações suportadas por ela e que aumentam a complexidade de suas estruturas, tornando a busca do uso mais eficiente dos seus recursos e da energia consumida em um desafio de pesquisa, principalmente diante do cenário exposto acima.

Considerando esse cenário, o presente artigo apresenta a concepção de uma arquitetura, denominada *Samsara*, que tem como objetivo o **gerenciamento de nuvens computacionais com foco na otimização do uso dos recursos para a eficiência energética**. Essa arquitetura foi desenvolvida empregando estratégias baseadas em Ciência da Situação, operando de forma autonômica e com intervenção humana mínima – importantes aspectos considerando o tamanho típico dos centros de processamento de dados de nuvens computacionais. A otimização buscada explora a capacidade máxima de alocação de cada máquina física, realizando a consolidação das cargas de trabalho. A arquitetura *Samsara* foi avaliada experimentalmente e os resultados obtidos demonstraram que a mesma atingiu uma redução de consumo de energia ao redor de $12,31\%$ para as cargas de trabalho avaliadas. Este artigo está organizado em sete seções. Nas Seções 2 e 3 são discutidos os aspectos de eficiência energética em ambiente computacionais e os de Ciência de Situação. A Seção 4 apresenta a arquitetura *Samsara* e suas funcionalidades. A Seção 5 realiza uma discussão dos resultados obtidos na avaliação da *Samsara*. A Seção 6 aborda os trabalhos relacionados. Por fim, as considerações finais são apresentadas na Seção 7.

## 2. Eficiência Energética em Ambientes Computacionais

A atenção dada às questões relacionadas a eficiência energética tem aumentado junto com o crescimento da indústria de computação. Como consequência, tem-se notado também alta demanda de energia com esse crescimento. Para que sirva como ponto de partida, o consumo de energia em computação de 2005 a 2010 cresceu 56%, representando de 1,1% a 1,5% do total da energia consumida globalmente e 2% das emissões CO2, com previsão de aumento para os anos subsequentes [Mastelic et al. 2015].

As recentes iniciativas de desenvolvimento de *hardware*, que vêm direcionando seus esforços para a melhoria das tecnologias que lidem melhor com o consumo de energia de componentes e dispositivos, conseguiram aumentar a eficiência energética até um certo grau. As principais abordagens para aumentar a eficiência energética dos dispositivos computacionais buscam fazer que o consumo de energia acompanhe a carga das aplicações. Exemplo disso é o uso da técnica de controle de voltagem e frequência (*Dynamic Voltage and Frequency Scaling* - DVFS), a qual permite reduzir o consumo dos processadores ao cubo [Beloglazov et al. 2011].

Ademais, as iniciativas da indústria com o objetivo de desenvolver técnicas e métodos padronizados para a redução do consumo de energia, especialmente o consumo associado às grandes estruturas computacionais, ganharam força e fomentam cada vez mais a ideia da *Computação Verde* ou *Computação Sustentável*. Iniciativas como *The Green Grid*[1] têm direcionado sua atenção para questão da eficiência energética e trabalham para melhorar a eficiência dos recursos da tecnologia da informação e centros de dados em todo o mundo.

Em ambientes em larga escala, as questões relacionadas ao consumo de energia estão vinculadas mais fortemente com a forma com que os recursos disponíveis são utilizados. Na expectativa de que esses ambientes tenham que lidar com situações de pior caso ao fornecer os serviços de computação, tende-se a superdimensionar os recursos dos mesmos. Como consequência dessa estratégia, em momentos de menor demanda, os recursos

[1] http://www.thegreengrid.org

disponíveis nesses ambientes podem não atingir uma faixa ideal de eficiência, acarretando como efeito colateral um desperdício da energia consumida [Beloglazov et al. 2011].

Nos ambientes em nuvem, o uso da *virtualização* permite que possamos usar a *consolidação de cargas de trabalho* como uma abordagem capaz de promover uma maior eficiência energética ao migrar as máquinas virtuais em execução para um conjunto menor de servidores, explorando a estratégia de utilizar ao máximo a capacidade de alocação de cada máquina física, assim amortizando os custos de energia ociosa de forma mais eficiente [Varasteh and Goudarzi 2017].

## 3. Ciência de Situação

O desenvolvimento de tecnologias móveis e seu crescente uso impõem que as aplicações ganhem características que as permitam lidar com a complexidade decorrente desse processo, fazendo que as mesmas possuam dinamismo para efetuar adaptações em seu comportamento de acordo com as mudanças que ocorram ao seu redor. Essa capacidade de entender o ambiente ao redor, suas mudanças e a capacidade de adaptar-se de acordo, estão relacionadas com as pesquisas relativas à *Ciência de Situação (Situation Awareness)* [Onwubiko 2012].

As informações provenientes de sensores, de forma geral, são dados brutos com o mínimo de processamento. Uma forma de lidar com a limitação de trabalhar com dados brutos é derivar essas informações de um contexto de baixo nível para uma camada de mais alto nível. A noção de *situação* pode ser usada como um conceito de alto nível para a representação de um estado.

Uma *situação* consiste da interpretação de um conjunto de elementos contextuais instanciados, provenientes de sensores, relacionando cada um de forma a prover alguma informação válida em um intervalo de tempo específico [Liu et al. 2017]. É um conceito subjetivo que depende de um conjunto de elementos para ganhar significado. Por exemplo, pode depender dos sensores disponíveis no sistema para determinar quais são os contextos que podem ser usados em uma especificação.

Uma *situação* se distingue de outras informações por incluir aspectos estruturais de temporalidade – refletindo o instante de um evento, duração, frequência e sequência – e de outras naturezas que podem ser um simples estado abstrato de uma certa entidade (e.g. servidor desligado). Ela depende também do ambiente onde o sistema trabalha, que determina o domínio de conhecimento que pode ser aplicado e das necessidades requeridas pelas aplicações, determinando quais estados são interessantes. Portanto, informações fornecidas por um mesmo sensor podem ser interpretadas para diferentes situações de acordo com a necessidade das aplicações.

Além disso, uma *situação* pode ser também ser composta de outras situações abstraídas de dados de baixo nível e reusadas em diferentes ambientes e aplicações, de forma que esse relacionamento obtido entre situações permita aumentar a abstração desses estados para diminuir sua complexidade.

A grande vantagem de usar situações está na habilidade de proporcionar uma representação compreensível aos seres humanos dos dados capturados por sensores para as aplicações, ao passo que esse uso abstrai delas a complexidade da leitura dos dados, ruídos nesses dados e das atividades de inferência.

## 4. Samsara: Arquitetura Proposta e Funcionalidades

Na modelagem proposta para uso na arquitetura *Samsara*, o ambiente em nuvem é representado por três entidades: *Célula*, *Nodo Físico* e *Instância*. A *Célula* representa o ambiente em nuvem gerenciado pela arquitetura *Samsara*. Dentro da *Célula* existe um conjunto de *Nodos Físicos*, que representam os servidores ou *hosts* responsáveis por hospedar e executar as *Instâncias*. Cada *Instância*, por sua vez, representa uma máquina virtual em execução em *Nodo Físico*.

A Figura 1 apresenta a arquitetura *Samsara*, organizada em três subsistemas: (i) *Subsistema de Gerenciamento de Contextos*, responsável pela coleta e armazenamento das informações contextuais; (ii) *Subsistema de Identificação de Situações*, responsável pelo processamento e análise das informações contextuais, realizando ajustes e conversões das informações obtidas e armazenadas (*processamento*) e, a partir delas, realizar a identificação de situações e a tomada de decisão a partir das mesmas (*análise*); e por último, o (iii) *Subsistema de Adaptação*, responsável por executar ações que realizem mudanças no ambiente gerenciado.

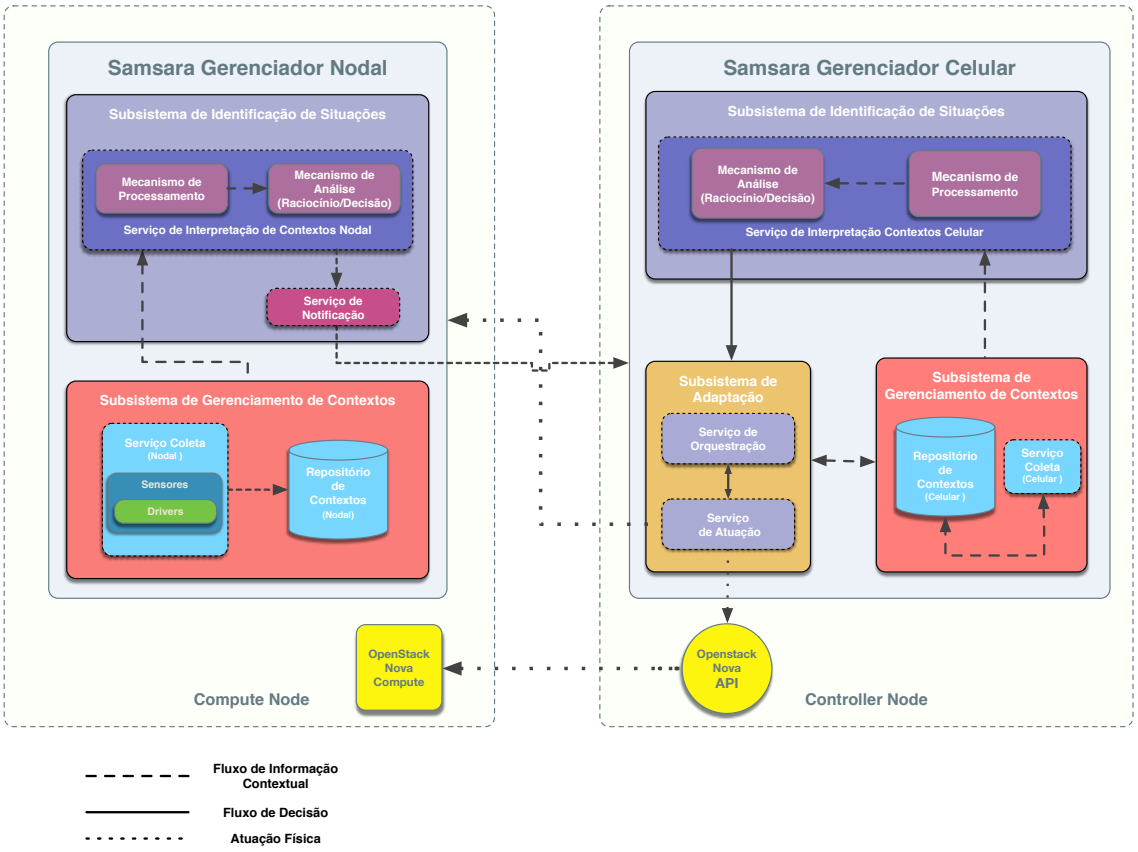



**Figura 1. Organização da Arquitetura Samsara**

Cada subsistema possui serviços que são executados sobre duas instâncias da arquitetura: (a) os *Gerenciadores Nodais*, instalados nos *Compute Nodes*[2], são responsáveis por interagirem diretamente com o nodo físico, coletando e armazenando suas informações por meio de um conjunto de sensores, e por realizarem uma análise local das informações contextuais; e (b) o *Gerenciador Celular*, instalado no *Controller Node*[2],

é responsável por gerenciar o ambiente em nuvem (*Célula*), utilizando as informações de contexto situacional provenientes dos *Gerenciadores Nodais*, para analisar e inferir situações nas quais a célula se encontra e disparar ações que modifiquem essas situações quando necessário, de acordo com as regras definidas.

A arquitetura *Samsara* implementa funcionalidades de três naturezas:

- **Coleta das Informações Contextuais** - No escopo do nodo físico, são coletadas informações de capacidade dos recursos, sua identificação dentro do ambiente da nuvem, além das informações contextuais referentes a sua utilização.
  Essas informações são coletadas pelo *Serviço de Coleta*, que periodicamente consulta os sensores para produzir informações contextuais e as armazena no Repositório de Contextos Nodal. Estas informações são utilizadas pelo *Serviço de Interpretação Nodal* e enviadas ao Gerenciador Celular (Figura 2).

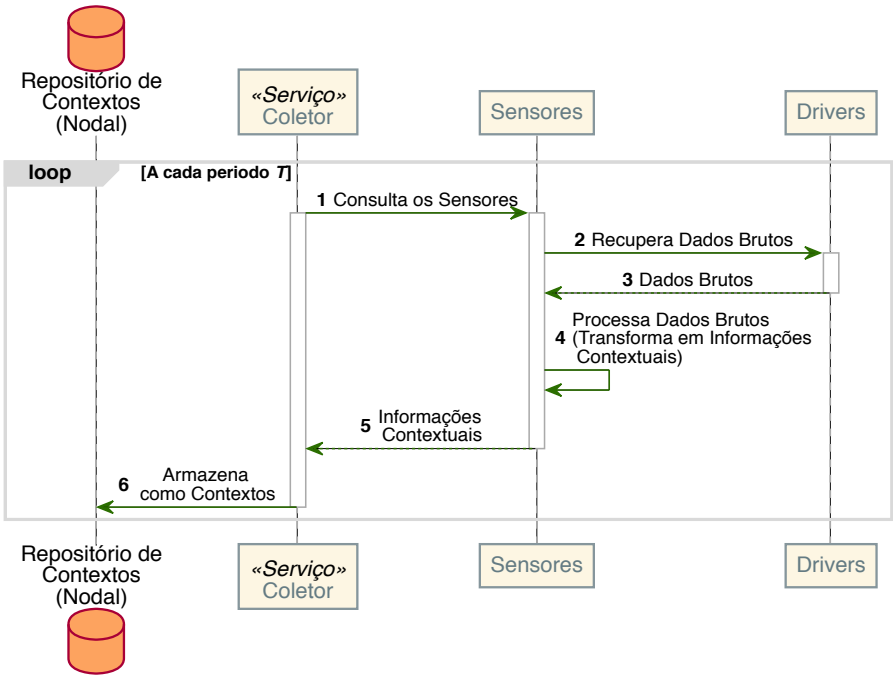



**Figura 2. Coleta das Informações Contextuais: Diagrama de Sequência**

- **Interpretação dos Contextos e Identificação de Situações** - A interpretação dos contextos e a identificação de situações no nodo físico ficam a cargo do Serviço de Interpretação de Contextos Nodal, componente de raciocínio do Gerenciador Nodal.
  Para esta identificação é realizada uma consulta ao Repositório de Contextos Nodal (vide Figura 3). As informações contextuais recuperadas são analisadas através da aplicação de regras que irão identificar a *situação* do nodo físico.
  A Situação no nível celular é construída a partir da notificação da situação dos nodos ao Subsistema de Identificação de Situações do *Gerenciador Celular*. Uma vez identificada uma *situação* que demande modificações no ambiente celular, é disparado um evento de adaptação correspondente a *situação* identificada, que será então tratado pelo *Subsistema de Adaptação* (Figura 4).

[2]Os termos fazem referência à arquitetura do *OpenStack*.

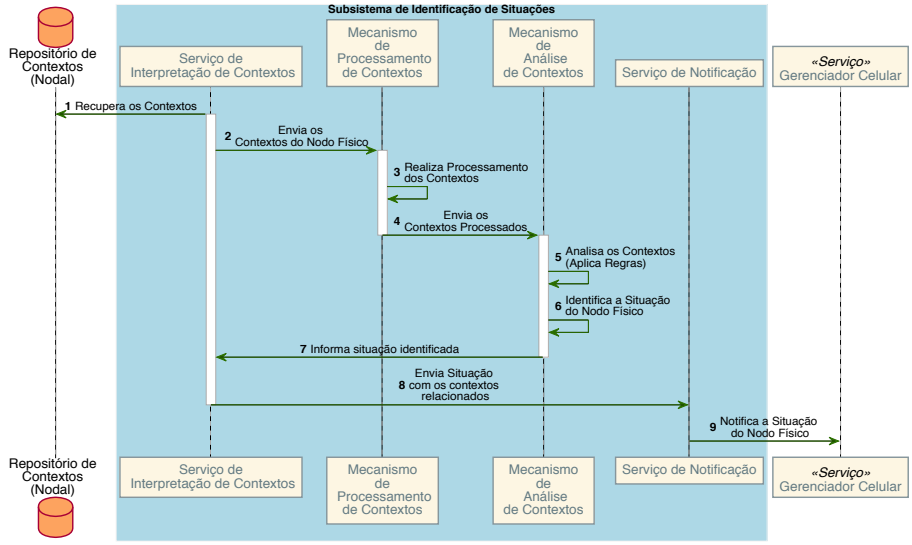



**Figura 3. Interpretação dos Contextos no Gerenciador Nodal: Diagrama de Sequência**

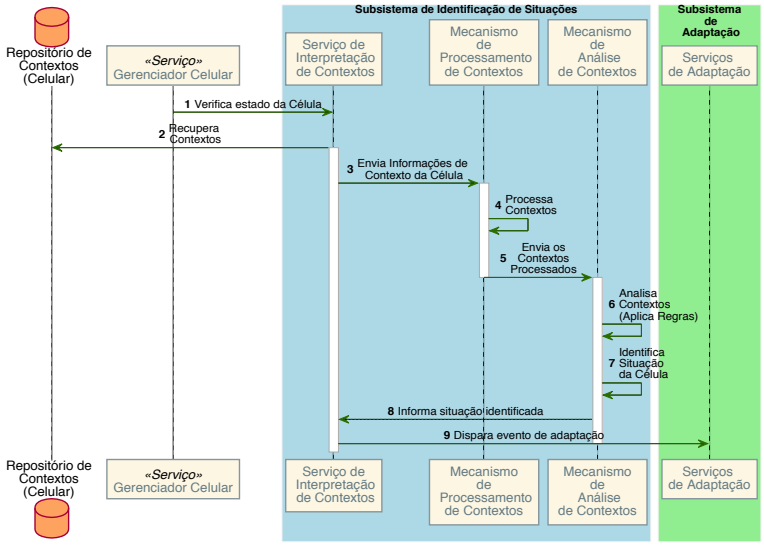



**Figura 4. Interpretação dos Contextos no Gerenciador Celular: Diagrama de Sequência**

- **Adaptação do Ambiente** - O *processo de adaptação celular* visa realizar mudanças na organização do ambiente em nuvem para que o mesmo atinja um objetivo definido por seus administradores ou reaja a eventos disparados pela detecção de uma determinada *situação*.
  Durante sua execução, o *Serviço de Orquestração* analisa o estado dos nodos físicos que compõem o ambiente celular, realiza a aplicação de *Políticas de Adaptação*[3] gerando a partir disso um *Plano de Adaptação*. Em seguida, aciona o *Serviço de Atuação*, cujo o conjunto de atuadores irá executar ações sobre a

[3]As políticas de adaptação podem suportar diversas heurísticas para a realização da alocação de instâncias. Atualmente é utilizado um algoritmo de alocação usando a abordagem *Best-Fit-Decreased* para realizar o processo de alocação das instâncias nos nodos físicos da *Célula*.

*Célula* de acordo com o *Plano de Adaptação* gerado.
A arquitetura atualmente está preparada para executar dois tipos de adaptação: (i) *Adaptação com Objetivo de Otimizar o Consumo Energético* (Figura 5), por meio do processo de consolidação das cargas e da desativação dos nodos físicos ociosos; e (ii) *Adaptação com Objetivo Otimizar o Balanceamento de Cargas* (Figura 6), entre nodos físicos.

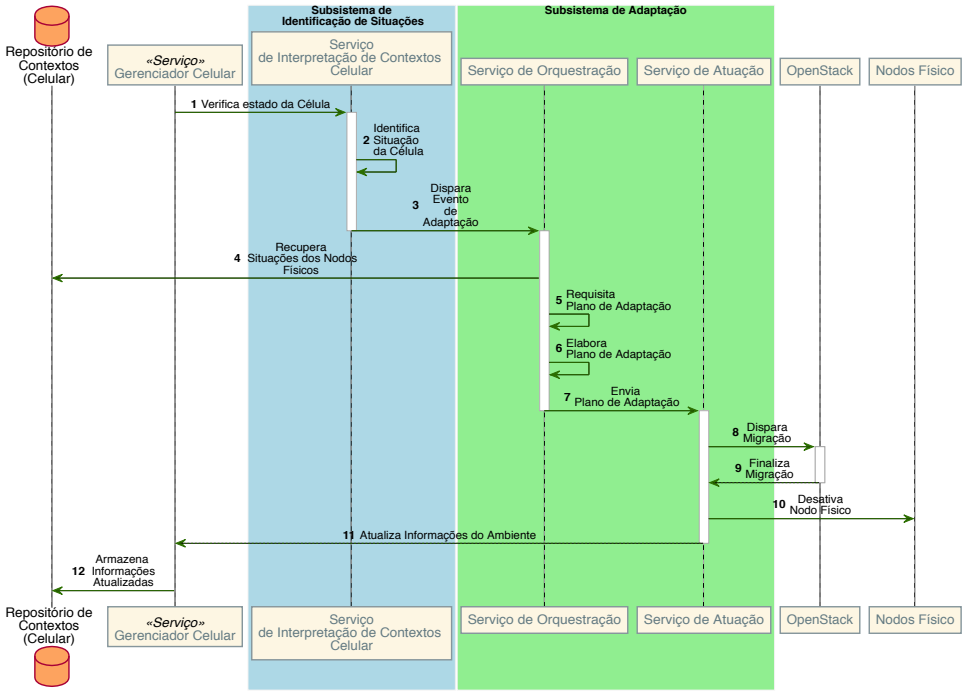



**Figura 5. Processo de Adaptação para Otimizar o Consumo Energético**

## 5. Avaliação Experimental e Resultados

Durante a avaliação da arquitetura *Samsara*, partiu-se da premissa que a mesma seria capaz de identificar períodos de subutilização ou sobrecarga dos nodos físicos e realizar adaptações na *Célula* de acordo com o estado geral da mesma. O objetivo aqui é que o consumo de energia da *Célula* corresponda à carga de trabalho a qual a mesma está submetida.

Para a execução do conjunto de avaliações, foram utilizados 5 servidores Dell PowerEdge T430, com a seguinte descrição cada: processador Intel(R) Xeon(R) CPU E5-2420 @ 1.90GHz (6 núcleos, 12 *threads*), 8GB DIMM DDR3 *Synchronous* 1600 MHz, executando *Ubuntu 14.04 LTS - Trusty x86_64*, versão estável dada a característica de possuir um suporte de longo prazo.

O ambiente em nuvem foi implementado sobre o *OpenStack* versão 2015.2 (*Liberty*) e seus servidores foram assim organizados: (i) um servidor escolhido como *Controller Node*, responsável por hospedar e executar o *Controlador Celular* da Samsara; (ii) e os demais quatro servidores como *Compute Nodes* responsáveis pela execução do *Controlador Nodal* da *Samsara* e pela hospedagem das máquinas virtuais.

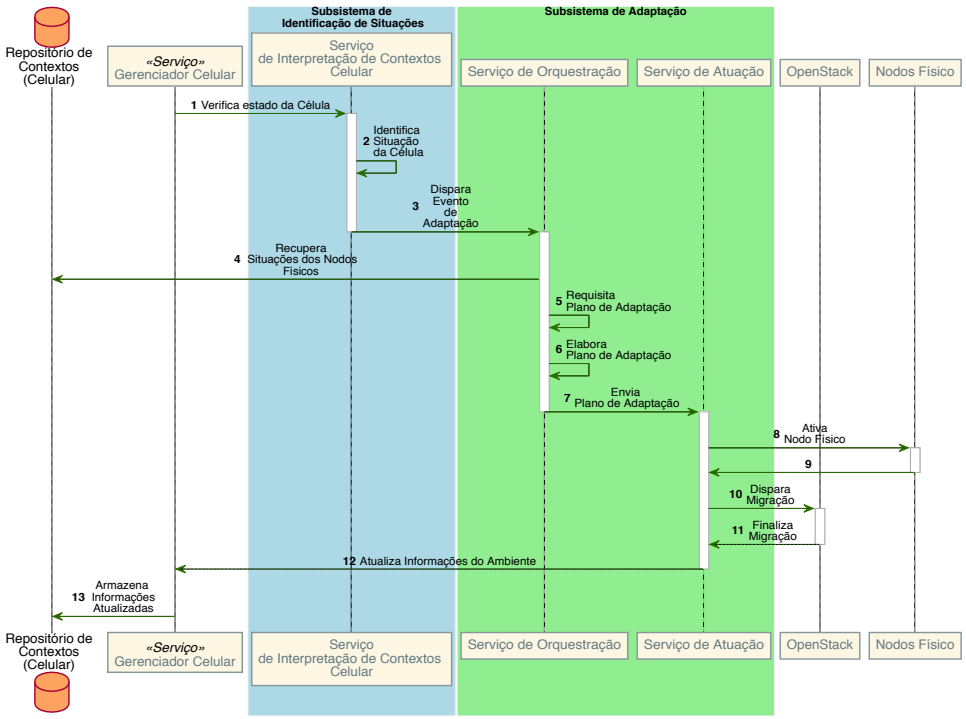



**Figura 6. Processo de Adaptação com Objetivo de Otimizar o Balanceamento de Carga**

Na avaliação da arquitetura *Samsara*, foram definidos dois cenários: (i) os mecanismos de adaptação da Célula não estão ativados, buscando construir uma base para avaliação dos mecanismos da arquitetura desenvolvida; e (ii) os mecanismos de adaptação da *Samsara* são ativados, configurando os valores para detecções de situações do nodo físico (NF) para trabalhar na faixa entre 30–70% de utilização de CPU (situação normal), identificando valores abaixo disso como subutilização e acima como sobrecarga, considerando em todas elas a existência de pelo menos uma máquina virtual ativa (MV), de acordo com a Tabela 1. Em ambos os cenários, as 30 máquinas virtuais são criadas e distribuídas entre os 4 nodos físicos de acordo com os mecanismos do *OpenStack*.

**Tabela 1. Regras do Nodo Físico**

| Situação | Contextos de Interesse | Regras | Ação |
|---|---|---|---|
| Sobrecarregado | Uso de CPU e Máquinas Virtuais Ativas | $CPU_{UsoMédio} \geq 70\%$ e $MV_{ativas} > 0$ | Notifica Situação |
| Subutilizado | Uso de CPU e Máquinas Virtuais Ativas | $CPU_{UsoMédio} \leq 30\%$ e $MV_{ativas} > 0$ | Notifica Situação |
| Ocioso | Uso de CPU e Máquinas Virtuais Ativas | $CPU_{UsoMédio} \leq 30\%$ e $MV_{ativas} = 0$ | Notifica Situação |

As configurações da célula foram ajustadas para identificar ineficiência energética quando pelo menos uma máquina estiver subutilizada nos últimos 5 minutos e para identificar sobrecarga se existir aos menos uma máquina sobrecarregada nos últimos 90 segundos. Essas regras são vistas na Tabela 2. Durante a execução dos experimentos baseados

no segundo cenário, a arquitetura *Samsara*, além de detectar as *situações* definidas em suas regras, deve adaptar-se de acordo com o *Plano de Adaptação* gerado.

**Tabela 2. Regras da Célula**

| Situação | Contextos de Interesse | Regras | Ação |
|---|---|---|---|
| Energeticamente Ineficiente | Situação dos Nodos Físicos | $NF_{subutilizado} \geq 2$ e $Periodo_{subutilizado} \geq 300s$ | Dispara Evento (Consolidação) |
| Energeticamente Otimizada | Situação dos Nodos Físicos | $(NF_{subutilizado} < 1$ e $Periodo_{subutilizado} \geq 300s)$ ou $(NF_{sobrecarregado} < 1$ e $Periodo_{sobrecarregado} \geq 90s)$ | - |
| Sobrecarregada | Situação dos Nodos Físicos | $NF_{sobrecarregado} \geq 1$ e $Periodo_{sobrecarregado} \geq 90s$ | Dispara Evento (Balanceamento) |

Em cada cenário, a *Célula* é submetida a um conjunto de cargas de trabalho pré-definidas que buscam explorar a utilização de CPU, organizadas por meio dos seguintes passos: (i) nos primeiros 3 minutos, nada é executado, sendo esse período usado para estabilização do sistema; (ii) nos próximos 55 minutos, são gerados valores entre 10 a 100% de utilização de CPU, executados em ordem decrescente, começando em 100% de ocupação de CPU, reduzindo 10% a cada período de execução até chegar à 10 % de utilização de CPU. Cada valor gerado é executado durante aproximadamente 5 minutos e 30 segundos; (iii) ao término da primeira etapa de execução, o processo inverter-se, a partir de então, são gerados valores em ordem crescente.

### 5.1. Avaliação dos Resultados

A avaliação dos resultados obtidos ao final da realização dos experimentos caracterizaram que a *Samsara* por meio de seus mecanismos, quando ativados, conseguiu atingir o objetivo de reduzir o consumo de energia, promovendo um melhor aproveitamento dos recursos físicos da nuvem computacional.

A Tabela 3 mostra as informações relacionadas ao **consumo médio de energia** registradas durante os experimentos.

**Tabela 3. Consumo Médio de Energia**

| Adaptação Desabilitada (MJ) | Adaptação Habilitada (MJ) | Redução (MJ) | Redução ($\mathrm{W\,h^{-1}}$) | Redução (%) |
|---|---|---|---|---|
| 2,04 | 1,79 | 0,25 | 69,44 | 12,31 |

A comparação entre os dois cenários mostrou que, ao habilitar os mecanismos de adaptação da arquitetura *Samsara*, o consumo de energia médio do ambiente foi reduzido em aproximadamente $12,31\%$. O consumo médio foi de $2,04$ MJ[4] no cenário de teste onde os mecanismos de adaptação são mantidos desabilitados e de $1,79$ MJ no cenário onde os mesmos são ativados, com uma redução de $0,25$ MJ. A redução

[4]Megajoule

alcançada quando convertida em *watt hora* ($\mathrm{W\,h^{-1}}$), mostra que foi possível economizar aproximadamente $69,44\mathrm{W\,h^{-1}}$. Os valores médios de consumo de energia acima discutidos foram obtidos com um intervalo de confiança (IC) de $95\%$.

Para observar o **comportamento do ambiente em relação ao consumo de energia**, foram executadas 30 rodadas de testes para cada cenário, selecionando-se dentre as mesmas as que representaram o *maior* e o *menor* consumo médio nos seus respectivos cenários. Para validação dos valores, foi utilizado o modelo de regressão local (*LOESS*). A Figura 7 mostra a variação da potência instantânea dos dois cenários, de forma que essa variação, corresponde a curva de variação da utilização de CPU durante o experimento.

É possível notar que a curva que representa a variação da potência instantânea ao longo de 120 minutos quando a adaptação está habilitada *(AH)*, por diversos momentos, mantém-se abaixo da curva que representa o cenário no qual a adaptação está desabilitada *(AD)*, tanto na comparação entre as rodadas de maior consumo *(AD4 e AH3)* quanto na comparação entre as rodadas de menor consumo *(AD13 e AH5)*. Essa variação na potência instantânea também corresponde aos momentos quando ocorre a desativação ou reativação dos nodos físicos, gerando uma redução ou um aumento na potência instantânea de forma mais significativa.

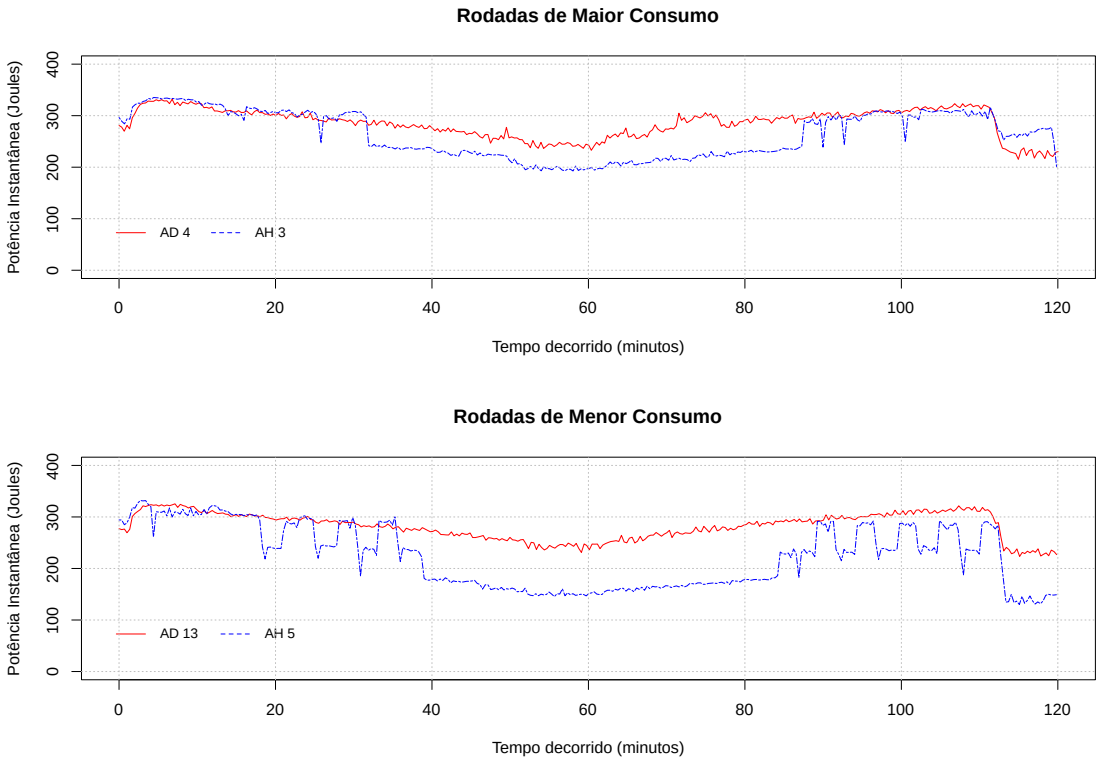



**Figura 7. Potência Instantânea ao longo de 120 minutos**

## 6. Trabalhos Relacionados

O trabalho desenvolvido em [Beloglazov 2013, Beloglazov and Buyya 2015] apresenta um *framework* distribuído para realizar a consolidação dinâmica de máquinas virtuais

com uma abordagem caracterizada agnóstica quanto ao tipo de carga de trabalho e às aplicações executadas. Ele aborda o problema de realizar a consolidação dinâmica de máquinas virtuais dividindo o mesmo em quatro subproblemas: (i) a detecção de períodos de subutilização dos recursos dos *hosts*; (ii) a detecção de períodos de sobrecarga dos recursos dos *hosts*; (iii) seleção das máquinas virtuais a serem migradas; e (iv) alocação das mesmas nos *hosts* através de *live-migration*.

Em [Feller et al. 2012, Feller et al. 2014] é apresentado o *Snooze*, um *framework* para gerenciamento de máquinas virtuais, ciente do consumo de energia e com uma abordagem holística quanto ao uso dos recursos, com aplicação prática de servir para gerenciar de forma eficiente um *datacenter* em produção através do uso da consolidação dinâmica de máquinas virtuais. Sua arquitetura é organizada por meio da implementação de uma gerência hierarquizada e em camadas, responsável coletar informações sobre a utilização de recursos, realizar a estimativa dos mesmos e determinar escalonamento das máquinas virtuais e as ações de gerenciamento de energia.

O projeto *Entropy* [Hermenier et al. 2009] visa realizar o gerenciamento de recursos de *clusters* de nodos homogêneos, realizando a consolidação dinâmica de máquinas virtuais com base na *Programação por Restrição* (*Constraint Programming*), considerando possíveis estados de sobrecarga nesse processo, principalmente a duração da migração de máquinas virtuais. Ele possui dois objetivos: (i) manter a configuração do *cluster*, de maneira que todas as máquinas virtuais são alocadas com recursos suficientes para realizar suas tarefas; e (ii) realizar a minimização do número de servidores (*hosts*) ativos como estratégia para economizar energia.

Os trabalhos relacionados apresentados buscam abordagens para o aumento da eficiência na utilização de recursos em ambientes computacionais de larga escala com foco na eficiência energética. A arquitetura apresentada nesse artigo aborda essa questões propondo elevar o nível de abstração para modelagem gerenciamento de recursos de uma nuvem computacional, empregando a *Ciência da Situação* para tratar com a complexidade relacionada com o gerenciamento desse tipo de ambiente.

## 7. Considerações Finais

O objetivo central desse trabalho foi a concepção de uma arquitetura que contribua para a qualificação dos mecanismos responsáveis pelo gerenciamento de recursos de um ambiente em nuvem. A mesma foi desenvolvida para gerenciar esse ambiente de forma autonômica e com intervenção humana mínima, buscando otimizar o uso dos recursos quanto a eficiência energética sem acarretar impacto significativo nos serviços fornecidos. Para tanto, a *Samsara* explora mecanismos para Ciência de Situação, tanto para determinar o estado atual do ambiente, quanto para atuar sobre o mesmo. Durante sua concepção foram considerados aspectos que contribuíssem com a flexibilidade e extensibilidade dos mecanismos da própria *Samsara*, permitindo adicionar diversos mecanismos e estratégias ao longo de sua arquitetura.

A abordagem utilizada para alcançar equilíbrio entre desempenho e consumo de energia, consiste em explorar a consolidação de cargas de trabalho, buscando aumentar o número de nodos que possam ser postos em um estado de minimização do consumo de energia. Durante períodos de aumento na demanda por processamento, ao identificar situações de sobrecarga no ambiente, a *Samsara* dispara o evento de adaptação que realiza

o balanceamento das cargas de trabalho entre os nodos físicos, o qual, se necessário, irá reativar nodos em repouso.

A arquitetura *Samsara* atingiu uma redução de consumo de energia ao redor de $12,31\%$ para as cargas de trabalho avaliadas. Estes resultados experimentais obtidos se mostraram promissores e apontam para continuidade das pesquisas.

Dentre os aspectos levantados para dar continuidade ao trabalho, destacam-se: (i) explorar a utilização de diferentes algoritmos para a detecção de eventos de subutilização e sobrecarga; (ii) explorar o suporte a outros mecanismos de atuação sobre o ambiente, como o redimensionamento de máquinas virtuais e as técnicas de *CPU pinning* e *CPU capping*; (iii) empregar lógica *fuzzy* nos mecanismos de interpretação e tomada de decisão; e (iv) desenvolver e disponibilizar uma API para a *Samsara*, bem como uma interface de gerenciamento acessível para administradores via navegador.

## Referências

Beloglazov, A. (2013). *Energy-Efficient Management of Virtual Machines in Data Centers for Cloud Computing*. PhD thesis, The University of Melbourne.

Beloglazov, A. and Buyya, R. (2015). Openstack neat: a framework for dynamic and energy-efficient consolidation of virtual machines in openstack clouds. *Concurrency and Computation: Practice and Experience*, 27(5):1310–1333.

Beloglazov, A., Buyya, R., Lee, Y. C., Zomaya, A., et al. (2011). A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Advances in Computers*, 82(2):47–111.

Feller, E., Rilling, L., and Morin, C. (2012). Snooze: A scalable and autonomic virtual machine management framework for private clouds. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 482–489. IEEE Computer Society.

Feller, E., Simonin, M., Jégou, Y., Orgerie, A.-C., Margery, D., and Morin, C. (2014). *Snooze: A Scalable and Autonomic Cloud Management System*. PhD thesis, Inria Rennes; INRIA.

Hermenier, F., Lorca, X., Menaud, J.-M., Muller, G., and Lawall, J. (2009). Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/-SIGOPS international conference on Virtual execution environments*, pages 41–50. ACM.

Liu, P., Jajodia, S., and Wang, C. (2017). *Theory and Models for Cyber Situation Awareness*, volume 10030. Springer.

Mastelic, T., Oleksiak, A., Claussen, H., Brandic, I., Pierson, J.-M., and Vasilakos, A. V. (2015). Cloud computing: Survey on energy efficiency. *Acm computing surveys (csur)*, 47(2):33.

Onwubiko, C. (2012). *Situational Awareness in Computer Network Defense: Principles, Methods and Applications: Principles, Methods and Applications*. Information Science Reference.

Varasteh, A. and Goudarzi, M. (2017). Server Consolidation Techniques in Virtualized Data Centers: A Survey. *IEEE Systems Journal*, 11(2):772–783.

# Análise da Virtualização do Controle de Congestionamento na Execução de Aplicações Hadoop MapReduce

**Vilson Moro, Maurício A. Pillon, Charles C. Miers, Guilherme P. Koslovski**

[1]Programa de Pós-Graduação em Computação Aplicada (PPGCA) - LabP2D
Universidade do Estado de Santa Catarina - UDESC - Joinville, SC - Brasil
*vilson.moro@edu.udesc.br, {mauricio.pillon, charles.miers, guilherme.koslovski}@udesc.br*

***Resumo.*** *Provedores de nuvens hospedam recursos de múltiplos clientes configurados com versões distintas de sistemas operacionais e bibliotecas. A variedade de algoritmos e parâmetros relacionados com o protocolo TCP constitui um cenário de comunicação heterogêneo, sendo que algoritmos não otimizados de controle de congestionamento podem comprometer o desempenho das aplicações. Devido ao total controle gerencial no data center, provedores podem aplicar a Virtualização do Controle de Congestionamento (VCC) obtendo algoritmos otimizados, independente da configuração da Máquina Virtual (MV). Na perspectiva dos inquilinos, a virtualização é uma operação transparente. Embora promissora, a aplicação de VCC requer uma análise profunda sobre o impacto causado nas aplicações finais. Nesse contexto, o presente trabalho analisa a execução de aplicações Hadoop MapReduce (HMR) sobre VCC. A análise experimental discute o tempo de execução e o comportamento das filas nos switches, ressaltando algumas limitações de VCC na execução de HMR.*

***Abstract.*** *Cloud providers host multiple virtual machines (VMs) configured based on specific versions of operating systems and libraries. The diversity of algorithms and parameters related to TCP constitutes a heterogeneous communication scenario. Legacy congestion control algorithms compromise the performance of VM-hosted applications. Due to total control in the data center, providers can apply the virtualization of congestion control (VCC) to generate optimized algorithms. From the tenant's perspective, virtualization is a transparently performed. Although promising, the application of VCC requires a deep analysis of the impact on the final applications. Thus, the present work dissects the execution of Hadoop MapReduce atop VCC-based scenarios. The experimental analysis discusses the execution time of Hadoop MapReduce and the behavior of intermediate switches queues, highlighting some VCC limitations.*

## 1. Introdução

Um dos principais adventos de nuvens computacionais que oferecem Infraestruturas como Serviço (IaaS) é observado na maleabilidade gerencial oferecida aos inquilinos. Especificamente, uma MV pode ser composta com diferentes configurações de memória, armazenamento e processamento. Sobretudo, o cliente possui total liberdade para controlar o sistema operacional (SO), atuando na instalação e atualização das aplicações, bibliotecas e núcleo do SO. Especificamente considerando o *Transmission Control Protocol* (TCP), dois cenários são observados quanto a heterogeneidade de configurações [Judd 2015]: *(i)* A complexidade na manutenção e as dependências de versões específicas (bibliotecas e

SOs) constituem fatores limitantes para a atualização das MVs. Em SOs com TCP não otimizado, os algoritmos do TCP para controlar e evitar congestionamento estão aquém dos últimos avanços. *(ii)* Ainda, configurações otimizadas relacionadas com os espaços de armazenamento temporários, algoritmos de partida lenta, confirmação de recepção seletiva, entre outros, podem burlar a equidade originalmente objetivada pelo TCP.

Algoritmos para controle de congestionamento auxiliam na recuperação do desempenho na ocorrência de gargalos, enquanto algoritmos para evitar congestionamento atuam na predição de eventuais perdas [Chiu and Jain 1989]. Tradicionalmente, o TCP controla e evita congestionamentos nas extremidades da rede [Jacobson 1988], sem suporte nativo dos equipamentos que compõem o núcleo, baseado apenas em informações inferidas sobre os fluxos de dados. Diversas versões do TCP foram propostas para otimizar o desempenho do tráfego em *data center*s (DCs) [Wu et al. 2012, Mittal et al. 2015, Alizadeh et al. 2010] alterando a interpretação do *feedback* binário originalmente concebido [Chiu and Jain 1989]. Em alguns casos, o núcleo da rede participa oferecendo marcações de pacotes com alertas de possíveis gargalos [Chowdhury et al. 2011]. De fato, é interesse do provedor de IaaS contornar o problema imposto pelo cenário heterogêneo de algoritmos de controle de congestionamento, qualificando o serviço ofertado aos clientes. Possíveis soluções compreendem a reserva de recursos de comunicação (largura de banda e configuração dos *switches*) [de Souza et al. 2017, Zahavi et al. 2016] e configuração dinâmica de filas de encaminhamento [Judd 2015]. Entretanto, as técnicas elencadas tendem a subutilizar os recursos de comunicação ou possuem elevada complexidade de gerenciamento [Popa et al. 2011]. Nesse contexto, a Virtualização do Controle de Congestionamento (VCC) foi proposta como uma alternativa simples para uniformizar os algoritmos não otimizados [Cronkite-Ratcliff et al. 2016, He et al. 2016]. Em resumo, os provedores possuem acesso administrativo aos *switches* virtuais ou hipervisores de MVs, responsáveis por encaminhar os pacotes entre as MVs dos inquilinos. Assim, uma camada de virtualização pode interceptar o tráfego não otimizado e manipular, quando necessário, para atender aos requisitos dos protocolos atualizados do DC.

Originalmente, VCC foi analisada com cargas sintéticas indicando a eficácia para cenários que objetivavam a equidade do compartilhamento e maximização da utilização em gargalos para fluxos majoritariamente longos [Cronkite-Ratcliff et al. 2016, He et al. 2016]. Entretanto, cargas de comunicação reais de aplicações tradicionalmente executada em DCs de nuvens não foram analisadas. Especificamente, aplicações baseadas em HMR possuem fluxos de comunicação com características distintas das inicialmente estudadas [Alizadeh et al. 2010, Roy et al. 2015]. Assim, o objetivo do presente trabalho é analisar a aplicabilidade de VCC para aplicações HMR. Como contribuição, o trabalho *(i)* analisa o tempo de execução do HMR, utilizando rastros reais de execução, em cenários heterogêneos (otimizados ou não) com VCC; *(ii)* discute o impacto percebido em HMR quanto à configuração da marcação nas filas dos *switches*; e *(iii)* correlaciona o descarte de pacotes nos *switches* com o tempo total de execução. Sobretudo, a análise experimental indica que aplicações baseadas em HMR sofrem um impacto profundo quando executadas em ambientes com VCC.

O trabalho está estruturado da seguinte maneira: Na Seção 2, a motivação é detalhada, revisando o fluxo de dados em HMR e o controle de congestionamento do TCP. VCC é introduzida na Seção 3, enquanto a análise experimental é descrita na Seção 4.

Trabalhos relacionados são apresentados na Seção 5 e as considerações finais na Seção 6.

## 2. Motivação e Definição do Problema

### 2.1. Fluxo de Dados em Hadoop MapReduce

O *framework* HMR é amplamente utilizado para processamento de dados estruturados e não estruturados [Dean and Ghemawat 2008]. É importante ressaltar que $33\%$ do tempo de execução do HMR é atribuído à tarefas de comunicação [Chowdhury et al. 2011], motivando a realização do presente trabalho. Sobretudo, aglomerados HMR apresentam uma dependência de localidade de dados, sobrecarregando *switches* e gerando congestionamentos durante diferentes etapas da execução [Roy et al. 2015].

DCs de nuvens hospedam aplicações com padrões distintos de comunicação. Aquelas baseadas em particionamento, processamento e agregação de informações, como HMR, trafegam dados de controle sensíveis à latência, bem como fluxos para sincronização de massas de dados. Ou seja, os fluxos sensíveis à latência concorrem com o tráfego de *background*. Analisando a ocupação das filas em *switches*, fluxos HMR maiores que $1$ MB possuem baixa multiplexação e consomem grande parcela do espaço disponível [Alizadeh et al. 2010], induzindo o aumento da latência para os demais fluxos.

DCs são projetados visando alta vazão e baixa latência. Entretanto, os *switches* comumente usados possuem arquiteturas baseadas em memória compartilhada. Assim, quando múltiplos fluxos convergem para a mesma interface *(incast)* [Wu et al. 2010], o espaço de armazenamento em filas pode ser totalmente consumido, ocasionando o descarte de pacotes [Alizadeh et al. 2010].

### 2.2. Controle de Congestionamento do TCP em DCs com Múltiplos Clientes

Originalmente, por ser baseado em *feedback* binário [Chiu and Jain 1989], o TCP apenas altera seu modo de transmissão na ocorrência de perda de segmentos. Na ausência de perdas, a janela de transmissão é aumentada, enquanto na ocorrência, a mesma é diminuída [Ha et al. 2008, Jacobson 1988]. Especificamente, o congestionamento é percebido quando um temporizador é finalizado ou quando um conjunto de ACKs duplicados são recebidos pelo emissor. Embora eficiente nos cenários originalmente propostos (principalmente relacionados com comunicação sobre redes com configurações e capacidades distintas), os algoritmos são inaptos para detectar e controlar o cenário previamente elencado para HMR, comum em DCs [Alizadeh et al. 2010].

Assim, a técnica de *Explicit Congestion Notification* (ECN), uma extensão da pilha TCP/IP, alterou o *feedback* recebido pelo emissor TCP, ou seja, a perda de pacotes é eventualmente substituída por um aviso da possível ocorrência de congestionamento. De posse do aviso, o emissor pode preventivamente reduzir o volume de dados trafegados, atenuando a formação de filas nos dispositivos intermediários. Quanto aos protocolos, a implementação desse mecanismo foi realizada através da introdução de $2$ bits na camada de rede, *ECN-Capable Transport* (ECT) e *Congestion Experienced* (ECE). O primeiro, indica que o equipamento é capaz de transportar a notificação de congestionamento, enquanto o segundo sinaliza que uma situação de congestionamento está acontecendo. Na camada de transporte, quando o bit *ECE* é percebido, o destinatário acrescenta a mesma informação ao segmento de confirmação de recebimento. Ao receber a notificação, o

emissor reduz a janela de congestionamento para evitar a perda de segmentos, e informa sua ação ao receptor, através do bit *Congestion Window Reduced* (CWR).

A marcação de pacotes que experimentam congestionamento é realizada pelos *switches* intermediários. Para tal, um monitoramento constante das filas de encaminhamento é realizado, disparando ações de acordo com parâmetros previamente informados. Nesse contexto, configurações de *Random Early Detection* (RED) [Wu et al. 2012] podem ser aplicadas, especificando: *(i)* a largura de banda máxima do enlace (bytes/s); *(ii)* tamanho mínimo, máximo e instantâneo (para atender rajadas) da fila (bytes); *(iii)* tamanho médio dos pacotes; *(iv)* tolerância para o tamanho instantâneo da fila em pacotes; e *(v)* probabilidade de descarte (de 0,0 a 1,0). Quando o pacote é recebido pelo equipamento e se mantêm abaixo do mínimo especificado, não ocorre marcação. Entretanto, pacotes que estão entre os limiares mínimo e máximo são marcados de acordo com a probabilidade informada. Por fim, os pacotes acima do limiar máximo são descartados.

É fato que ECN permitiu um melhor compartilhamento dos recursos de comunicação em DCs [Alizadeh et al. 2011]. Todavia, a efetiva aplicação requer uniformidade dos algoritmos, ou seja, os servidores devem compreender o significado das marcações efetuadas pelo núcleo da rede. Esse requisito não é observado em DCs de nuvens IaaS, sendo que cada inquilino pode executar versões distintas. Ainda, diversas versões de SOs não possuem suporte ativado por padrão para ECN [Kühlewind et al. 2013]. Ou seja, embora o DC implemente mecanismos para otimizar o tráfego, as configurações realizadas nas MVs dos usuários podem estar em desacordo com o cenário ideal.

## 3. Virtualização do Controle de Congestionamento

A virtualização do controle de congestionamento consiste em criar uma camada para tradução do protocolo TCP utilizado pelas MVs em uma versão otimizada e reconhecida pelo DC [He et al. 2016, Cronkite-Ratcliff et al. 2016]. Dessa forma, a comunicação ocorre com a versão TCP selecionada pelo provedor. É importante ressaltar que nenhuma alteração é efetuada nos hospedeiros finais (MVs ou SOs). Duas abordagens para implementação da VCC foram recentemente propostas. O AC/DC [He et al. 2016] implementa VCC em *switches* virtuais, obtendo uma fina granularidade no controle. Assim, algoritmos podem ser selecionados para diferentes tipos de fluxos, por exemplo, Cubic para fluxos externos ao DC e *Data Center TCP* (DCTCP) para fluxos internos. Como o controle é implementado no *datapath* do *switch* virtual [Pfaff et al. 2015], todo o tráfego pode ser monitorado. Por sua vez, em [Cronkite-Ratcliff et al. 2016], a VCC foi implementada diretamente no hipervisor de MVs.

Em ambos, a percepção de congestionamento é obtida através dos pacotes marcados pelo núcleo da rede. A Figura 1 apresenta a arquitetura canônica para VCC. Quando uma aplicação não otimizada (ou legada) estabelece uma conexão, o hipervisor (ou *switch* virtual) monitora a troca de pacotes e introduz as informações necessárias para que o tráfego não otimizado seja reconhecido pela rede como tráfego capaz de suportar ECN. Inicialmente, o TCP não otimizado envia um pacote solicitando a conexão (1), que é interceptado (2) para acrescentar as informações de suporte ECN. O destinatário responde confirmando o estabelecimento (3, ECE). Novamente, o pacote é interceptado para notificar o remetente com um reconhecimento da sincronização (4). É importante observar que

a confirmação enviada para o emissor com algoritmo TCP não otimizado não possui as informações sobre ECN, que foram removidas pelo virtualizador. O envio efetivo dos dados inicia (5), sendo interceptado para acréscimo do bit ECT, informando que esse fluxo é capaz de transportar informação de congestionamento (6). Posteriormente, o destinatário reconhece o pacote recebido (7). O remetente recebe do hipervisor o reconhecimento de pacote (8). No caso de possível ocorrência de congestionamento na rede do DC, o bit *ECE* é ativado (9). Quando ocorre congestionamento, o emissor com algoritmo não otimizado é obrigado a reduzir o envio de dados através do estrangulamento da janela de receptação (10), realizado pelo hipervisor. Por fim, o remetente continua o envio de dados (11) para o hipervisor que transfere os dados e o tamanho da janela ajustada para o destinatário (12).

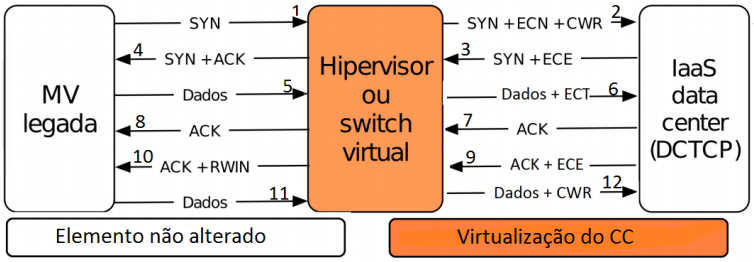



**Figura 1. Exemplo de utilização da VCC.**

Para induzir a desaceleração de transmissão do hospedeiro com TCP não otimizado sem aplicar uma técnica intrusiva, o controle de congestionamento é aplicado sobre a janela de recepção, *Receiver Window* (RWND). As informações internamente aferidas pelo algoritmo na MV, sobre a janela de congestionamento, *Congestion Window* (CWND), permanecem inalteradas. Nativamente, o TCP verifica $min(cwnd, rwnd)$ para identificar o volume que podem ser trafegado. Com VCC, o valor de RWND é alterado para representar o correto valor de CWND calculado pelo algoritmo do virtualizador, baseado em ECN. Por exemplo, uma MV utilizando o Cubic para controle de congestionamento, sem suporte à ECN, será convertida em um controle realizado por DCTCP.

Como a implementação do mecanismo AC/DC não é disponibilizada à comunidade, foi selecionado a proposta de [Cronkite-Ratcliff et al. 2016] para realização da análise na Seção 4. O VCC foi implementado em Linux aplicando uma alteração no núcleo do hipervisor. Em suma, um conjunto de *buffers* intermediários são criados para cada conexão TCP, ou seja, o hipervisor monitora a comunicação das MVs. Para diferenciar fluxos não otimizados e de configurações ECN, os recursos do VCC e ECN são ativados diretamente através da manipulação do *sysctl*.

## 4. Análise Experimental

### 4.1. Ambiente, Métricas e Rastros de Execução

Os experimentos objetivam a análise do impacto da VCC na execução de aplicações HMR, quando executadas em MVs com TCP não otimizado. O tráfego de dados em HMR foi emulado pelo MRemu [Neves et al. 2015] usando rastros da execução do *benchmark* HiBench em um aglomerado composto por 16 servidores (12 núcleos $x86\_64$ e 128 GB RAM), interconectados por enlaces de 1 Gbps. É importante ressaltar que MRemu emula apenas o tráfego HMR, objeto de avaliação do presente trabalho, sem efetuar o real processamento de dados. A comunicação de MRemu é efetuada sobre Mininet [Lantz et al. 2010].

Para composição do cenário experimental, 16 hospedeiros virtuais foram utilizados, interconectados por uma topologia *Dumbbell* composta por 2 *switches*. *Dumbbell* foi selecionada por simplificar a representação da ocorrência de disputa por recursos e consequentemente gargalos. Cada *switch* conecta 8 servidores com enlaces individuais de 1 Gbps. Um enlace de igual capacidade interconecta os *switches*, compondo o gargalo. O tráfego de *background* foi introduzido com a ferramenta *iperf*, variando o número de pares TCP comunicantes para representar múltiplos inquilinos disputando os recursos de comunicação. Os servidores *iperf* estão conectados no *switch 1* enquanto os clientes estão conectados no *switch 2* (origem dos dados). Quanto ao hospedeiro físico, MRemu e Mininet foram executados em um equipamento com SO GNU/Linux Ubuntu 14.04, processador AMD Phenom II X4 e 8 GB RAM. As MVs executaram originalmente o TCP *New Reno*. Por fim, os recursos de processamento e armazenamento não representaram gargalos pois MRemu apenas emula a comunicação.

Para analisar os resultados, quatro métricas foram coletadas: *(i)* tempo de execução do HMR; *(ii)* volume de pacotes descartados pelos *switches*; *(iii)* ocupação das filas dos *switches*; e *(iv)* volume do tráfego de *background*. A primeira métrica representa a visão do inquilino, enquanto que as demais métricas permitem a análise do descarte e a formação de fila nos *switches* oriundos de tráfegos TCP não otimizados, virtualizados ou otimizados, ou seja, a perspectiva do provedor IaaS.

### 4.2. Cenários Experimentais

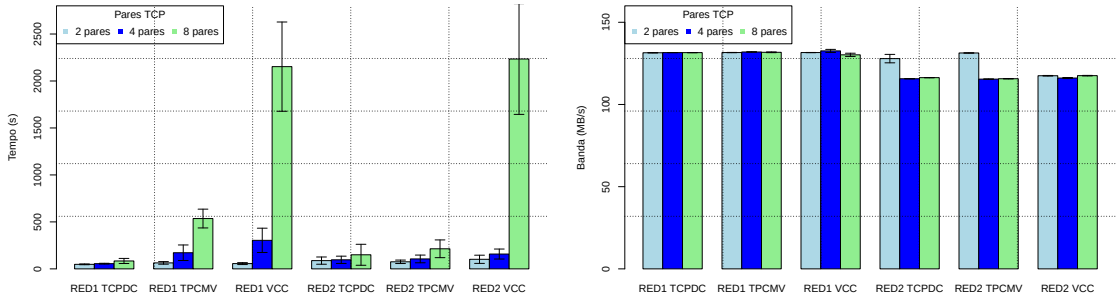
Para composição do cenário experimental, três configurações TCP foram utilizadas:
*(i)* execução com TCP não otimizado sem suporte ao ECN (TCPMV);
*(ii)* com TCP otimizado pelo DC, com suporte ECN (TCPDC); e
*(iii)* VCC.
O tráfego de *background* executou com TCPDC para todos os cenários. Por sua vez, o HMR foi executado sobre TCPMV, TCPDC e VCC. Quanto as configurações do ECN e RED, realizadas nos *switches*, duas configurações oriundas da bibliografia foram aplicadas (Tabela 1) [Alizadeh et al. 2010, Cronkite-Ratcliff et al. 2016]. Na primeira configuração, RED1, os valores de *min* e *max* estão próximos, portanto, sem margem de adequação. Além disso, o parâmetro *prob* está configurado para marcar 100% dos pacotes, forçando a redução rápida do tráfego. Na configuração identificada como RED2, tem-se um intervalo de adequação estabelecido entre os valores *min* e *max*. Este intervalo permite a adequação do tráfego de acordo com a notificação de congestionamento através da marcação de pacotes. Ainda, existe a diferença no tamanho máxima da fila. É importante ressaltar que a configuração dos *switches* é realizada pelo provedor, sem a participação dos inquilinos.

| **Configuração** | ***min*** | ***max*** | ***limit*** | ***burst*** | ***prob*** |
|---|---|---|---|---|---|
| RED1 | 90000 | 90001 | 1M | 61 | 1,0 |
| RED2 | 30000 | 90000 | 400K | 55 | 1,0 |

**Tabela 1. Configurações RED para marcação nas filas dos *switches*.**

### 4.3. Discussão dos Resultados

Para cada cenário foram executadas 10 rodadas e os gráficos compreendem média, desvio padrão e variabilidade dos dados. A primeira análise foca no impacto do protocolo TCP escolhido e das duas configurações do RED, na visão do inquilino. Para tal, observou-se o tempo de execução do HMR, considerando que, quanto menor o tempo, melhor o resultado. Na Figura 2(a), pode-se observar que TCPDC tem os menores valores, independente do número de pares concorrentes e da configuração do RED. TCPMV é o protocolo mais suscetível a diferença de configuração do RED. Por exemplo, com 8 pares, o tempo médio com RED1 de $535s$ foi reduzido para $213s$ com RED2. Constate-se, neste caso, a importância da configuração adequada do RED ao comportamento da aplicação.




(a) Tempo de execução do HMR.

(b) Tráfego de *background*.

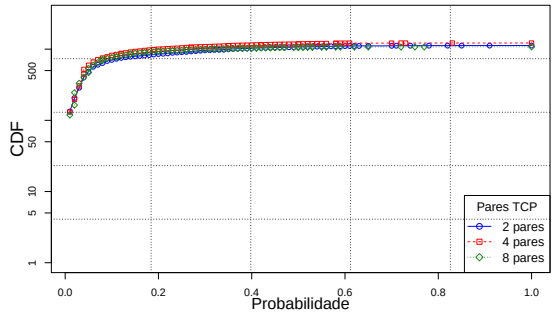
**Figura 2. Tempo de execução do HMR e vazão do tráfego de *background*.**

A maior peculiaridade dos resultados foi o comportamento do VCC. Este obteve resultados compatíveis com os demais protocolos quando analisado com 2 pares, tanto com RED1 quanto RED2. Com 4 pares, manteve-se competitivo somente com RED1, e demonstrou-se ineficiente com 8 pares. Com 8 pares, enquanto o HMR com TCPDC em RED1 finalizou com tempo médio de $84s$, o VCC chegou a $2153s$. Resultados piores foram observados com RED2, chegando a $2234s$. Diferente dos resultados inicialmente obtidos com VCC [Cronkite-Ratcliff et al. 2016, He et al. 2016], as aplicações HMR analisadas sofrem uma sobrecarga considerável no tempo de execução. Ainda, a aplicação de VCC é relacionada com as configurações ECN e RED, abstraídas do inquilino.

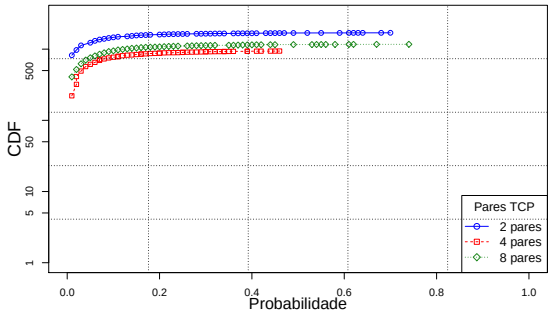
Por outro lado, o tráfego de *background* (Figura 2(b)) é praticamente o mesmo, independente do protocolo TCP escolhido, quando se tem a configuração RED1. Com RED2, o comportamento difere somente com 2 pares. O *iperf* atinge valores próximos a 130MB/s com TCPDC e TCPMV e 120MB/s com o VCC. A menor vazão obtida com RED2 é oriunda da marcação conservadora de pacotes, ou seja, devido ao intervalo largo de marcação, os emissores TCP possuem um comportamento conservador, reduzindo o envio de dados para evitar a ocorrência de congestionamento.

A segunda abordagem é a análise na visão do provedor. Neste caso, a observação está no tráfego dos *switches*, através da ocupação das filas. Os gráficos das Figuras 3 e 4 têm em seu eixo $x$ a probabilidade de ocorrência e, no eixo $y$, o valor acumulado das filas (expresso como CDF - *Cumulative Distribution Function*). A tendência do comportamento de formação de filas nos *switches* é quanto maior o número de pares TCP, maior o número de filas. Este comportamento pode ser observado em todas as variações de

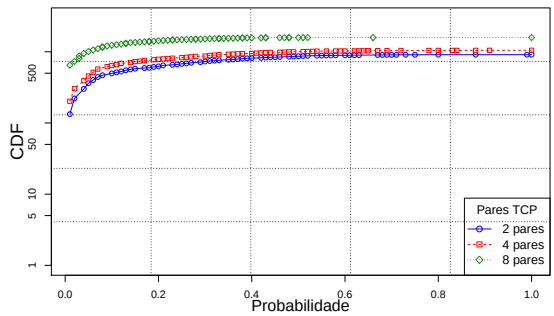
cenários para os protocolos TCPMV (Figuras 3(c), 3(d), 4(c) e 4(d)) e VCC (Figuras 3(f), 4(e) e 4(f)), com a única exceção Figura 3(d) que são iguais. Para o protocolo TCPDC e RED2, independente do *switch*, 2 pares têm maiores valores acumulados de ocorrência de filas do que 8 pares, indicando a total ocupação (400000 bytes) e mantendo a equidade de compartilhamento (indicado pela vazão, Figura 2).



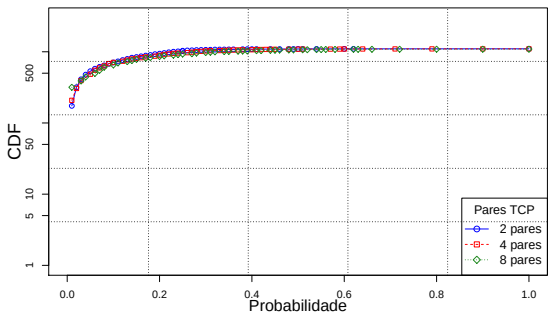

(a) TCPDC com RED1.



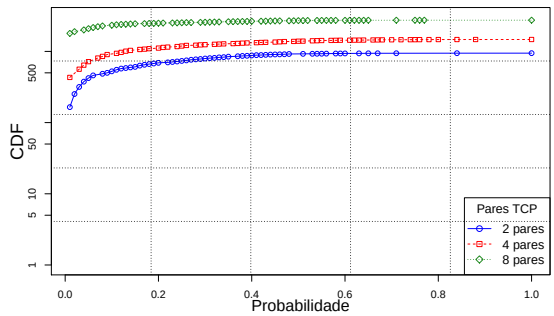

(b) TCPDC com RED2.



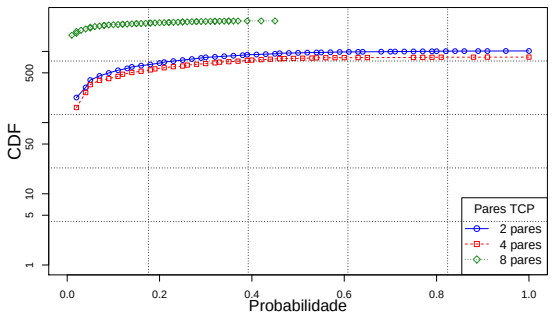

(c) TCPMV com RED1.




(d) TCPMV com RED2.



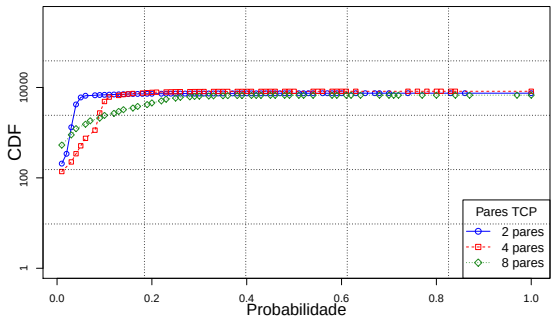

(e) VCC com RED1.



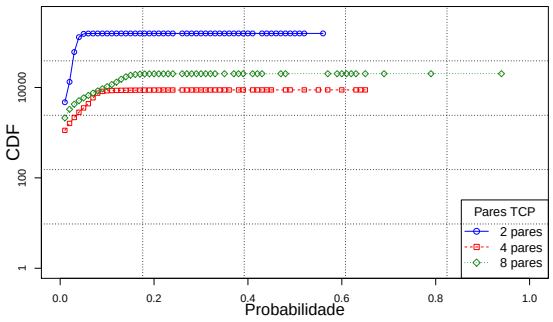

(f) VCC com RED2.

**Figura 3. Ocupação das filas no *switch* 1.**

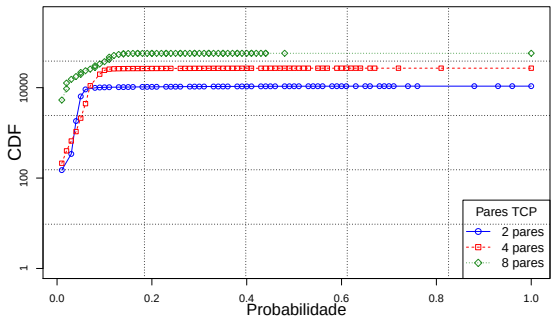
Para finalizar a análise, contabilizou-se o número de pacotes descartados em cada cenário (Figuras 5(a) e 5(b)). As menores perdas são com o protocolo TCPDC, independente do número de pares ou *switch*. No *switch* 1 (conectando HMR e servidores *iperf*), o número de pacotes descartados com RED2 é superior ao número descartado por RED1, tanto para TCPMV quanto para VCC. Para o *switch* 2 (conectando HMR e clientes *iperf*), o destaque é o elevado número de pacotes descartados pelo VCC para 8 pares, independente da configuração RED adotada.



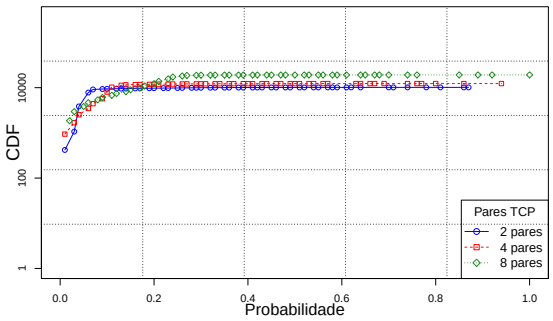

(a) TCPDC com RED1.



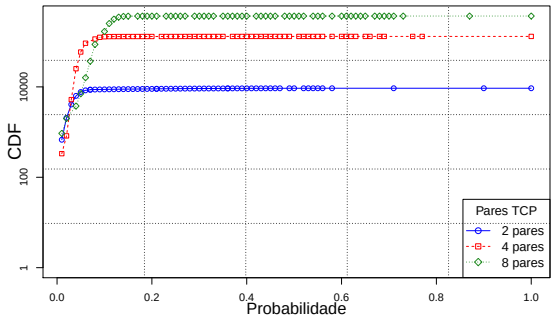

(b) TCPDC com RED2.



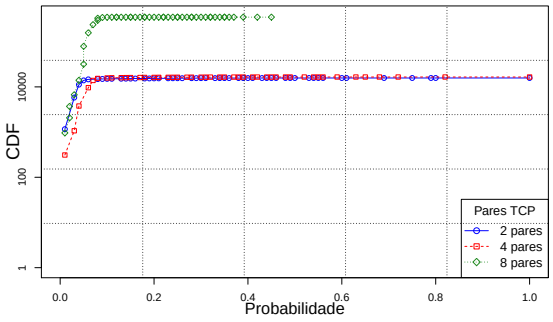

(c) TCPMV com RED1.




(d) TCPMV com RED2.

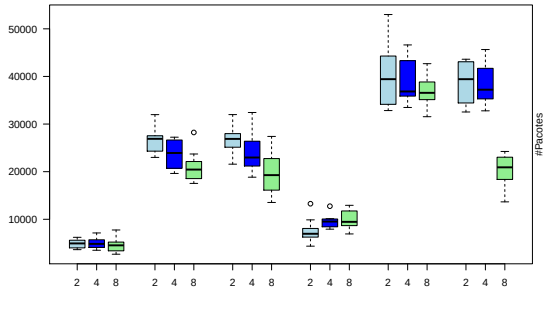



(e) VCC com RED1.

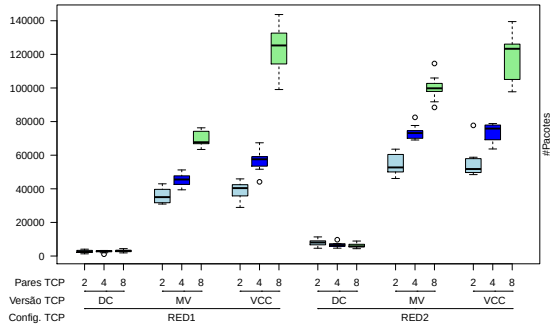



(f) VCC com RED2.

**Figura 4. Ocupação das filas no *switch* 2.**




(a) *Switch* 1.




(b) *Switch* 2.

**Figura 5. Pacotes descartados nos *switches*.**

Por fim, é importante ressaltar que aproximadamente $75\%$ dos fluxos de dados trafegados nas aplicações HMR analisadas transportam no máximo $7$ MB, sendo $10$ MB o maior volume trafegado entre os pares HMR. O volume é inferior ao originalmente analisado em VCC [Cronkite-Ratcliff et al. 2016, He et al. 2016], sendo suscetível ao impacto da marcação ECN. A ocupação das filas nos *switches* indicou que a configuração do RED é essencial e diretamente relacionada com a aplicação. Mesmo controlando a ocupação, o volume de pacotes descartados (Figura 5, sobretudo para o *switch* 2) é abusivo para VCC, estando aquém do cenário esperado (TCPDC). Em resumo, a análise evidenciou que a aplicação de VCC é promissora por buscar a equidade em um ambiente heterogêneo como nuvens IaaS (indicado pela vazão de *background*, Figura 2(b)). Entretanto, a utilização de VCC para aplicações HMR requer a configuração adequada das filas dos *switches*, tarefa comumente realizada pelos provedores por necessitar acesso administrativo [Judd 2015, Popa et al. 2011].

## 5. Trabalhos Relacionados

Os trabalhos relacionados compreendem estudos sobre notificação explícita de congestionamento, técnicas para VCC, caracterização e otimização do tráfego HMR em DC. Inicialmente, [Floyd 1994] discute o uso do ECN no TCP. As simulações utilizando o algoritmo *New Reno* e marcações guiadas por RED indicaram os benefícios deste recurso no controle de congestionamento, evitando perdas ao antecipar e prever situações prováveis de saturação da rede. Por sua vez, [Kühlewind et al. 2013] aborda a evolução histórica do ECN em SOs, apresentando uma visão de outras formas de prevenir e contornar congestionamento. A justiça no compartilhamento de canais congestionados foi analisada em [Hasegawa and Murata 2001]. Em suma, os trabalhos indicam a ausência de uma solução *de facto* para contornar a disparidade de compartilhamento quando múltiplos algoritmos para controle de congestionamento estão compartilhando gargalos.

Consciente da diversidade de aplicações executando em um DC de nuvem com pilhas TCP/IP desatualizadas, os autores [Cronkite-Ratcliff et al. 2016, He et al. 2016] propuseram a flexibilização na configuração, definindo o conceito de VCC. Os trabalhos motivaram a realização da presente análise com HMR. Por sua vez, [Alizadeh et al. 2010] identificou padrões de comunicação em DC que hospedam HMR: tráfego de consulta, tráfego de *background*, fluxos concorrentes e de diferentes tamanhos. De forma complementar, [Wu et al. 2012] aborda o congestionamento causado por tráfego *incast* que ocorre quando múltiplos fluxos convergem para o mesmo receptor.

Quanto ao gerenciamento de DCs IaaS, [Popa et al. 2011] aborda a dura negociação ao estabelecer políticas de alocação de largura de banda visando garantir uma proporcionalidade de utilização da rede, concedendo uma garantia mínima para o fluxo das MVs e ao mesmo tempo evitar ociosidade, induzindo a alta ocupação da rede. Por sua vez, [Zahavi et al. 2016] propõe uma nova abstração de serviços de nuvem, denominada Links como Serviço (LaaS), com isolamento entre os enlaces de comunicação virtualizados. O cliente pode optar por introduzir no enlace o algoritmo de controle de congestionamento que melhor atende as necessidades de sua aplicação, ou seja, a decisão não pertence ao provedor.

## 6. Considerações Finais & Trabalhos Futuros

Nuvens que oferecem IaaS permitem que inquilinos instalem e configurem sistemas operacionais de acordo com os requisitos das aplicações hospedadas. Tal advento difundiu a maleabilidade de MVs, entretanto, resultou em um cenário heterogêneo em execução sobre o DC. Sobretudo, diversos clientes não atualizam as bibliotecas, módulos e SOs devido a restrições técnicas de aplicações legadas. Consequentemente, versões desatualizadas de algoritmos para controle de congestionamento competem com versões atualizadas em DCs. Para amenizar a disparidade, a virtualização do controle de congestionamento foi recentemente proposta.

O presente trabalho investigou a aplicação da VCC na execução de aplicações HMR, executadas em MVs com TCP não otimizado. Analisando os resultados, evidenciou-se que a virtualização é diretamente dependente das configurações aplicadas nos *switches*. Sobretudo, os experimentos seguindo as configurações indicadas pela literatura (denominados RED1 e RED2) resultaram em um impacto no tempo de execução das aplicações HMR. Como perspectiva de continuidade, é evidente que VCC é uma tecnologia promissora, entretanto, como os resultados indicaram, a definição de configurações RED e ECN otimizadas são essenciais e naturalmente uma linha de continuidade.

**Agradecimentos:** Financiado pela UDESC e FAPESC, sendo desenvolvido no LabP2D.

## Referências

Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. (2010). Data center tcp (dctcp). *SIGCOMM Comput. Commun. Rev.*, 41(4):–.

Alizadeh, M., Javanmard, A., and Prabhakar, B. (2011). Analysis of dctcp: Stability, convergence, and fairness. In *Proc. of the SIGMETRICS Joint Int. Conf. on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, pages 73–84. ACM.

Chiu, D.-M. and Jain, R. (1989). Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17(1):1 – 14.

Chowdhury, M., Zaharia, M., Ma, J., Jordan, M. I., and Stoica, I. (2011). Managing data transfers in computer clusters with orchestra. *SIGCOMM Comput. Commun. Rev.*, 41(4):98–109.

Cronkite-Ratcliff, B., Bergman, A., Vargaftik, S., Ravi, M., McKeown, N., Abraham, I., and Keslassy, I. (2016). Virtualized congestion control. In *Proceedings of the 2016 ACM SIGCOMM Conf.*, SIGCOMM '16, pages 230–243, New York, NY, USA. ACM.

de Souza, F. R., Miers, C. C., Fiorese, A., and Koslovski, G. P. (2017). QoS-Aware Virtual Infrastructures Allocation on SDN-based Clouds. In *Proc. of the Int. Symp. on Cluster, Cloud and Grid Computing*, CCGRID, Madrid, Spain. IEEE.

Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

Floyd, S. (1994). Tcp and explicit congestion notification. *SIGCOMM Comput. Commun. Rev.*, 24(5):8–23.

Ha, S., Rhee, I., and Xu, L. (2008). Cubic: A new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74.

Hasegawa, G. and Murata, M. (2001). Survey on fairness issues in tcp congestion control mechanisms.

He, K., Rozner, E., Agarwal, K., Gu, Y. J., Felter, W., Carter, J., and Akella, A. (2016). Ac/dc tcp: Virtual congestion control enforcement for datacenter networks. In *Proc. of the SIGCOMM Conf.*, pages 244–257. ACM.

Jacobson, V. (1988). Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329.

Judd, G. (2015). Attaining the promise and avoiding the pitfalls of tcp in the datacenter. In *Proc. of the 12th Conf. on Networked Systems Design and Implementation*, NSDI'15, pages 145–157, Berkeley, CA, USA. USENIX.

Kühlewind, M., Neuner, S., and Trammell, B. (2013). On the state of ecn and tcp options on the internet. In *Proc. of the 14th Int. Conf. on Passive and Active Measurement*, PAM'13, pages 135–144, Berlin, Heidelberg. Springer-Verlag.

Lantz, B., Heller, B., and McKeown, N. (2010). A network in a laptop: Rapid prototyping for software-defined networks. In *Proc. of the 9th SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6. ACM.

Mittal, R., Lam, V. T., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., and Zats, D. (2015). Timely: Rtt-based congestion control for the datacenter. *SIGCOMM Comput. Commun. Rev.*, 45(4):537–550.

Neves, M. V., Rose, C. A. F. D., and Katrinis, K. (2015). Mremu: An emulation-based framework for datacenter network experimentation using realistic mapreduce traffic. In *Proc. of 23rd MASCOTS*, pages 174–177. IEEE.

Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., and Casado, M. (2015). The design and implementation of open vswitch. In *12th USENIX Symp. on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA. USENIX Association.

Popa, L., Krishnamurthy, A., Ratnasamy, S., and Stoica, I. (2011). Faircloud: Sharing the network in cloud computing. In *Proc.s of the 10th Workshop on Hot Topics in Networks*, HotNets-X, pages 22:1–22:6. ACM.

Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. (2015). Inside the social network's (datacenter) network. *SIGCOMM Comput. Commun. Rev.*, 45(4):123–137.

Wu, H., Feng, Z., Guo, C., and Zhang, Y. (2010). Ictcp: Incast congestion control for tcp. In *ACM CONEXT 2010*. Association for Computing Machinery, Inc.

Wu, H., Ju, J., Lu, G., Guo, C., Xiong, Y., and Zhang, Y. (2012). Tuning ecn for data center networks. In *Proc. of the 8th Int. Conf. on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 25–36. ACM.

Zahavi, E., Shpiner, A., Rottenstreich, O., Kolodny, A., and Keslassy, I. (2016). Links as a service (laas): Guaranteed tenant isolation in the shared cloud. In *2016 ACM/IEEE Symp. on Architectures for Networking and Communications Systems (ANCS)*, pages 87–98.

# An Interference-aware Virtual Machine Placement Strategy for High Performance Computing Applications in Clouds

**Maicon Melo Alves[1,2], Luan Teylo[1], Yuri Frota[1] and Lúcia Maria de A. Drummond[1]**

[1]Universidade Federal Fluminense

[2]Petroleo Brasileiro, S.A. - Petrobras

{mmelo,luanteylo,yuri,lucia}@ic.uff.br

***Abstract.*** *Cross-interference may happen when applications share a common physical machine, affecting negatively their performances. This problem occurs frequently when high performance applications are executed in clouds. Some papers of the related literature have considered this problem when proposing strategies for Virtual Machine Placement. However, they neither have employed a suitable method for predicting interference nor have considered the minimization of the number of used physical machines and interference at the same time. In this paper, we define the Interference-aware Virtual Machine Placement Problem for HPC applications in Clouds (IVMP) that tackles both problems by minimizing, at the same time, the interference suffered by HPC applications which share common physical machines and the number of physical machines used to allocate them. We propose a mathematical formulation for this problem and a strategy based on the Iterated Local Search framework to solve it. Experiments were conducted in a real scenario by using applications from oil and gas industry and applications from the HPCC benchmark. They showed that our method reduced interference in more than 40%, using the same number of physical machines of the most widely employed heuristics to solve the problem.*

## 1. Introduction

High Performance Computing (HPC) applications are typically executed in dedicated datacenters. However, in the past few years, cloud omputing has emerged as an attractive option to run these applications due to several advantages that it brings when compared with a dedicated infrastructure. Clouds provide a significant reduction in operational costs, besides offering a rapid provisioning of computing resources like virtual machines and storage. Moreover, with the recent increase of the number of cores in modern physical machines[1], HPC applications can scale up to many cores even when fully allocated to a single physical machine. Consequently, clouds can offer a reasonable service performance for small-scale HPC applications, that can be entirely placed into a single physical machine, thus avoiding the performance penalty usually imposed by the physical network of clouds [Gupta et al. 2013][Tomi et al. 2017].

[1]Recently, Intel® launched processor Xeon Platinum 8180M with 28 cores. Thus, a physical server that supports two of these processors, such as Gigabyte® MD61-SC2, provides 56 cores in total. Similarly, the server Supermicro® AS-1023US-TR4 provides 64 cores with two AMD® EPYC 7601. In addition, the SW26010 processor, a homegrown Chinese manycore chip, has 260 cores in total.

Despite these advantages, some challenges must be overcome to bridge the gap between performance provided by a dedicated infrastructure and the one supplied by clouds. Overhead introduced by virtualized layer [Chen et al. 2015] and hardware heterogeneity [Gupta et al. 2014], for example, affect negatively the performance of HPC applications when executed in clouds. In addition, the absence of a high-performance network can prevent synchronous and tightlycoupled HPC applications from being satisfactorily executed in this environment. Besides these problems, the performance of HPC applications can be particularly affected by resource sharing policies usually adopted by cloud providers. In general, one physical server can host many virtual machines holding distinct applications [Chen et al. 2016]. Although virtualization layer provides a reasonable level of resource isolation [Heiser 2008], some shared resources, like cache and main memory, cannot be sliced over all applications running in the virtual machines. As a consequence, these co-located applications can experience mutual interference, resulting in performance degradation [Gupta et al. 2013][Jersak and Ferreto 2016].

Some works, such as [Gupta et al. 2013], [Jin et al. 2015], [Yokoyama et al. 2017], [Basto 2015], [Jersak and Ferreto 2016], and [Chen et al. 2016] proposed Virtual Machine Placement (VMP) strategies to avoid or, at least, minimize the effect that interference imposes in co-located HPC applications. Some of those works used a static matrix of interference or proposed a naive procedure to determine it. Other works, even proposing more general methods to determine interference, considered just one kind of shared resource, the Shared Last Level Cache (SLLC), or adopted an approach that evaluates only general characteristics of HPC applications. However, as discussed in [Alves and de Assumpção Drummond 2017], all of these methods are not suitable for determining the interference because this problem is directly related to the amount and similarity of concurrent access to SLLC, DRAM (Dynamic Random Access Memory) and virtual network. Moreover, although some of those works, [Gupta et al. 2013] and [Jersak and Ferreto 2016], have also considered to reduce the number of used physical machines when trying to minimize interference, none of them treated both problems together.

In this work, we define the Interference-aware Virtual Machine Placement Problem for Small-scale HPC Applications in clouds (IVMPP), a multi-objective problem that aims to minimize, at the same time, (i) the interference experienced by small-scale HPC applications, i.e., HPC applications that are small enough to run on a multicore machine in a reasonable amount of time, and (ii) the number of physical machines used to allocate them. We present a mathematical formulation that formally defines the IVMPP, and introduces a strategy based on the Iterated Local Search (ILS) framework to solve it. To predict the interference level experienced by co-located applications, we use a quantitative and multivariate model originally presented in [Alves and de Assumpção Drummond 2017]. This model takes into account the amount of access to SLLC, DRAM, and virtual network, besides the similarity among application access profiles, to estimate interference with a reasonable prediction error. This model is a step forward for the cross-interference problem since it treats this problem from a multivariate and quantitative perspective. Unlike other strategies proposed in the related literature, this model evaluates the amount of access to shared resources instead of considering general and subjective characteristics of HPC applications. Besides that, the model assumes that the interference problem is influenced by more than one factor to deal with its multivariate nature.

Our proposed VMP strategy was evaluated by using a set of instances created from real-life HPC applications. We compared our strategy with the most widely used heuristics to minimize the number of machines in clouds and also with the Multi-Dimensional Online Bin Packing (MDOBP), a heuristic proposed in [Gupta et al. 2013], whose goal is to minimize the interference among co-located HPC applications.

This paper is organized as follows. Section 2 presents basic assumptions and definitions used throughout this paper. Section 3 describes the mathematical formulation for IVMPP, while Section 4 introduces our proposed VMP strategy to solve it. Section 5 presents experimental results and Section 6 describes conclusions and directions for future work.

## 2. Basic Assumptions and Definitions

In this work, we assume that the allocation of one application to a physical machine is equivalent to the allocation of all virtual machines, holding that application, to the same physical machine. As mentioned before, we used in our proposed VMP strategy the model introduced in [Alves and de Assumpção Drummond 2017] for predicting interference experienced by applications that share a common physical machine. This model relies on the amount of accumulated access to (i) SLLC, (ii) DRAM and (iii) virtual network[2]. This accumulated access, which is the sum of applications individual access, represents the total pressure which all co-located applications put in a given shared resource. Notice that an application individual access can be easily obtained by using performance monitoring tools.

From the applications accumulated access, the model estimates the level of interference experienced by the set of co-located applications. This interference level is defined as the average slowdown presented by all applications allocated to the same physical machine. In this paper, the *slowdown* of one application is particularly defined as the percentage of additional time spent by this application when it is executed concurrently with other ones. For example, suppose that the execution times of two applications, namely A1 and A2, when executed in a dedicated physical machine, were equal to 60 and 80 seconds, respectively. Suppose also that both applications, when executed concurrently in that physical machine, spent 100 seconds. In this case, the slowdown of applications A1 and A2 would be, respectively, 67% and 25%. This percentage represents how much additional time these applications need to complete their executions when allocated to the same physical machine. Thus, the interference level between both applications would be 46% which means that these two applications would experience, on average, 46% of mutual interference when allocated to the same physical machine.

## 3. Mathematical Formulation of IVMPP

Let $M$ be the set of physical machines available in a cloud environment and $A$ the set of applications to be allocated to it. We define the total amount of main memory and number of CPUs provided by each physical machine as $Mem$ and $Cpu$, respectively. Similarly, the amount of CPU and main memory required by an application $i \in A$ is defined as $m_i$

[2]In the context of this work, the amount of access to SLLC and DRAM are measured in terms of millions of references per second (MR/s), while the access to virtual network is expressed as the number of megabytes transmitted per second (MB/s).

and $c_i$, respectively. It is worth mentioning that, in this work, we assume that all physical machines have the same hardware configuration, i.e., they are equipped with the same amount of CPU and main memory.

Moreover, we define $\gamma_Q$ as the interference level experienced by a subset of applications $Q \subseteq A$ when allocated to a same physical machine and $\omega$ as the maximum interference level that can be reached on each physical machine. We now define a binary variable $X_{i,j}$ for each $i \in A$ and $j \in M$ such that $X_{i,j}$ is equal to 1 if and only if application $i$ is allocated to the physical machine $j$, and equal to 0, otherwise. In addition, we define a binary variable $Y_j$ for each $j \in M$ such that $Y_j$ is equal to 1 if and only if the physical machine $j$ is being used, i.e., if there is, at least, one application allocated to it. $Y_j$ is equal to 0 if no application is allocated to $j$.

This problem can be formulated as the integer programming problem described in Equations (1) to (8). The objective function defined in Expression 1 seeks for minimization of the sum of interference levels presented in each physical machine and the number of machines used to allocate all applications. Remark that the sum of interference levels, which is used to measure the overall interference of the cloud environment, was normalized with respect to the maximum sum of interference levels that can be reached in our environment ( $\omega . \sum_{j \in M} Y_j$).

Besides that, we normalized the number of used machines concerning the total number of physical machines available in the cloud environment. As both objectives were normalized between 0 and 1, we can use the weight $\alpha$ to determine the relevance of each objective. Therefore, when $\alpha$ is set to 1 the objective function prioritizes the minimizing of the sum of interference levels, and when $\alpha$ is close to 0 the function minimizes the number of used physical machines. Thus, this parameter can be adjusted to reach a desirable trade-off between both objectives.

$$min \left( \left\{ \frac{\sum_{j \in M} \left[ \sum_{Q \subseteq A} \left( \prod_{i \in Q} X_{i,j} \right) . \gamma_Q \right]}{\omega . \sum_{j \in M} Y_j} \right\} . \alpha + \left( \frac{\sum_{j \in M} Y_j}{|M|} \right) . (1 - \alpha) \right) \tag{1}$$

$$\sum_{j \in M} X_{i,j} = 1, \forall i \in A \tag{2}$$

$$\sum_{i \in A} X_{i,j} . m_i \leq Mem . Y_j, \forall j \in M \tag{3}$$

$$\sum_{i \in A} X_{i,j} . c_i \leq Cpu . Y_j, \forall j \in M \tag{4}$$

$$X_{i,j} \leq Y_j, \forall i \in A, \forall j \in M \tag{5}$$

$$Y_{j+1} \leq Y_j, \forall j = \{1...|M| - 1\} \tag{6}$$

$$X_{i,j} \in \{0, 1\}, \forall i \in A, \forall j \in M \tag{7}$$

$$Y_j \in \{0, 1\}, \forall j \in M \tag{8}$$

Constraints described in Equation 2 ensure that each application is entirely allocated to just one physical machine. Inequalities defined in (3) and (4) enforce that the total amount of CPU and main memory available in physical machines are not exceeded. In Inequalities (5) we defined constraints to guarantee that a physical machine is used if and only if it holds, at least, one single application. As all physical machines permutation yield feasible solutions, we added a set of constraints, defined in (6), that eliminates symmetry by establishing an order at which machines shall be used in the cloud environment. At last, constraints described in (7) and (8) define the binary and integrality requirements on the variables.

## 4. Multistart Iterated Local Search for the IVMPP Problem

The classic VMP problem is classified as an NP-Hard problem [Pires and Barán 2015]. So, IVMPP, that is a variant of VMP, can also be classified as an NP-Hard problem. For this sort of problem, exact procedures have often proven to be incapable of finding solutions in a reasonable time. Therefore, we propose in this work a solution based on the Iterated Local Search (ILS) metaheuristic which employs a multistart approach to explore larger areas of the search space.

In order to present the proposed algorithm, we need to introduce some additional notation. We define a placement solution $s = \{(a1, m1), (a2, m2), ...\}$ as the set of 2-tuples $(a, m)$ representing that the application $a$ is allocated to physical machine $m$. Moreover, we define $Z(s)$ as the normalized sum of interference levels predicted for solution $s$. As previously described, we used the prediction model originally proposed in [Alves and de Assumpção Drummond 2017] to calculate the interference level experienced by applications allocated to a single physical machine.

In addition, we define $M(s)$ as the normalized number of physical machines used in $s$, and $E_{CPU}(s)$ and $E_{MEM}(s)$ as the percentage of amount of CPU and main memory exceeded in solution $s$, respectively. A solution is considered feasible if and only if $E_{CPU}(s)$ and $E_{MEM}(s)$ are equal to zero. We define in Equation 9 a cost function $f : \mathbb{S} \rightarrow \mathbb{R}$, where $\mathbb{S}$ is the set of all possible solutions. Similarly to Equation 1, this function $f(s)$ attempts to minimize the sum of levels of interference and the number of used machines. In addition, $f(n)$ penalizes, in accordance with parameter $\lambda$, unfeasible solutions, which use more CPU and memory than the available amount in the environment.

Our proposed solution, called *ILSivmp*, is described in Algorithm 1. For each iteration of the *Multistart Loop* (lines 2 to 15), the algorithm executes the procedure *ConstructivePhase* (line 3) to create a new solution to be submitted to the ILS. Then, *ILS Loop* (lines 4 to 13) performs a local search in the neighborhood of the current solution through *VariableNeighborhoodDescent* method (line 5). This method executes three classical local search movements for VMP problem: *MOVE-1*, *MOVE-2* and *SWAP-1*. Heuristics *MOVE-1* and *MOVE-2*, move, respectively, one and two applications to another used physical machine, while *SWAP-1* performs one swap operation, i.e., it swaps two applications between two distinct physical machines. As we adopted the strategy of first improving, these local search procedures stop executing upon improving the current solution.

After this local search phase, the *ILSivmp* evaluates the quality of the current solution $\overline{s}$ (line 6). If $\overline{s}$ represents an improvement on the best current solution $s^*$, it is set as the

**Algorithm 1** *ILSivmp*

**Input:** $A$,$M$,$\beta$,$iterMaxILS$, $iterMaxMultiStart$
**Output:** $s^*$
1: $s^* = \emptyset$; $f(s^*) = \infty$; $i = 1$; $j = 1$;
*/*Multistart Loop*/*
2: **while** $i \leq iterMaxMultiStart$ **do**
3: $s = ConstructivePhase(A, M, \beta)$;
*/*ILS Loop*/*
4: **repeat**
5: $\overline{s} = VariableNeighborhoodDescent(s, A, M)$;
6: **if** $(f(\overline{s}) < f(s^*))$ **then**
7: $s^* = \overline{s}$;
8: $j = 1$;
9: **else**
10: $j = j + 1$;
11: **end if**
12: $s = Perturbation(\overline{s}, j)$;
13: **until** $j \leq iterMaxILS$
14: $i = i + 1$;
15: **end while**
16: **return** $s^*$

actual best solution and the counter $j$ is reset to 1 (lines 7 and 8). Otherwise, the counter $j$ is incremented (line 10). Then, the algorithm executes the procedure $Perturbation$ (line 12) that applies a perturbation on the current solution by executing $j$ random movements of each type (i.e., *MOVE-1*, *MOVE-2*, and *SWAP-1*) in such a way that the resulting modification is sufficient to escape from local optima. Furthermore, procedure $Perturbation$ also adds a new physical machine to the solution, before executing the last perturbation (i.e., when $j$ is equal to $iterMaxILS$) of the current solution.

$$f(s) = Z(s).\alpha + M(s).(1 - \alpha) + E_{CPU}(s).\lambda + E_{MEM}(s).\lambda \tag{9}$$

Algorithm 2 presents the *ConstructivePhase*, called in line 3 of Algorithm 1. Firstly, the algorithm sorts the list of applications in decreasing order (line 1) by considering the following sequence: (i) amount of accesses to SLLC, (ii) amount of requested CPU and main memory, and (iii) amount of accesses to virtual network and DRAM. Next, the algorithm inserts one physical machine into the set of used machines (lines 2 and 3). Then, the procedure executes *Allocation Loop* (lines 4 to 20) that basically performs the selection of two applications from the list of sorted applications and selection of a physical machine to allocate them. The process of selecting this pair of applications is described as follows. Firstly, the algorithm creates the sets of applications $First$ and $Last$ that are composed, respectively, of the first and last $N_a$ elements of the sorted list of applications (lines 5 to 7). This number of applications is defined by the parameter $\beta$ that is used to control the degree of randomness of this constructive phase. Next, the algorithm selects, randomly, applications $i$ and $j$ from $First$ and $Last$, respectively (line 8). By selecting applications from the head and tail of the sorted list, this process aims to co-locate applications with complementary access profiles and amount of requested

resources, thus resulting in a low interference level.

**Algorithm 2** *ConstructivePhase*

**Input:** $A$,$M$,$\beta$
**Output:** $s$
*App* = *Sort(A)*;
*Used* = *Used* $\cup \{m_1\}$;
$M = M \backslash \{m_1\}$;
*/*Allocation Loop*/*
**while** *App* $\neq \emptyset$ **do**
    $N_a = \lceil \beta . |App| \rceil$;
    *First* = first $N_a$ elements of *App*;
    *Last* = last $N_a$ elements of *App*;
    *Choose randomly applications* $i \in First$ *and* $j \in Last$;
    **if** There is $m \in Used$ with enough resources to allocate $i$ and $j$ **then**
        *Choose the best one to allocate* $i$ *and* $j$;
    **else if** $M \neq \emptyset$ **then**
        *Pick up any* $m \in M$
        *Used* = *Used* $\cup \{m\}$;
        $M = M \backslash \{m\}$;
    **else**
        *Choose randomly* $m \in$ *Used*;
    **end if**
    $s = s \cup \{(i, m), (j, m)\}$;
    *App* = *App*$\backslash \{i, j\}$;
**end while**
**return** $s$

After selecting this pair of applications, the best physical machine to allocate them is chosen among the set of physical machines with enough resources to allocate both applications. The method selects the best machine by considering two metrics. These metrics are obtained from the following vectors of resources (CPU and main memory): (i) remaining resources of the physical machine ($\overrightarrow{REM}$), (ii) requested resources ($\overrightarrow{REQ}$) and (iii) remaining resources after allocating requested resources ($\overrightarrow{RAF}$). Each of these vectors has two dimensions, corresponding to the normalized amount of CPU and main memory. The first metric, called *alignment factor*, was also used in [Gupta et al. 2013], and is calculated as the angle between vectors $\overrightarrow{REQ}$ and $\overrightarrow{REM}$. This factor is used to evaluate the shape of the exploitable volume of resources, allowing to assess whether required and remaining resources are complementary to each other. The other metric, called *residual factor*, is equal to the length of vector $\overrightarrow{RAF}$ and is used to assess the size of exploitable volume of resources, i.e, it measures the amount of resources left in the physical machine after the allocation of the requested resources. This second metric is proposed as a tiebreaker for the first metric, i.e., for cases with the same alignment factor, the residual factor is used to select the physical machine which will result in the smallest resource wastage.

After calculating those metrics, the *ConstructivePhase* chooses the physical ma-

chine with the smallest alignment and residual factors (in this order) to allocate applications $i$ and $j$ (line 10). However, if none of the used physical machines has available resources to allocate applications $i$ and $j$, the *ConstructivePhase* uses a new physical machine from $M$ to allocate them (lines 12 to 14). This approach aims to reduce the number of used machines because it takes new machines only when it is absolutely necessary. Moreover, if none of physical machines is capable of allocating that pair of applications, the method chooses randomly a physical machine to hold them (line 16). At last, both applications $i$ and $j$ are allocated to the selected physical machine and removed from the sorted list of applications (lines 18 and 19). Notice that unfeasible solutions can be generated by this constructive method. However, their corresponding costs are very high due to the penalty applied in the proposed cost function.

## 5. Experimental Tests and Results

Experimental evaluation of our proposal was conducted in two phases. At first, we compared our VMP strategy with some of the most employed heuristics to minimize the number of used machines: Best Fit (BF), First Fit (FF), Worst Fit (WF), Best Fit Decreasing (BFD), First Fit Decreasing (FFD) and Worst Fit Decreasing (WFD). Since these greedy algorithms have shown to be effective for solving this problem in previous works [Jersak and Ferreto 2016], they were also used as a baseline here, in our experiments. In this way, we could assess the ability of our strategy on reducing interference, while using a small number of physical machines.

After that, we compared our solution with the Multi-Dimensional Online Bin Packing (MDOBP) [Gupta et al. 2013], the closest VMP work related to our proposal, since it aims to minimize the interference among co-located HPC applications. All tests were executed in a Itautec MX214 server equipped with two Intel Xeon X5675 3.07GHz processors and 48GB of main memory. Moreover, all codes were compiled with GCC (Gnu C Compiler) version 4.9.2. All presented results are average of 10 execution.

To accomplish this experimental evaluation, we created a set of instances for the IVMPP problem by considering two real-life HPC-applications from petroleum industry: MUFITS (Multiphase Filtration Transport Simulator) [Otto and Kempka 2017] and PKTM (Pre-stack Kirchhoff Time Migration) [Melo Alves et al. 2017]. We have also considered applications available in HPCC[3], a widely used benchmark for evaluating HPC systems.

An instance of the IVMPP problem is composed by (i) a set of applications to be allocated in the cloud environment and (ii) a number of physical machines available in that environment. For each application, the amount of individual access to SLLC, DRAM and virtual network, and the normalized amount of requested CPU and main memory are given as input. We created 100 instances for the IVMPP problem, where the number of applications for each instance varied from 5 to 50. Those instances were generated by selecting some applications from the workload randomly. Notice that an application can appear more than once in an instance.

In addition, for all instances, the number of machines available in the cloud environment was set as the total number of applications to be allocated. From this approach,

[3]http://icl.cs.utk.edu/hpcc/.

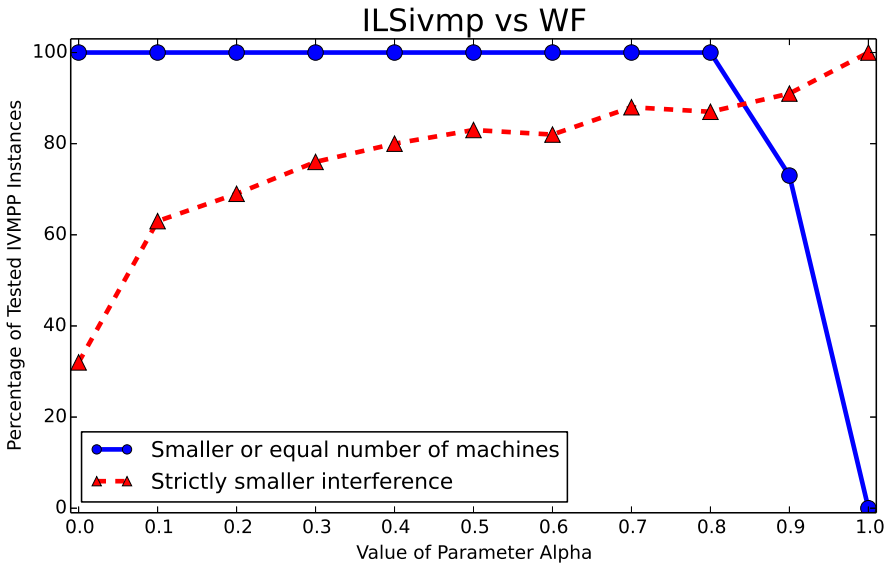



**Figure 1. ILSivmp against WF for minimizing the number of used physical machines**

we can evaluate whether IVMPP_ILS sacrifices the minimization of physical machines in favor of the minimization of interference. Indeed, by adopting this configuration, IVMPP_ILS has the option to spread out the applications, using all physical machines, in order to fully avoid interference in the cloud environment.

### 5.1. Comparing *ILSivmp* and Heuristics

To evaluate our proposal concerning its capacity to reduce interference while using a small number of physical machines, we devised the following two metrics. In the first one, we determined the percentage of test cases where our solution achieved a *strictly smaller sum of interference levels* than the one reached by the heuristic. Concerning the minimization of interference, this metric allows to quantify the number of test cases where our solution outperformed the tested heuristic. In the second metric, we calculated the percentage of cases where our proposal used a *smaller or equal number of physical machines* than the one used by the heuristic. By using this metric, we expect to evaluate the number of test cases where our solution achieved the minimal number of physical machines needed to allocate all applications in the cloud environment.

In our approach, the best trade-off between reducing interference and number of used machines was achieved when $\alpha$ was equal to 0.70. Result of the comparison between *ILSivmp* and WF is presented in Figure 1. The results of comparison with the other heuristics were similar. Our proposal has achieved a smaller or equal number of used machines in 100% of instances for all heuristics, besides reaching a strictly smaller interference in 88% of instances.

In order to verify the accuracy of the results achieved in those experiments, we conducted an additional evaluation in a real scenario by using the same set of physical machines previously described. We selected five instances where *ILSivmp* and the heuristics achieved the same number of physical machines. So, by allocating applications according to the placement given by *ILSivmp* and to the heuristic that achieved the smallest

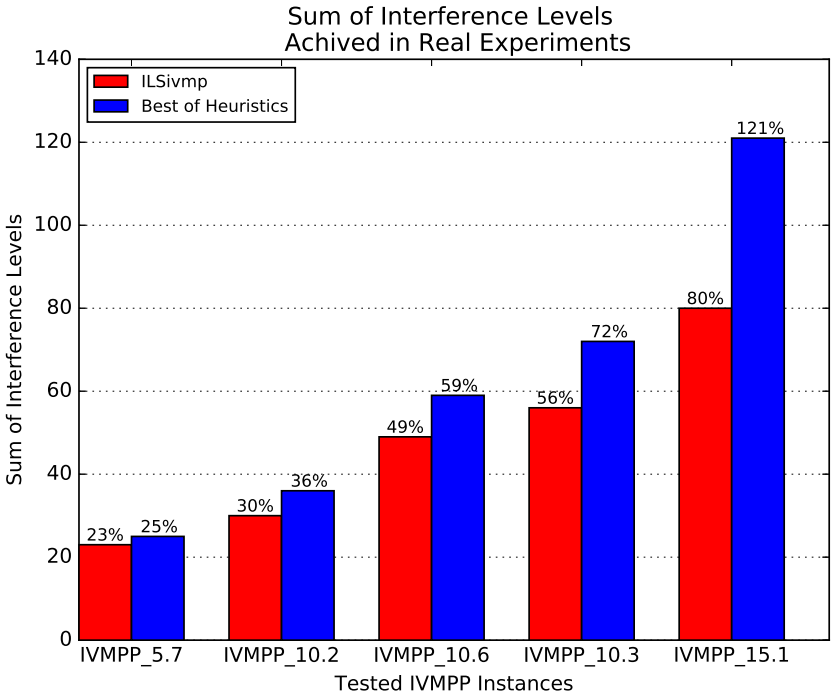



**Figure 2. Sum of interference levels achieved in a real scenario**

interference, we calculated the real sum of interference levels reached in practice.

Results show that our strategy, even using the same number of physical machines of the other heuristic, reduced the sum of interference levels by more than 40%, as presented in Table 2. Moreover, it is worth mentioning that the difference between the predicted sum of interference levels and the one achieved in real experiments was, in average, around 8%. Those results indicate that our proposal was able to reduce the sum of interference levels, while using a small, perhaps minimal, number of physical machines.

**Table 1. Comparing ILSivmp ($\alpha = 1.0$) with MDOBP.**

| Instance | ILSivmp $\alpha = 1.0$ | | | MDOBP | | Difference Between Interference Levels |
|---|---|---|---|---|---|---|
| | Sum of Interference Levels | std | Number of Used Hosts | Sum of Interference Levels | Number of Used Hosts | |
| IVMPP_50.1 | 20.48% | 0.01 | 38 | 24.05% | 38 | 3.57% |
| IVMPP_50.2 | 21.37% | 0.01 | 33 | 28.40% | 33 | 7.03% |
| IVMPP_50.3 | 25.15% | 0.01 | 33 | 37.07% | 33 | 11.92% |
| IVMPP_50.4 | 28.80% | 0.01 | 36 | 38.68% | 36 | 9.88% |
| IVMPP_50.5 | 28.00% | 0.02 | 36 | 33.67% | 36 | 5.67% |
| IVMPP_50.6 | 24.22% | 0.01 | 33 | 28.47% | 33 | 4.25% |
| IVMPP_50.7 | 27.24% | 0.01 | 33 | 34.42% | 33 | 7.18% |
| IVMPP_50.8 | 24.60% | 0.01 | 36 | 32.30% | 36 | 7.70% |
| IVMPP_50.9 | 20.64% | 0.01 | 34 | 30.67% | 34 | 10.03% |
| IVMPP_50.10 | 36.77% | 0.01 | 32 | 44.58% | 32 | 7.81% |

## 5.2. Comparing *ILSivmp* and MDOBP

Results described in the previous section showed that our solution was able to reduce interference even using a small number of physical machines. However, we acknowledge

that, unlike our proposal, those heuristics seeks only for the minimization of the number of used machines.

Therefore, to fairly evaluate our proposal, we compared it with a VMP strategy aware of interference and number of used physical machines. This VMP strategy, called MDOBP (Multidimensional Online Bin Packing), was proposed in [Gupta et al. 2013]. At first, it classifies each HPC application according to its communication pattern. Then, based on these classes, the algorithm decides where each virtual machine, holding an application, should be allocated. Tightly-coupled synchronous HPC applications, for example, are always allocated to a dedicated physical machine. On the other hand, for the remaining classes of HPC applications, the VMP strategy seeks for the minimization of the number of physical machines, also observing acceptable interference criteria. Thus, depending on the class of the HPC application, MDOBP focuses on fully avoiding interference or minimizing the number of used physical machines.

Unlike our proposal, MDOBP employs an online approach, i.e., it promptly allocates a virtual machine to a physical one upon its request. Consequently, the sequence (order) at which the applications appear in the instance influences the behavior of MDOBP. Our solution, on the other hand, has a global and complete view of the problem. To workaround this issue in the MDOBP case, we generate, for each IVMPP instance, 50.000 different sequences of application. Then, we selected the sequence at which MDOBP achieved the smallest sum of interference levels.

We tested both *ILSivmp* and MDOBP over a set of instances containing 50 applications, that is the highest number of applications in all instances. The number of physical machines in the instances was also 50. Concerning our proposal, the input parameter $\alpha$ was set to 1.0 so that *ILSivmp* prioritizes the minimization of interference.

As can be seen in Table 1, *ILSivmp* was able to outperform MDOBP in terms of interference in all tested instances, using the same number of physical machines. In fact, MDOBP just tries to fully avoid interference or attempts to keep interference within bounds when minimizing the number of physical machines, while *ILSivmp* treats both problems at the same time.

## 6. Conclusion and Future Work

In this work, we defined the Interference-aware Virtual Machine Placement Problem for Small-scale HPC applications (IVMPP), and presented a mathematical formulation and a strategy based on the Iterated Local Search (ILS) framework to solve it. Experiments conducted in a real scenario, with real HPC applications, showed that our proposal reduced interference by more than 40%, while keeping the same number of physical machines given by the most employed heuristics for the problem. Besides that, in comparison with MDOBP, the closest approach found in literature to solve VMP, *ILSivmp* reduced the interference levels in 7.51%.

We conclude that the performance of small-scale HPC applications executed in clouds can be improved when the virtual placement strategy considers complementary access profiles in co-allocation, minimizing consequently the cross- application interference. As future work, we intend to investigate an online strategy for the IVMPP problem. In order to propose such strategy, initially, we intend to determine in what extent the live migration of virtual machines affect the execution of small-scale HPC applications.

## References

Alves, M. M. and de Assumpção Drummond, L. M. (2017). A Multivariate and Quantitative Model for Predicting Cross-application Interference in Virtual Environments. *Journal of Systems and Software*, 128:150–163.

Basto, D. T. (2015). Interference Aware Scheduling for Cloud Computing. Master's thesis, Universidade do Porto.

Chen, L., Patel, S., Shen, H., and Zhou, Z. (2015). Profiling and Understanding Virtualization Overhead in Cloud. In *44th International Conference on Parallel Processing (ICPP)*, pages 31–40. IEEE.

Chen, L., Shen, H., and Platt, S. (2016). Cache contention aware virtual machine placement and migration in cloud datacenters. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE.

Gupta, A., Faraboschi, P., Gioachin, F., Kale, L. V., Kaufmann, R., Lee, B., March, V., Milojicic, D., and Suen, C. H. (2014). Evaluating and Improving the Performance and Scheduling of HPC Applications in Cloud. *IEEE Transactions on Cloud Computing*, 7161(c):1–1.

Gupta, A., Kale, L. V., Milojicic, D., Faraboschi, P., and Balle, S. M. (2013). HPC-aware VM Placement in Infrastructure Clouds. In *International Conference on Cloud Engineering (IC2E)*, pages 11–20. IEEE.

Heiser, G. (2008). The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pages 11–16. ACM.

Jersak, L. C. and Ferreto, T. (2016). Performance-aware Server Consolidation with Adjustable Interference Levels. In *Proceedings of the 31st Annual Symposium on Applied Computing*, pages 420–425. ACM.

Jin, H., Qin, H., Wu, S., and Guo, X. (2015). CCAP: A Cache Contention-aware Virtual Machine Placement Approach for HPC Cloud. *International Journal of Parallel Programming*, 43(3):403–420.

Melo Alves, M., da Cruz Pestana, R., Alves Prado da Silva, R., and Drummond, L. M. A. (2017). Accelerating pre-stack kirchhoff time migration by manual vectorization. *Concurrency and Computation: Practice and Experience*, 29(22):1–20.

Otto, C. and Kempka, T. (2017). Prediction of Steam Jacket Dynamics and Water Balances in Underground Coal Gasification. *Energies*, 10(6):739.

Pires, F. L. and Barán, B. (2015). A Virtual Machine Placement Taxonomy. In *15th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 159–168. IEEE/ACM.

Tomi, D., Car, Z., and Ogrizovi, D. (2017). Running hpc applications on many million cores cloud. In *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 209–214.

Yokoyama, D., Schulze, B., Kloh, H., Bandini, M., and Rebello, V. (2017). Affinity Aware Scheduling Model of Cluster Nodes in Private Clouds. *Journal of Network and Computer Applications*, 95:94–104.

# Otimização automática do custo de processamento de programas SPITS na AWS

**Nicholas T. Okita**[1], **Charles B. Rodamilans**[1,2], **Tiago A. Coimbra**[1], **Martin Tygel**[1] **e Edson Borin**[1]

[1]Centro de Estudos de Petróleo (CEPETRO)
Universidade Estadual de Campinas (UNICAMP) – Campinas – SP – Brasil

[2]Faculdade de Computação e Informática (FCI)
Universidade Presbiteriana Mackenzie (UPM) – São Paulo – SP – Brasil

nicholas.okita@ggaunicamp.com,

{charlesrodamilans, tgo.coimbra, mtygel}@gmail.com, edson@ic.unicamp.br

***Resumo.*** *Plataformas de serviço de computação em nuvem oferecem uma ampla variedade de recursos computacionais que possuem características de desempenho com custos diferenciados. Neste trabalho, investigamos como as instâncias Spot e as zonas de disponibilidade da Amazon Web Services (AWS) podem ser utilizadas para a redução do custo de processamento. Em adição, propomos um algoritmo de gerenciamento automático de instâncias na AWS para otimizar custo na execução de programas implementados sobre o modelo de programação Scalable Partially Idempotent Task System (SPITS). Os resultados obtidos indicam que o método proposto é capaz de identificar para ajustar dinamicamente os tipos de máquinas virtuais que oferecem o melhor custo-benefício.*

## 1. Introdução

O modelo de negócios de infraestrutura como serviço (*Infrastructure as a Service* - IaaS) é oferecido para os usuários pelos principais provedores de serviço em nuvem computacional, tais como, *Amazon Web Services* (AWS) da Amazon [Amazon 2018], *Azure* da Microsoft [Azure 2018] e Google Cloud Platform [Google 2018]. Neste modelo é permitido aos usuários a instanciação de máquinas virtuais com diferentes combinações de *hardware*, como número de núcleos de processamento e quantidade de memória RAM, e a configuração com sua própria pilha de *software*.

Por exemplo, na AWS existem três formas de se adquirir uma máquina virtual [Li et al. 2016]. Primeiro, instâncias *On-Demand*, na qual os consumidores pagam um custo fixo pela utilização do recurso por um pequeno período de tempo (e.g., horas ou segundos). Segundo, instâncias reservadas, nas quais os consumidores pagam uma taxa fixa para a utilização do recurso por um tempo longo (e.g., meses ou anos). Por último, instâncias *Spot*, as quais possuem preços voláteis de forma que os consumidores fazem lances de até quanto desejam pagar em um dado instante. O provedor de nuvem, baseado nesses lances, seleciona qual cliente poderá utilizar a instância. As três formas apresentam diferentes custos para diferentes instâncias, acarretando, portanto, na existência de um problema de otimização: a escolha da melhor configuração de *hardware* pelo menor custo.

Além do mais, o serviço da AWS é distribuído em várias regiões do mundo, e cada região possui múltiplas zonas de disponibilidade. Estas zonas são conectadas por *links* de baixa latência que permitem a comunicação dos processos entre si. Assim como existem diferentes custos relacionados a diferentes configurações de *hardware*, também existe diferença de custo para instâncias em diferentes zonas de disponibilidade e regiões.

Instâncias do tipo *Spot* apresentam menor custo devido ao risco de oscilação de preços causado pelo sistema de lances. Neste trabalho investigamos como minimizar a despesa de execução de programas de alto desempenho implementados com o modelo de programação *Scalable Partially Idempotent Task System* (SPITS) [Borin et al. 2016] em máquinas da nuvem computacional. O SPITS possui mecanismos de provisionamento dinâmico de recursos e tolerância a falhas, minimizando o risco de insdisponibilidade e permitindo sejam criadas novas instâncias do tipo *Spot*. Baseado nisso, desenvolvemos um método que se utiliza desses mecanismos para substituir instâncias de baixo custo-benefício. Através dos experimentos, vimos que o algoritmo foi capaz de ajustar dinamicamente a execução de um programa de processamento sísmico de alto desempenho de forma que apenas instâncias com alto custo-benefício fizessem parte do processamento.

## 2. Trabalhos Relacionados

Ao utilizar a nuvem computacional, o cliente está sujeito a alguns riscos, tais como: (a) risco da indisponibilidade da instância *Spot* para o usuário: este risco pode estar relacionado à falta de recursos ocioso por parte do provedor, o qual começa a recuperar instância *Spot* para ser utilizada como instância Reservada, *On-Demand* ou para uso próprio; (b) risco de indisponibilidade da região do provedor de nuvem computacional: isto pode ser devido a, por exemplo, falha na rede de comunição e, consequentemente, indisponibilidade de acesso à instância *Spot*; (c) risco de oscilação de preços: a alteração de preços pode tornar menos favorável utilizar uma instância com maior capacidade de processamento. O risco abordado neste trabalho está relacionado à oscilação de preços.

No passado, a AWS oferecia as instâncias *Spot* no formato de leilão [Agarwal et al. 2017], ou seja, o cliente que desse o maior lance tinha o direito de utilizá-la. Tal leilão ocorria periodicamente e a depender do provedor poderia ser a cada hora. Atualmente, nos provedores de nuvem (AWS, Azure e *Google Cloud*), o cliente informa o valor máximo que deseja pagar por hora, e o próprio provedor gerencia qual cliente utilizará a instância *Spot*. Quando o provedor necessita desta instância, ele envia um aviso para o cliente e, após um tempo prefixado, a instância *Spot* é terminada pelo provedor, causando o risco de indisponibilidade da instância *Spot*. Dessa forma, os lances excessivos que existiam no antigo modelo de leilão foram eliminados. Assim, os preços ficaram mais previsíveis e os métodos que tentam otimizar os lances [Zheng et al. 2015, Tang et al. 2012] não fazem mais sentido.

A ferramenta *HotSpot* [Shastri and Irwin 2017] busca reduzir o custo de processamento nas instâncias *Spot* movendo a aplicação para uma instância que oferece o menor custo naquele instante. Utiliza-se o custo da transação (deixar de utilizar a máquina e continuar pagando momentaneamente por ela) e a economia de custo esperada para se determinar quando deve haver a migração. Quando o preço de uma instância sobe, o *HotSpot* migra a aplicação para outra instância que possui o preço mais baixo. A migração da aplicação entre as instâncias é realizada utilizando *containers* e o *HotSpot* permite

trabalhar com instâncias do tipo *On-demand* e *Spot*.

O *EC2 Spot Fleet* [Fleet 2018] é um serviço da AWS para escalar a aplicação utilizando instâncias *Spot*. Na requisição, o usuário especifica o preço máximo desejado por instância por hora, a capacidade alvo e as regras de lançamento, tais como tipo de instância e zona de disponibilidade. Esta ferramenta automatiza os lances nas instância *Spot*, não ultrapassando o limite máximo de custo estipulado. Este serviço permite utilizar instâncias *On-demand* e não possui tolerância a falhas.

O *SpotOn* [Subramanya et al. 2015] seleciona e configura automaticamente instâncias *Spot* para executar a tarefa e também seleciona o mecanismo de tolerância a falhas (*checkpoint* ou replicação) mais adequado quando ocorre uma revogação da instância *Spot* por parte do provedor. Para realizar a troca de instância, é necessário que a revogação da instância *Spot* tenha ocorrido, isto é, o servidor solicitar a instância de volta.

Apesar das propostas anteriores otimizarem o custo com a seleção de tipos de instâncias, os critérios de escolha são baseados na utilização de recursos, como o uso de *CPU* e memória. A nossa proposta utiliza informações de desempenho do programa alvo (enviadas periodicamente, com frequência determinada pelo usuário) para escolher as instâncias que oferecem o menor custo-benefício e essas informações refletem de forma mais precisa o custo do processamento.

## 3. Materiais e Metodologia

O modelo de programação *SPITS* e o *runtime PY-PITS* [Borin et al. 2016] foram utilizados para a implementação de um programa de processamento sísmico de alto desempenho, o qual tem como finalidade a computação dos parâmetros do método *Non-hyperbolic Common Reflective Surface*, ou *NCRS* [Fomel and Kazinnik 2012]. Este programa utiliza a heurística *Differential Evolution* (*DE*) [Storn and Price 1997] para buscar os melhores parâmetros para o método.

O programa *NCRS* foi compilado utilizando o compilador *gcc* versão *5.4.0* com a *flag* de compilação *-O3*, enquanto o motor *PY-PITS* foi executado utilizando *python* versão *3.5.2*. A entrada do programa foi um dado sísmico de aproximadamente 1,3 GB e escolheu-se para o *DE* uma população de tamanho 31 iterando por 31 gerações.

Para os experimentos foram utilizadas as instâncias apresentadas na Tabela 1. A escolha foi feita baseada na execução do programa *NCRS* para uma parte do dado de aproximadamente 200 MB. Em cada instância foi montado um disco de 20 GB do tipo *Provisioned IOPS* (ou *io1*) com uma configuração de 1000 operações de entrada/saída por segundo (*Input/Output operations Per Second - IOPS*) e uma imagem de máquina virtual personalizada com o sistema operacional *Ubuntu* 16.04, a qual possuía alguns pacotes já instalados (*awscli*, *binutils*, *cloud-utils*, *efs-utils*, *gcc*, *make*, *python3-pip*, *sshpass*, *unzip* e *zip*) e uma cópia do dado utilizado. O programa e seus resultados foram armazenados no *Amazon Elastic File Storage* (EFS), o qual é um sistema de arquivos que pode ser compartilhado entre diferentes instâncias. Para computar o custo de execução, desprezamos, tanto o custo do *EFS*, quanto o custo do disco *io1*.

O algoritmo de gerenciamento foi implementado em *python 3.6* no serviço *Lambda* da AWS, o qual é responsável por iniciar e encerrar instâncias do tipo *Spot* baseado no custo benefício da instância. O custo-benefício é calculado pela razão do

**Tabela 1. Instâncias AWS selecionadas**

| Instância | vCPUS (número de núcleos virtuais) | RAM (GB) | Custo (USD/h) | Otimização |
|---|---|---|---|---|
| c4.4xlarge | 16(Xeon E5-2666 v3 Haswell) | 30 | 0,796 | Computação |
| c4.8xlarge | 32(Xeon E5-2666 v3 Haswell) | 60 | 1,591 | Computação |
| c5.4xlarge | 16(Xeon Platinum 8124 3GHz) | 32 | 0,680 | Computação |
| c5.9xlarge | 36(Xeon Platinum 8124 3GHz) | 72 | 1,530 | Computação |
| c5.18xlarge | 72(Xeon Platinum 8124 3GHz) | 144 | 3,060 | Computação |
| d2.4xlarge | 16(Xeon E5-2676 v3 Haswell) | 122 | 2,760 | Armazenamento |
| d2.8xlarge | 36(Xeon E5-2676 v3 Haswell) | 244 | 5,520 | Armazenamento |
| m4.4xlarge | 16(Xeon E5-2676 v3 Haswell) | 64 | 0,800 | Uso geral |
| m4.10xlarge | 40(Xeon E5-2676 v3 Haswell) | 160 | 2,000 | Uso geral |
| m5.4xlarge | 16(Xeon Platinum 8175 2.5GHz) | 64 | 0,768 | Uso geral |
| m5.12xlarge | 48(Xeon Platinum 8175 2.5GHz) | 192 | 2,304 | Uso geral |
| m5.24xlarge | 96(Xeon Platinum 8175 2.5GHz) | 384 | 4,608 | Uso geral |
| r4.4xlarge | 16(Xeon E5-2686 v4 Broadwell) | 122 | 1,064 | Memória |
| r4.8xlarge | 32(Xeon E5-2686 v4 Broadwell) | 244 | 2,128 | Memória |
| r4.16xlarge | 64(Xeon E5-2686 v4 Broadwell) | 488 | 4,256 | Memória |

desempenho da instância, em interpolações por segundo [Silva et al. 2016] pelo custo da instância, em dólares por hora, resultando em uma medida em interpolações por dólar [Okita et al. 2018]. Realizamos trocas de somente uma instância para detectar e corrigir anomalias nas escolhas futuras. Segue o pseudo código que cria uma nova instância do tipo com melhor custo benefício e encerra a instância que apresentou pior custo benefício.

**Algorithm 1** Algoritmo Gerenciamento

```
1: L: Lista de tuplas (id, tipo, custo benefício)   ▷ Em ordem cresc. de custo benefício
2: procedure SELEÇÃO DE INSTÂNCIA(L)
3:     if L[0].tipo ≠ L[len(L)].tipo then
4:         z ← zona de disponibilidade de menor custo para instância L[len(L)].tipo
5:         crie uma instância nova do tipo L[len(L)].tipo em z
6:         encerre a instância L[0].id
```

## 4. Resultados Experimentais

Aqui apresentamos uma avaliação do custo e do desempenho da aplicação em diferentes instâncias e mostramos os resultados atingidos com o algoritmo.

### 4.1. Instâncias *Spot*

Nesta subseção mostramos a fronteira de Pareto para o experimento. Em seguida, como as zonas de disponibilidade podem afetar o custo do processamento em nuvem.

#### 4.1.1. Fronteira eficiente

Executou-se um teste de desempenho nas instâncias mostradas na Tabela 1 seguindo as configurações descritas na Seção 3, porém, com uma seção de cerca de 15% do dado original. Este teste foi executado cinco vezes e as médias de tempo e de custo de cada instância são apresentadas na Figura 1. Além disso, com esses pontos foi traçada a fronteira Pareto, destacando-se as instâncias tais que nenhuma outra apresenta desempenho e custo total menores que a instância escolhida.

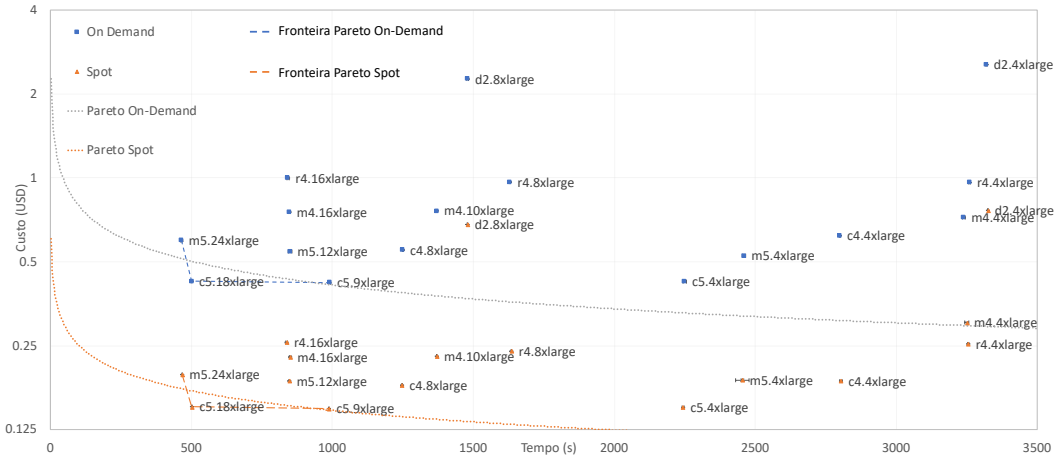



**Figura 1. Custo e tempo de execução de cada instância**

A Figura 1 apresenta as instâncias do tipo *On-Demand* como quadrados azuis e as instâncias do tipo *Spot* como triângulos laranjas. Observou-se que as instâncias *Spot* são cerca de três vezes menos custosas que suas equivalentes *On-Demand* para uma diferença de desempenho desprezável. Além disso, percebe-se que a má escolha de instâncias para o teste pode acarretar em custos e tempos de execução consideravelmente maiores, como por exemplo, ao comparar a instância *d2.4xlarge On-Demand* com a instância *c5.18xlarge Spot* vemos um tempo de execução cerca de sete vezes maior e um custo dezessete vezes maior.

A fronteira de Pareto foi composta por três tipos de máquinas virtuais tanto para instâncias *On-Demand* quanto para instâncias *Spot*, sendo elas consideradas as melhores escolhas para a execução do programa caso seja utilizada apenas uma instância. A utilização de mais instâncias resulta em alteração dessa fronteira, entretanto o foco desta seção é mostrar que instâncias *Spot* oferecem redução de custo, desde que o programa tenha mecanismos de tolerância a falhas capazes de lidar com a possibilidade de terminação por parte do provedor, a qual foi de fato verificada.

### 4.1.2. Zonas de disponibilidade

Durante a escolha das instâncias do tipo *Spot* é oferecida ao usuário a escolha da região e zona de disponibilidade. Como explicado na Seção 1, uma região é isolada de outra, tanto geograficamente quanto em questão de comunicação, enquanto zonas de disponibilidade são conectadas entre si por conexões de baixa latência. Portanto, fixando-se uma região, é possível executar os processos paralelos em múltiplas zonas de disponibilidade pois há comunicação entre instâncias de diferentes zonas.

Instâncias de diferentes zonas de disponibilidade possuem custos diferentes no mercado *Spot*, tal que isso torna a troca de zonas de disponibilidade uma possibilidade para a otimização de custo. A Figura 2 apresenta a variação de custo da instância *c5.18xlarge* em diferentes zonas em uma dada janela de tempo.

Como mostrado na Figura 2, uma boa escolha de zona de disponibilidade pode garantir uma redução de custo no lugar de uma escolha ingênua em qualquer zona de

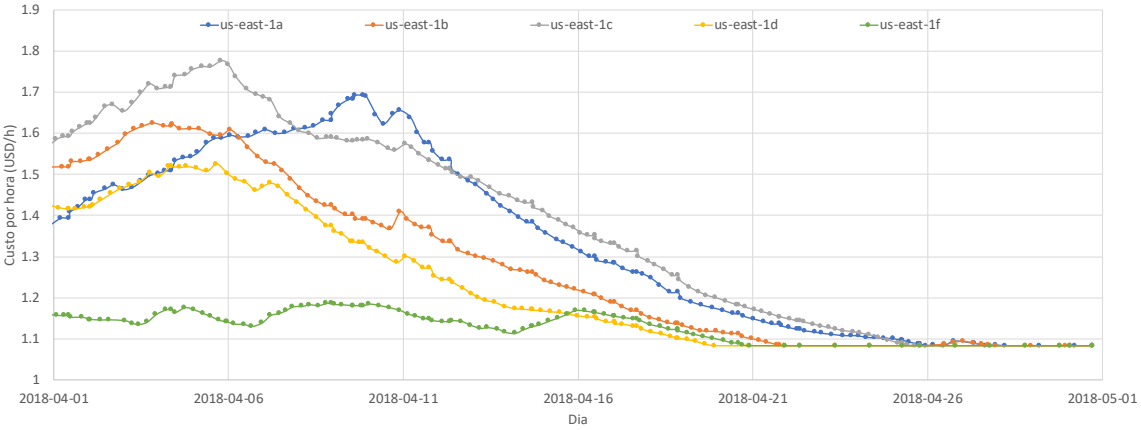



**Figura 2. Variação do custo da instância *c5.18xlarge* em diferentes zonas de disponibilidade**

disponibilidade. Por exemplo, no dia 6 de abril de 2018, a zona *us-east-1f* oferecia custo 50% inferior à zona *us-east-1c* para a mesma instância.

### 4.2. Algoritmo de gerenciamento de instâncias

Esta subseção avalia a capacidade do algoritmo de gerenciamento de instâncias, descrito na Seção 3, de reduzir o custo da execução de um programa sem ter conhecimento prévio de qual é o melhor tipo de instância para este programa.

Para esta avaliação foram realizados cinco experimentos, sendo dois de controle e três de validação do conceito, isto é, redução de custo a partir de informações dinâmicas geradas pelo programa. Nestes, fixou-se o número total de instâncias em 15 e somente poderiam ser criadas instâncias que estavam presentes no grupo inicial de máquinas. Além disso, as máquinas disponíveis foram limitadas às mostradas na Tabela 1.

O primeiro teste de controle envolvia a execução do programa utilizando todas as instâncias mostradas na Tabela 1. Consideramos que este é o pior caso de execução, onde o usuário desconhece as características do programa e das instâncias e simplesmente executa com todas as instâncias disponíveis. O custo deste teste está representado pela linha tracejada verde (controle max) na Figura 3. O segundo teste de controle envolvia o conhecimento prévio de quais são as melhores instâncias no quesito custo, baseado nos experimentos de validação e convergência para estas instâncias, que serão descritos com mais detalhes nas próximas subseções. De posse desta informação, o usuário instanciou somente máquinas com esse tipo de instância para obter o menor custo possível para um número fixo de 15 instâncias. Consideramos que esta era a melhor situação e está representada pela linha tracejada amarela.

Os testes de validação do conceito de gerenciamento automático das instâncias partiam novamente do pressuposto de desconhecimento das circunstâncias do problema. Dessa forma, criou-se máquinas virtuais de todos tipos, assim como no teste de controle do pior caso. Foram definidos três intervalos de tempos diferentes para trocas de instâncias, ou seja, quanto tempo de execução seria realizado antes do *script* no *Lambda* atuar para encerrar a instância com pior custo e criar uma nova instância com melhor custo de processamento. Os intervalos de tempo foram decididos em (a) um minuto; (b) três minutos; e (c) cinco minutos.

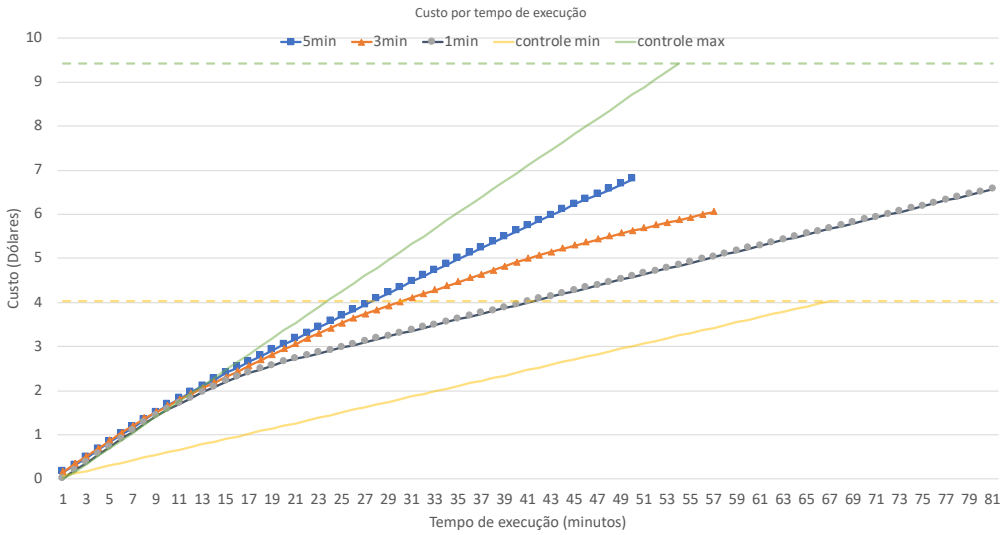



**Figura 3. Comparação de custo para diferentes intervalos de tempo entre trocas**

A Figura 3 mostra o custo total no decorrer do tempo de execução para as três situações de validação de conceito, o eixo das abscissas representa o tempo em minutos de execução e o eixo das ordenadas representa o custo total até o momento, além disso as linhas tracejadas indicam os custos das piores e melhores possibilidades de execução, assim como explicado em parágrafos anteriores. Conforme as expectativas, o algoritmo de troca de instâncias apresentou um custo consideravelmente inferior ao máximo e superior ao limitante inferior.

Na Figura 3 é possível notar que a troca de instâncias em intervalos de 1 minuto apresentou o pior tempo (81 minutos) dentre as três configurações. Além disso, atingiu um custo total no final da execução (de 6,6 dólares) semelhante ao custo da troca de instâncias em intervalos de 5 minutos (de 6,8 dólares). Por fim, o menor custo encontrado foi o da execução com a troca de instâncias em intervalos de 3 minutos (6 dólares).

### 4.2.1. Intervalos de três minutos

Nesta subseção serão discutidos os resultados obtidos na execução do teste com o programa NCRS utilizando o algoritmo de gerenciamento para troca de instâncias em intervalos de três minutos. Este foi o primeiro teste de validação; o tempo de 3 minutos foi escolhido devido ao tempo de inicialização da instância, somado com o tempo de leitura do dado de entrada e processamento da primeira tarefa ser cerca de três minutos.

Como observado na Figura 3, este intervalo de tempo foi o que proveu menor custo total ao final da execução dentre os demais testes de validação de conceito.

A Figura 4 apresenta o momento de inicialização e término das instâncias utilizadas na execução do programa. As instâncias do tipo *c5.4xlarge* apresentaram melhor custo benefício no decorrer da execução.

Na Figura 4 é perceptível que instâncias que não são otimizadas para a tarefa de computação são as primeiras a serem terminadas, enquanto instâncias otimizadas são

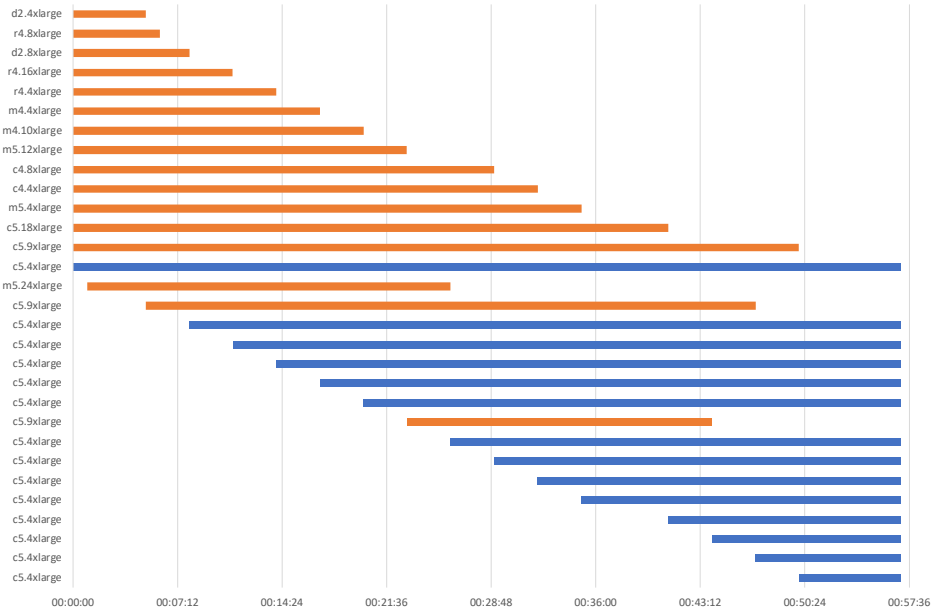



**Figura 4. Duração da execução das instâncias em trocas de três minutos, têm-se as instâncias do tipo *c5.4xlarge* em azul e as demais em laranja.**

sempre iniciadas. Por exemplo, as instâncias da família *d2* foram rapidamente trocadas por instâncias da família *c5*. A execução durou cerca de 50 minutos e possuiu um custo total de cerca de 6,05 dólares.

Esta execução apresentou custo entre os limitantes inferior e superior definidos e tempo de execução também entre ambos. Porém, neste teste a AWS encerrou uma das instâncias e, como o algoritmo de troca de instâncias implementado não contempla este tipo de interrupção, o teste finalizou com apenas quatorze instâncias. Apesar disto, como com quinze instâncias a expectativa era de uma execução em menor tempo, acreditamos que o custo total final seria semelhante ao obtido (cerca de 6 dólares), sendo ele o menor dentre as situações de controle. Também é importante destacar que, apesar da AWS ter finalizado uma das instâncias sendo utilizadas, o mecanismo de tolerância a falhas do SPITS permitiu que a aplicação finalizasse a execução corretamente.

### 4.2.2. Intervalos de um minuto

Esta subseção apresenta resultados para uma execução idêntica à anterior, salvo o intervalo de tempo entre troca de instâncias, o qual foi reduzido para um minuto. A motivação para a execução com este intervalo de tempo era a comparação com os resultados obtidos com três minutos de intervalo. Esperava-se que, com um intervalo reduzido, o algoritmo de escolhas convergiria mais rapidamente para a instância com custo benefício ótimo e apresentaria menor custo. Entretanto, como visto na Figura 3, obteve-se um custo e tempo de execução superiores ao da troca de instância em intervalos de três minutos.

Assim como a subseção 4.2.1, o gráfico apresentado na Figura 5 representa o tempo de inicialização e encerramento de cada instância. O tipo de instância que apresentou o melhor custo benefício foi novamente o tipo *c5.4xlarge*.

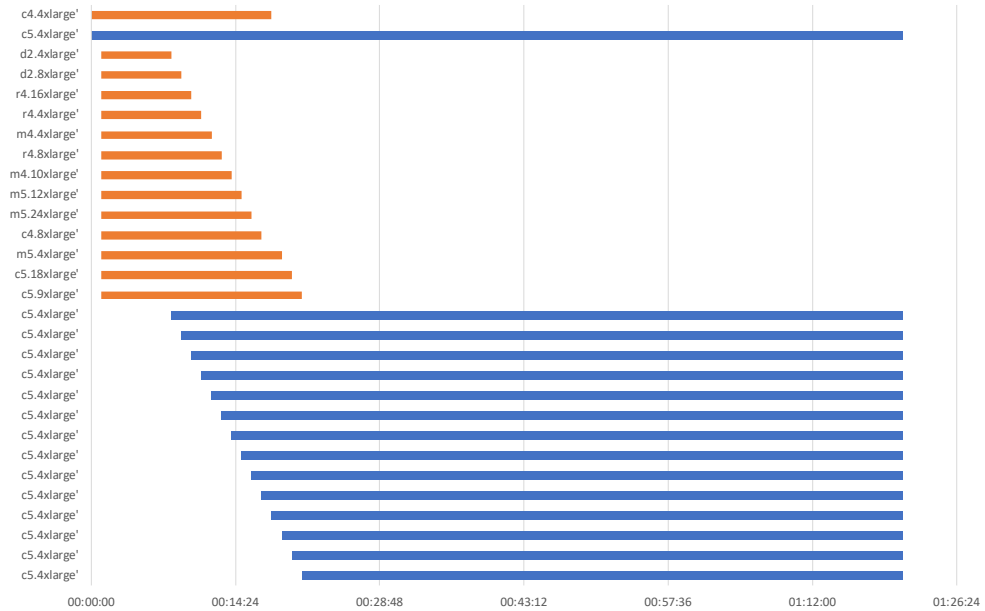



**Figura 5. Duração da execução das instâncias em trocas de um minuto, têm-se as instâncias do tipo *c5.4xlarge* em azul e as demais em laranja.**

Conforme observado na Figura 5, de fato houve uma rápida troca de todas as instâncias para instâncias do tipo *c5.4xlarge*. O custo total do experimento (cerca de 6,6 dólares) foi superior ao custo do experimento de controle utilizando apenas instâncias *c5.4xlarge* (aproximadamente 4 dólares) e também ao custo do experimento que utilizou intervalo de trocas de 3 minutos (cerca de 6 dólares). O problema deste teste está relacionado ao seu longo tempo de execução, o qual foi superior inclusive ao controle de custo ótimo, consequentemente carregando consigo um custo total superior.

Detectou-se que a perda de desempenho ocorreu devido ao tempo que instâncias novas demoravam para iniciar o processamento das tarefas. Apesar do tempo de inicialização dos *workers* ter sido constante entre instâncias (tempo de *boot* e tempo de leitura do dado), o gerenciador de tarefas do SPITS levou até 15 minutos para enviar tarefas às novas instâncias. Nossas investigações ainda não revelaram a causa deste problema.

### 4.2.3. Intervalos de cinco minutos

Nesta subseção realizamos o mesmo teste das subseções 4.2.1 e 4.2.2, salvo o intervalo de tempo para troca de instâncias, o qual foi configurado para cinco minutos.

A motivação para este teste era permitir que instâncias com melhor desempenho executassem por períodos mais longos, haja vista que o teste das subseções anteriores mostraram que a instância com melhor custo benefício possui desempenho inferior às demais, salvo algumas exceções. A expectativa era de redução do tempo total de execução, entretanto apresentaria uma melhoria menor em custo.

O gráfico da Figura 6 mostra os momentos em que as instâncias foram criadas e encerradas. As instâncias do tipo *c5.4xlarge* apresentaram o melhor custo benefício dentre as opções.

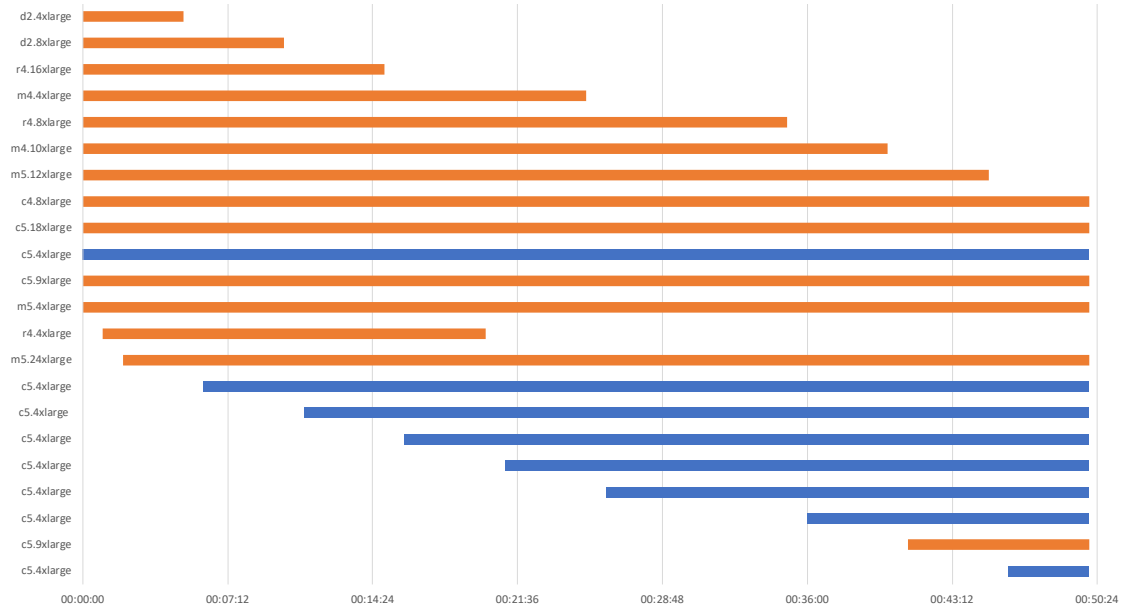



**Figura 6. Duração da execução das instâncias em trocas de cinco minutos, têm-se as instâncias do tipo *c5.4xlarge* em azul e as demais em laranja.**

É perceptível que, durante a execução do teste, várias instâncias subótimas não foram substituídas, como exemplo a instância do tipo *c5.9xlarge*. Isso ocorreu devido ao intervalo de tempo para troca de instância ser demasiado longo para o tempo total de testes; seriam necessários pelo menos 70 minutos para encerrar todas as instâncias enquanto o teste executou em cerca de 50 minutos. Observa-se que o programa estava convergindo para as instâncias do tipo *c5.4xlarge*, portanto independentemente do tempo de intervalo de troca as instâncias deste tipo mostraram custo superior às demais.

A situação com o intervalo de 5 minutos para troca de instâncias de fato apresentou o menor tempo de execução com relação aos outros testes, entretanto, devido ao maior custo das instâncias como *c5.18xlarge*, o custo total final foi cerca de 6,80 dólares, o maior dentre os testes de validação, como visto na Figura 3.

Com relação às situações de controle, o custo obtido foi dentro do esperado, isto é, superior ao custo ótimo (custo obtido realizando o teste em 15 instâncias *c5.4xlarge*) e inferior ao custo obtido utilizando as 15 instâncias da Tabela 1 sem nenhuma lógica. Por outro lado, o tempo de execução foi inferior ao tempo de execução de ambas situações de controle, isto é, apesar das perdas de desempenho causadas pela troca de instâncias, o algoritmo de gerenciamento foi capaz de entregar o resultado em tempo menor que a execução de todas as instâncias indiscriminadamente, no qual não ocorre essa perda. O motivo para para esse tempo inferior, foi a troca de instâncias mais lentas por instâncias mais rápidas, como a troca da instância *d2.4xlarge* pela *c5.4xlarge*.

O custo obtido com o algoritmo de gerenciamento automático foi em torno de 2 a 3 dólares maior do que o custo ótimo. Este custo está associado ao uso de máquinas ineficientes no início da computação e à execução do processo de inicialização em mais máquinas, 29, em vez de 15. Neste experimento, em particular, o tempo de inicialização (em torno de 3 minutos) das máquinas que foram descartadas tem um papel importante nesta diferença e contribui com cerca de 2 dólares. É importante salientar que esta

diferença é constante, dado que uma vez que o algoritmo trocou todas as instâncias por instâncias melhores, o desempenho/custo do restante do processamento será equivalente ao desempenho/custo da configuração ótima. Dessa forma, para processamentos longos, com várias horas, o custo e o desempenho tende a ser muito próximo do custo e do desempenho da execução com o melhor tipo de instância.

## 5. Conclusões

O serviço de nuvem computacional oferece uma grande variedade de configurações de máquinas virtuais para os usuários, permitindo a execução de código de alto desempenho com grande flexibilidade. Entretanto, as escolhas de configurações de máquinas virtuais podem acarretar em diferenças de desempenho e de custo.

Neste trabalho, executamos um código de processamento sísmico de alto desempenho em várias instâncias de máquinas virtuais da AWS, sejam individualmente, isto é, com apenas uma máquina na execução, ou em grupos de 15 instâncias, tendo como foco observar a relação custo versus desempenho e investigar como gerenciar as instâncias dinamicamente para otimização do custo.

Nos testes com apenas uma instância por execução observou-se que a escolha de uma boa instância garante um desempenho significantemente superior para um custo inferior, especialmente quando utilizamos instâncias *Spot*. Por exemplo, em nossos experimentos, a instância *d2.4xlarge On-Demand* executou o processamento em um tempo cerca de sete vezes maior e um custo dezessete vezes superior ao da instância *c5.18xlarge Spot*. Além do mais, com a variação de custo entre zonas de disponibilidade no mercado *Spot*, podemos obter maiores reduções a partir de boas escolhas de zonas.

Por fim, implementamos e avaliamos um algoritmo de gerenciamento automático de instâncias *Spot* que faz uso do desempenho instantâneo da aplicação. Este algoritmo tem como foco otimizar o custo de execução e não necessita de nenhuma informação prévia sobre o programa ou as instâncias. O algoritmo explora a variação de custo em zonas de disponibilidade na escolha de instâncias, sempre optando pela zona de menor custo, e foi capaz de convergir para o melhor tipo de instância e obter custos inferiores ao de uma execução ingênua com todas as instâncias. Ademais, mostramos que, na forma como foi implementado, o algoritmo apresenta diferenças, tanto em desempenho, quanto em custo para diferentes intervalos de tempo para trocas de instâncias.

A maior limitação do trabalho é a instrumentação do código para permitir obter as medidas de desempenho durante a execução, haja vista que essas medidas possibilitam inferir quais as melhores instâncias para nosso tipo de programa. Com isso, o desenvolvedor precisa modificar seu código para utilizar o algoritmo apresentado neste trabalho.

Esses resultados mostram que um bom algoritmo de gerenciamento pode de fato reduzir o custo de execução de um programa de alto desempenho na nuvem computacional, dado que este programa apresente possibilidade de migração entre máquinas e zonas de disponibilidade (provisionamento dinâmico de recursos e tolerâncias a falhas). Neste trabalho, uma solução de redução desses custos foi proposta através da utilização do modelo *SPITS*, com resultados encorajadores. No futuro, planejamos investigar as causas do problema de desempenho com intervalos de 1 minuto e refinar o algoritmo, otimizando o intervalo de tempo para troca de instâncias, mudanças nas políticas de trocas (tanto

em número de instâncias quanto seleção de tipos) e por fim execução de testes em maior escala.

## Agradecimentos

Os autores agradecem à Petrobras, à Fapesp, ao CNPq e à CAPES pelo apoio financeiro. Os autores também agradecem às equipes do laboratórios *High Performance Geophysics* (HPG) e LMCAD pelo suporte computacional.

## Referências

Agarwal, S. et al. (2017). Forecasting price of amazon spot instances using neural networks. *International Journal of Applied Engineering Research*, 12(20).

Amazon (2018). Amazon web services (aws). `https://aws.amazon.com`. Acessado em 20/07/2018.

Azure (2018). Microsoft azure. `https://azure.microsoft.com/`. Acessado em 20/07/2018.

Borin, E. et al. (2016). Py-pits: A scalable python runtime system for the computation of partially idempotent tasks. In *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*.

Fleet, A. E. S. (2018). How spot fleet works. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-fleet.html`. Acessado em 20/07/2018.

Fomel, S. and Kazinnik, R. (2012). Non-hyperbolic common reflection surface. *Geophysical Prospecting*, 61(1).

Google (2018). Google cloud. `https://cloud.google.com`. Acessado em 20/07/2018.

Li, Z. et al. (2016). Spot pricing in the cloud ecosystem: A comparative investigation. *Journal of Systems and Software*, 114.

Okita, N., Coimbra, T. A., and Borin, E. (2018). Análise de custo da nuvem computacional para a execução de algoritmos no processamento sísmico. In *ERAD-SP 2018*.

Shastri, S. and Irwin, D. (2017). Hotspot: automated server hopping in cloud spot markets. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM.

Silva, H. C. D., Pisani, F., and Borin, E. (2016). A comparative study of sycl, opencl, and openmp. In *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*.

Storn, R. and Price, K. (1997). Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4).

Subramanya, S. et al. (2015). Spoton: a batch computing service for the spot market. In *Proceedings of the sixth ACM symposium on cloud computing*. ACM.

Tang, S. et al. (2012). Towards optimal bidding strategy for amazon ec2 cloud spot instance. In *Cloud Computing, 2012 IEEE 5th International Conference on*. IEEE.

Zheng, L. et al. (2015). How to bid the cloud. In *ACM SIGCOMM Computer Communication Review*, volume 45. ACM.

# MapReduce with Components for Processing Big Graphs

**Cenez Araújo de Rezende**[1], **Francisco Heron de Carvalho Junior**[1]

[1]Pós-Graduação em Ciência da Computação (MDCC)
Universidade Federal do Ceará (UFC)
Campus Universitário do Pici, Bloco 912 – Fortaleza – CE – Brazil

{cenezaraujo,heron}@lia.ufc.br

***Abstract.*** *BigGraph applications have been mostly implemented with large-scale parallel processing frameworks based on MapReduce. However, it continues to be challenging to meet their particular requirements, even despite the emerging of alternative models, such as Pregel. This paper introduces a component-oriented design for graph processing frameworks, by using HPC Shelf, a component-based cloud computing platform for HPC services.*

## 1. Introduction

Big Data applications have origins in computer-aided analysis of large datasets, mostly collected from social networks and computational sciences. They are now important driving forces for the evolution of HPC technologies. Often, they demand for a huge amount of processing power that is not offered even by high-end parallel computing platforms, requiring the use of large-scale heterogeneous parallel computing resources.

Graphs are common data structures for representing Big Data workloads, making the design, development and evaluation of Big Graph processing frameworks a hot research topic in large-scale parallel processing [Lumsdaine et al. 2007].

MapReduce is a parallel processing model proposed by *Google Inc.* [Dean and Ghemawat 2008]. It has motivated the development of a number of large-scale processing frameworks [Li et al. 2016]. Some of them are concerned in attending particular Big Graph requirements, such as Pregel [Malewicz et al. 2010], originating new graph-processing frameworks, such as MR-MPI [Plimpton and Devine 2011], GraphX [Xin et al. 2013], GraphLab [Low et al. 2012], Giraph [Han and Daudjee 2015], among others [Doekemeijer and Varbanescu 2014, McCune et al. 2015].

CBHPC (Component-Based High Performance Computing) aims at applying component-based software engineering (CBSE) technologies for attending the scale and complexity requirements of modern HPC applications [Sarkar et al. 2004, Post and Votta 2005, van der Steen 2006]. A number of works have evaluated the use of CCA and Fractal, two HPC oriented component models, to realistic HPC software. In such a context, we have proposed a general model of parallel components, so-called Hash. Recently, we have introduced HPC Shelf, a new large-scale parallel processing platform that aims at offering HPC services through cloud computing abstractions, by using Hash components as an abstraction to represent parallel computing resources.

This paper reports two contributions. Firstly, a component-oriented MapReduce framework on top of HPC Shelf, aimed at providing large-scale parallel processing services through the service model of clouds. Secondly, a graph processing framework de-

rived from MapReduce, called Gust, supporting other existing graph-oriented parallel processing models, such as Pregel.

Section 2 briefly introduces HPC Shelf, MapReduce and the state-of-the-art in Big Graph frameworks. Section 3 presents a MapReduce framework for HPC Shelf, which is extended, in Section 4, for addressing Big Graph requirements, introducing Gust. In Section 5, the performance of the Gust prototype is compared to existing *state-of-the-art* alternatives, Giraph and GraphX. Finally, Section 6 concludes the paper, highlighting its contributions and pointing out at further works.

## 2. Background

In what follows, HPC Shelf is described. Then, it is presented the state-of-the-art in large-scale parallel processing systems for big graphs, starting with MapReduce.

### 2.1. HPC Shelf

HPC Shelf is a component-oriented platform for building cloud computing applications that provide, to *specialist users*, seamless access to HPC services over a large-scale parallel computing infrastructure comprising a set of clusters.

The components of HPC Shelf comply with Hash, a parallel component model motivated by the lack of a fully expressive model of parallel components in existing CBHPC platforms [Carvalho Junior and Rezende 2013]. The following component kinds are supported: *virtual platforms*, representing clusters; *computations*, representing parallel algorithm implementations that exploit the features of virtual platforms; *data sources*, for accessing data that interest to computations; *connectors*, which couple computations and data sources placed at distinct virtual platforms; *service bindings*, for client/server communication between components; and *action bindings*, for synchronization of computational actions among a set of components.

A connector comprises a set of *facets*, each one placed in one of the virtual platforms where the components it couples are placed. It may perform the following roles, possibly simultaneously: the orchestration of a set of computation components through action bindings; the support, as a coordination medium, for choreographies among computations and data sources through service bindings.

A service binding connects a *provider* to a *user port* of two components. A user and a provider port are compatible if there is a binding to connect them. If they have distinct interfaces, the service binding makes the role of an interface adapter.

An action binding connects a set of action ports of distinct computations and connectors having the same set of *action names*. Through an action binding, components may synchronize computations by activating actions and waiting for their completion, which occur when all the connected ports have a pending activation for the same action name.

A *parallel computing system* is an ensemble of components and bindings of the above kinds, including two special components: *Application* and *Workflow*. The other components are called *solution components*. *Application* represents the application itself, aimed at communication of solution components with the specialist user. For that, it may have service ports that are connected to compatible service ports of other components. *Workflow* is a special connector that drives computation and connector components

through a workflow execution. For that, it will have a set of action ports that will be connected to compatible action ports of computations and connectors.

HPC Shelf targets specialist users interested in solving problems described using a high-level interface offered by an application. For that, it recognizes three other kinds of users: *providers*, which develop applications that generate parallel computing systems; *developers*, which develop components that may explore the features of a class of virtual platforms; and *maintainers*, which control the instantiation of virtual platforms over a certain parallel computing infrastructure.

HPC Shelf comprises three architectural elements: *Front-End*, *Core* and *Back-End*. The *Front-End* is SAFe (Shelf Application Framework), a SWfMS (Scientific Workflow Management System) from which applications may be derived [Silva and Carvalho Junior 2016]. The *Core* services a library of components that are used by applications to build parallel computing systems. Also, it supports a system of *contextual contracts* for component selection according to a context, including application requirements and platform features [Carvalho Junior et al. 2016]. Finally, a *Back-End* manages a parallel computing infrastructure where virtual platforms may be instantiated, possibly using virtualization technologies.

Parallel computing systems are dynamically generated by applications for solving problems described by specialist users through their high-level interfaces. They are described using SAFeSWL (SAFe Scientific Workflow Language), a workflow description language with an architectural subset, for describing the architecture of paralell computing systems, and an orchestration subset, for specifying an activation flow for the actions of their computations and connectors.

### 2.2. MapReduce and Big Graph Processing

MapReduce is a large-scale parallel processing model introduced by *Google* [Dean and Ghemawat 2008], implemented by many parallel processing frameworks [Li et al. 2016], such as Hadoop [Foundation 2008]. A number of frameworks have extended MapReduce model for increasing its expressiveness and implementing efficiently certain algorithms [Elnikety et al. 2011][Plimpton and Devine 2011][Bu et al. 2012]. In such a context, special attention has been deserved to graph algorithms.

Li et al. [Li et al. 2016] have discussed the shortcomings of current MapReduce solutions, the following which are of special interest in our work: inefficient scheduling mechanisms for heterogeneous clusters; synchronization barrier between mapping and reduction; poor expressiveness to deal with iterative algorithms; inappropriateness in supporting real-time applications; no support of large-scale parallelism with many clusters.

A number of research efforts have addressed issues in implementing large-scale parallel graph algorithms using MapReduce [Doekemeijer and Varbanescu 2014, McCune et al. 2015, Li et al. 2016]. The resulting system proposals are very influenced by the "vertex-centric" paradigm of Pregel [Malewicz et al. 2010].

Pregel is based on BSP (Bulk-Synchronous Parallel) [Valiant 1990], by combining computational processes, routing to deliver messages and a global synchronization step. In Pregel computations, vertices are the central programming objects. The alternative to the "vertex-centric" model is the "graph-centric" one [Tian et al. 2013,

Aridhi et al. 2016], focused on subgraphs of the whole graph.

Giraph[Foundation 2011] is an open source Java implementation of Pregel that employs a "graph-centric" model. It is a reference for other frameworks, such as Giraph++ and GiraphUC[Han and Daudjee 2015]. GiraphUC implements an asynchronous variation of BSP so-called *Barrierless Asynchronous Parallel*.

Dryad[Isard et al. 2007] is a general-purpose alternative for data parallelism. For that, it allows the combination of computational vertices with communication channels, aiming at the formation of acyclic data flows. In this way, developers can describe communication flows as acyclic directed graphs.

GPS (Graph Processing System)[Salihoglu and Widom 2013] introduces mechanisms for dynamic graph partitioning and repartitioning across a set of compute nodes, concerned with dynamic reduction of communication costs among them. In turn, GraphLab[Low et al. 2012] has introduced a graph-based data consistency model representing data, vertices, edges and computational dependencies.

PowerGraph[Gonzalez et al. 2012] is a vertex-centric framework that has introduced the GAS model, with three phases of vertex computing. In **G**ather, information about adjacent vertices and edges are collected for sum. In **A**pply, the **G**ather result is applied, and a central vertex is updated, since vertices are partitioned (vertex-cut) in several mirrors and one central vertex. In **S**catter, the current vertex value is used to update adjacent edges. PowerGraph employs the data consistency model of GraphLab.

Spark [Zaharia et al. 2010] has introduced RDD (Resilient Distributed Dataset), which represents a collection of read-only Scala objects partitioned among a set of machines. RDD is fault-tolerant through *lineage*, also known as RDD dependency graph, which allows retrieving information about how an RDD was derived, making object rebuilds possible. Users may cache their RDD for use in MapReduce operations.

On top of Spark, GraphX [Xin et al. 2013] has implemented PowerGraph and Pregel abstractions. It provides a system of RDD that distribute immutable graphs as tabular data structures, exploits in-memory computation, and ensures fault-tolerance.

The high data-access ratio of graph computations[Lumsdaine et al. 2007] have motivated framework designers to work on decreasing communication and synchronization costs by dealing with the *push/pull* dichotomy [Besta et al. 2017]. In the push mode, a vertex pushes its updates to a shared state (e.g. an adjacent vertex). In the pull mode, a vertex pulls the updates to its private state. Such dichotomy constraints the direction of data access, motivating frameworks to support both push and pull modes.

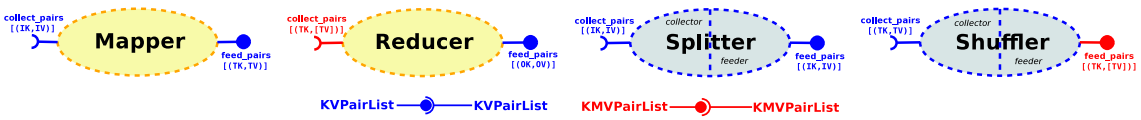



**Figure 1. Building Blocks for a MapReduce Component-Oriented Framework**

## 3. A Component-Oriented MapReduce framework

Figure 1 presents the components of MapReduce parallel computing systems in HPC Shelf. The computations MAPPER and REDUCER represent mapping and reduction


**Figure 2. MapReduce Stages - Computation and Communication**

agents, respectively. In turn, SPLITTER and SHUFFLER represent their connectors. Each component has a set of user ports, called **collect_pairs**, and a set of provider ports, called **feed_pairs**. MAPPER and REDUCER have have single **collect_pairs** and **feed_pairs** ports. For SPLITTER and SHUFFLER, each port belongs to a distinct facet. For that, these connectors are formed by two sets of facets, so-called *collector* and *feeder* ones. A SPLITTER component receives KV-pairs[1] from their **collect_pairs** ports and sends each one through one of its **feed_pairs** ports, according to a partition function applied to the keys. In turn, SHUFFLER groups KV-pairs having the same key in a single KMV-pair[2] before distributing them among **feed_pairs** ports using a possibly distinct partition function.

Data sources may be connected to SPLITTER and SHUFFLER connectors, for representing input, output and auxiliary data repositories. For that, bindings must exist for adapting their services ports to the types of **collect_pairs** and **feed_pairs** ports.

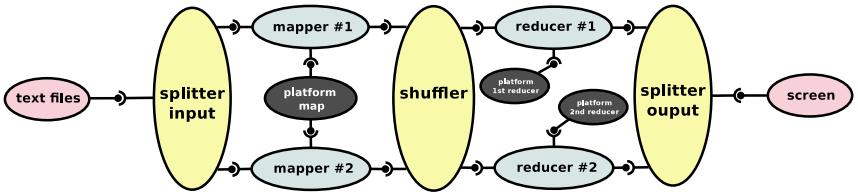



**Figure 3. *Word Counter* MapReduce Component Architecture**

In general, a MapReduce computation is a sequence, possibly iterative, of computation and communication stages (Figure 2). A computation stage may be implemented by a set of parallel mapper or reducer agents, whereas a communication stage is implemented by a single splitter or shuffler. Figure 3 outlines an architecture of a MapReduce parallel computing system for the well-known word counting problem, comprising pairs of parallel mappers and reducers, where the mappers shares the same virtual platform.

Tables 1.1 and 1.2 show the context parameters of the contextual signatures of MRCOMPUTATION and MRCONNECTOR. The former is the base component of MAPPER and REDUCER, whereas the last one is the base component of SPLITTER and SHUFFLER. Indeed, a developer could derive other computation and connector components from MRCOMPUTATION and MRCONNECTOR, with alternative semantics.

Basically, when building a MapReduce parallel computing system for some particular purpose, the developer must assign context arguments to the components, defining the contextual contracts that will guide the selection of appropriate component implementations. For instance, Table 1.3 presents a set of context arguments for *word counting*. COUNTWORDS and REDUCESUM, representing the *map* and *reduce* functions, respectively, are the only components provided by the user. In turn, the default partition function

[1] A KV-pair is a pair $\langle k, v \rangle$, where $k$ is a key and $v$ is a value.

[2] A KMV-pair is a pair $\langle k, [v_1\ v_2\ \ldots\ v_n] \rangle$, where $k$ is a key, mapped to values $v_i$, for $i \in \{1, 2, \ldots, n\}$.

**Table 1. Tables of Context Parameters and Arguments**

1. Context Parameters of MRCOMPUTATION

| name | bound | description |
|---|---|---|
| ***input_key_type*** | DATA | the type of keys in input pairs |
| ***input_value_type*** | DATA | the type of values in input pairs |
| ***function*** | FUNCTION | the custom function (map or reduce) |
| ***output_key_type*** | DATA | the type of keys in output pairs |
| ***output_key_value*** | DATA | the type of values in output pairs |

2. Context Parameters of MRCONNECTOR

| name | bound | description |
|---|---|---|
| ***key_type*** | DATA | The type of keys in pairs |
| ***value_type*** | DATA | The type of values in pairs |
| ***partition_function*** | PARTITION | The custom function for distributing keys across mappers or reducers |

3. Contracts of MAPPER and REDUCER in WordCounter

| Parameter Name | Component | Contextual Bound |
|---|---|---|
| ***input_key_type*** | MAPPER<br>REDUCER | INTEGER<br>STRING |
| ***input_value_type*** | MAPPER<br>REDUCER | STRING<br>INTEGER |
| ***function*** | MAPPER<br>REDUCER | COUNTWORDS[. . . ]<br>SUMVALUES[. . . ] |
| ***output_key_type*** | MAPPER<br>REDUCER | STRING |
| ***output_value_type*** | MAPPER<br>REDUCER | INTEGER |

4. Context Signature of Gusty

| Parameter Name | Var. | Contextual Bound |
|---|---|---|
| ***intermediary_key_type*** | $TK$ | DATA |
| ***intermediary_value_type*** | $TV$ | DATA |
| ***gusty_function*** | $Rf$ | GUSTYFUNCTION[. . . ] |
| ***output_key_type*** | $OK$ | DATA |
| ***output_value_type*** | $OV$ | DATA |
| ***graph_type*** | $G$ | DATA |
| ***input_bin_type*** | $IB$ | INPUTBIN |

5. Context Arguments for the Components of Triangle Enumeration, SSSP and PageRank

| Parameter Name | *Triangle Enumeration* | *SSSP* | *PageRank* |
|---|---|---|---|
| ***input_key_type*** | VERTEXBLOCK | VERTEXBLOCK | VERTEXBLOCK |
| ***input_value_type*** | DATATRIANGLE | DATASSSP | DATAPGRANK |
| ***intermediary_key_type*** | VERTEXBLOCK | VERTEXBLOCK | VERTEXBLOCK |
| ***intermediary_value_type*** | DATATRIANGLE | DATASSSP | DATAPGRANK |
| ***reduce_function*** | TC0[. . . ] / TC1[. . . ] / TC2[. . . ] | SSSP[. . . ] | PAGERANK[. . . ] |
| ***output_key_type*** | VERTEXBLOCK | VERTEXBLOCK | VERTEXBLOCK |
| ***output_value_type*** | DATATRIANGLE | DATASSSP | DATAPGRANK |

is applied to the splitters and the shuffler, since they are not specified in the contract.

Mappers, reducers and multiple facets of splitters and shufflers, as well as sources and sinks, may be arbitrarily connected through compatible bindings between their ports. For example, Figure 4 presents a two-stage iterative MapReduce architecture (ignoring platform components). Also, using appropriate binding adapters, reducers may be connected to the output of splitters and mappers may be connected to the output of shufflers, as well as chains of mappers and reducers may be formed through appropriate indirect bindings, without intermediate connectors. Such a higher flexibility in combining building blocks of MapReduce systems shows how a component-oriented approach may deal with the expressiveness limitations of existing MapReduce frameworks.

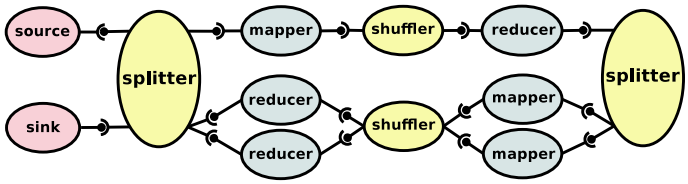



**Figure 4. An Iterative MapReduce Architecture**

## 4. Gust (Graphs Upon Shelf archiTecture)

Gust is a Big Graph framework that extends MapReduce by introducing the abstract components GUSTY, derived from REDUCER, and GUSTYFUNCTION, derived from REDUCEFUNCTION. An application provider must supply the ***reduce_function*** parameter of GUSTY with a contextual contract of an abstract component derived from GUSTYFUNCTION. The contextual signature of GUSTY is presented in Table 1.4.

GUSTYFUNCTION has an inner component of type GRAPH that implements small-scale graph algorithms based on vertices and edges. It can offer either directed


**Figure 5.** ***Triangle Enumeration (non-iterative, no mappers)***

or undirected behavior, depending on the contextual contract arguments. Also, it is distributed, providing a subgraph in each processing unit. Each subgraph extends a *graph interface* from JGraphT [Naveh, B. et al 2003]. GRAPH takes advantage of C# primitive types on generic programming for decreasing memory consumption, compared to the JGraphT's Java implementation. GUSTYFUNCTION allows a programming model where the programmer may codify on either "vertex-centric" or "graph-centric" paradigm.

INPUTBIN is a component used by GUSTY and GUSTYFUNCTION for extracting the graph structure and for creating an instance of GRAPH. At the beginning of the computation, INPUTBIN acts on the partitioning of the graph by cutting vertices, for creating a main vertex and its set of distributed mirrors. During this partitioning, subgraphs are transferred to the processing units, feeding the GRAPH instances.

In GUSTYFUNCTION implementations, the user must provide code for the *unroll*, *compute* and *scatter* methods. ***Unroll*** collects new key/value pairs, produced asynchronously, in the input iterator, for incrementing values that belong to the same key. That is, when the programmer inserts a pair into the output buffer (available in the scatter method) this pair is already available for consumption in the next gusty iteration, being recovered and incremented in that subsequent *unroll*. In the GAS model of PowerGraph, unroll can be seen as an asynchronous gather. In turn, ***compute*** is called after all unroll is completed. It is where the developer inserts the computational logic. Finally, ***scatter*** is used to send data to the output iterator. Enabled after the compute step, it is supposed that the developer has already performed their optimized computations, to finally release their data to neighboring partitions.

*Gusty functions* have been developed for parallel computing systems implementing three graph algorithms often applied to evaluate the expressiveness and performance of Big Graph processing frameworks. They are: ***triangle enumeration*** (TE), ***single-source shortest path*** (SSSP) and ***page rank*** (PR).

Figure 5 shows the architecture of a parallel computing system for *triangle enumeration*, employing three sequential GUSTY agents placed at distinct virtual platforms. Each agent performs a phase of the Triangle Enumeration algorithm. Table 1.5 presents the context arguments of TE components. DATATRIANGLE stores triangles from a partitioned subgraph and VERTEXBLOCK (Inherited from VERTEX) is its key. Thus, the message exchange involves data blocks for subgraph, which is seen as a vertex block. TC0, TC1 and TC2 are the *gusty functions* for each GUSTY component, respectively.

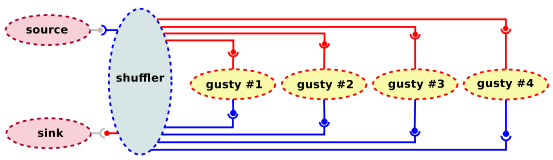



**Figure 6. SSSP and PageRank in virtual platforms #1,#2,#3, and #4**

In turn, Figure 6 depicts the architecture of the SSSP parallel computing system. It is iterative and performs on either vertex-centric or graph-centric paradigm, depending on the key/value types informed through the contextual contracts. *Page rank* may also follow such an iterative architecture. The "graph-centric" paradigm has been used for PageRank. DATAPGRANK (***value_type*** argument) types the value exchanged between gusty agents.

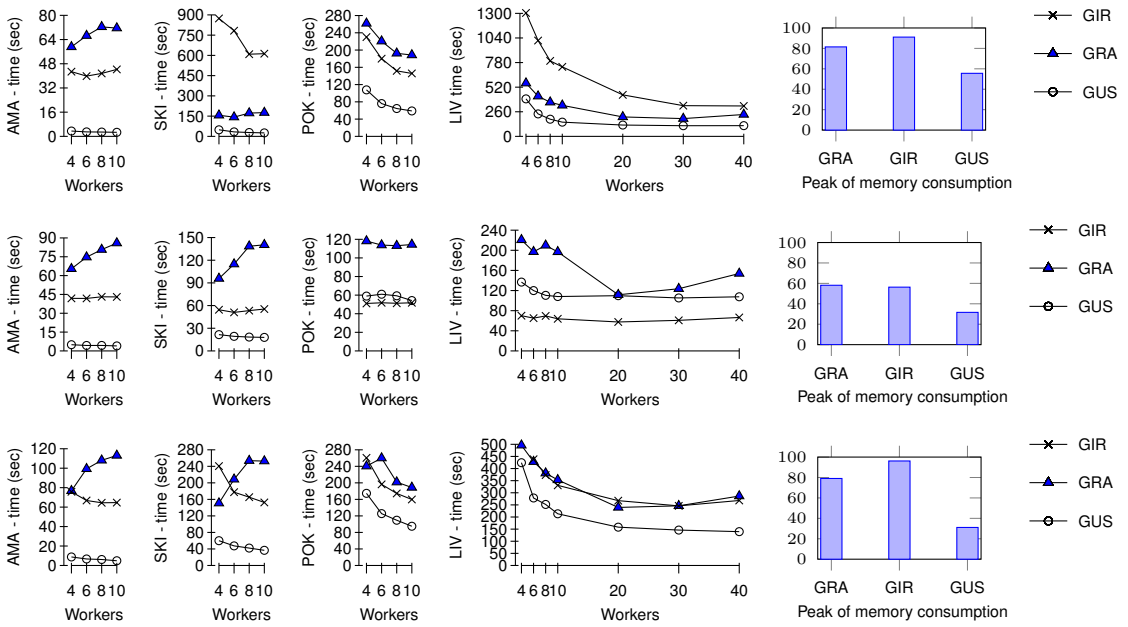



**Figure 7. Triangle Enumeration (top), SSSP (middle) and PageRank (bottom)**

## 5. Case Studies and Performance Evaluation

The performance evaluation presented in this section is intended to give rigorous evidences that Gust may compete with other state-of-the-art frameworks, such as Giraph and Spark/GraphX, in terms of raw computation performance. For that, some decisions have been taken for improving comparison fairness without compromising the generality of the results. Firstly, single cluster execution is assumed. While Gust naturally supports multi-cluster execution, the other frameworks run on single clusters (master/slave model). For Gust, multi-cluster execution means the allocation of mapping and reducing agents at distinct virtual platforms (clusters), so that, for big workloads that may take advantage of multiple clusters, such as that ones which does not fit single cluster resources, interaction costs among mappers and reducers running in different virtual platforms are negligible. Secondly, results of executions recovered from failures are discarded for Giraph and Spark, since fault tolerance, contrariwise to them, is not addressed by Gust yet.

### 5.1. Testbed and Experimental Cases

The experiments have been carried out in a infrastructure comprising 40 virtual machines forming a cluster. Each virtual machine (processing node) has a maximum of 16GB of RAM and 4 vCPUs (virtual CPU). We have used two configurations. The first one is a large cluster with 20, 30, and 40 processing nodes (8GB of RAM per VM). The

**Table 2. Loads**

| Loads | Aliases | Description |
|---|---|---|
| amazon0302 | AMA | 262111 nodes and 1234877 edges |
| skitter | SKI | 1696415 nodes and 11095298 edges |
| pokec | POK | 1632803 nodes and 30622564 edges |
| LiveJournal | LIV | 4847571 nodes and 68993773 edges |

second one is a small cluster with 2, 4, 6, 8 and 10 processing nodes (16GB of RAM per VM). A set of graphs proposed by Stanford University [Leskovec and Krevl 2014], listed in Table 2, has been used in the experiments. Among them, *LiveJournal* (LIV) deserves special attention, since it is the biggest one and has been used in other works [Gonzalez et al. 2012, Xin et al. 2013, Aridhi et al. 2016], including an evaluation of GraphX. The graphs have been randomly partitioned across processing nodes, without load balance. The average execution time of 25 runs for each experimental case has been taken by applying outliers elimination. It combines a choice of benchmark (TE, SSSP, PR), an workload (AMA, SKI, POK and LIV), a framework (GUS, GRA, GIR nad MET) and a number of processing nodes (2, 4, 6, 8, 10, 20, 30, and 40)[3].

## 5.2. Results and Discussion

Figure 7 compares Gust, Giraph and GraphX. Indeed, each line chart shows how their execution times vary with increasing number of processing nodes, for given choices of graph workload and benchmark. The large cluster configurations, with 20, 30 and 40 processing nodes, is applied only to *LiveJournal*, the biggest one. In turn, the bar charts at the right side show the smaller peak memory consumption, in *LiveJournal*, reached by Gust, compared with the other frameworks. Giraph has not been able to run *PageRank* for *LiveJournal* on 4 processing nodes due to the exhaustion of memory resources.

The charts shows, to the reader, Gust outperforming Giraph and GraphX in most of the benchmarks and workloads, in terms of both memory usage and execution time.

[3] http://lia.ufc.br/~cenezaraujo/results.html.

**Table 3. Gust *versus* GraphX and Giraph - *LiveJournal***

| **Gust *versus* GraphX - *LiveJournal*** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\alpha = 0.05$ | | PR | | | | SSSP | | | | TE | | | |
| workers | | **10** | **20** | **30** | **40** | **10** | **20** | **30** | **40** | **10** | **20** | **30** | **40** |
| average | GUS | **213.3** | **158.1** | **146.1** | **139.6** | **108.1** | **109.8** | **105.3** | **107.6** | **150.3** | **119.6** | **112.6** | **113.3** |
| | GRA | **353.1** | **239.5** | **246.2** | **286.5** | **196.9** | **111.6** | **123.6** | **153.9** | **327.8** | **205.3** | **187.9** | **231.1** |
| minimum | GUS | **204.7** | **153.3** | **141.4** | **137.6** | **102.1** | **106.1** | **101.6** | **105.9** | **144.5** | **115.4** | **108.7** | **111.1** |
| | GRA | **336.8** | **225.8** | **232.8** | **270.9** | **164.5** | **101.9** | **114.9** | **139.7** | **313.4** | **186.3** | **171.7** | **214.7** |
| maximum | GUS | **222.3** | **163.1** | **150.5** | **141.5** | **115.1** | **112.2** | **107.5** | **109.2** | **157.6** | **122.6** | **116.2** | **114.6** |
| | GRA | **369.0** | **249.5** | **264.0** | **317.8** | **208.4** | **118.8** | **132.1** | **173.5** | **337.4** | **220.1** | **202.0** | **239.7** |
| st.dev. | GUS | **5.5** | **2.8** | **2.7** | **1.1** | **3.8** | **1.8** | **1.7** | **0.8** | **3.9** | **2.0** | **1.6** | **0.9** |
| | GRA | **8.9** | **6.8** | **9.1** | **12.5** | **11.4** | **4.4** | **4.9** | **8.4** | **6.3** | **8.7** | **8.7** | **6.3** |
| GUS vs GRA | overhead | **-139.8** | **-81.3** | **-100.0** | **-146.9** | **-88.8** | **-1.8** | **-18.3** | **-46.3** | **-177.5** | **-85.7** | **-75.4** | **-117.8** |
| | conf.interval | **1.5** | **1.6** | **2.5** | **4.5** | **3.2** | **1.0** | **1.3** | **3.0** | **1.1** | **2.7** | **2.8** | **2.1** |
| T-test | reject(T-test$<\alpha$)? | **yes** | **yes** | **yes** | **yes** | **yes** | **no** | **yes** | **yes** | **yes** | **yes** | **yes** | **yes** |
| **Gust *versus* Giraph - *LiveJournal*** | | | | | | | | | | | | | |
| $\alpha = 0.05$ | | PR | | | | SSSP | | | | TE | | | |
| workers | | 10 | 20 | 30 | 40 | 10 | 20 | 30 | 40 | 10 | 20 | 30 | 40 |
| average | **GUS** | **213.3** | **158.1** | **146.1** | **139.6** | **108.1** | **109.8** | **105.3** | **107.6** | **150.3** | **119.6** | **112.6** | **113.3** |
| | **GIR** | **330.6** | **267.5** | **245.9** | **268.2** | **63.8** | **57.5** | **60.8** | **66.5** | **734.7** | **438.7** | **324.6** | **321.4** |
| minimum | **GUS** | **204.7** | **153.3** | **141.4** | **137.6** | **102.1** | **106.1** | **101.6** | **105.9** | **144.5** | **115.4** | **108.7** | **111.1** |
| | **GIR** | **313.2** | **251.4** | **229.5** | **242.4** | **51.2** | **50.4** | **55.2** | **61.5** | **640.6** | **327.4** | **259.8** | **281.1** |
| maximum | **GUS** | **222.3** | **163.1** | **150.5** | **141.5** | **115.1** | **112.2** | **107.5** | **109.2** | **157.6** | **122.6** | **116.2** | **114.6** |
| | **GIR** | **344.8** | **278.7** | **259.1** | **289.5** | **75.7** | **64.1** | **66.7** | **71.7** | **875.9** | **547.6** | **374.6** | **400.6** |
| st.dev. | **GUS** | **5.5** | **2.8** | **2.7** | **1.1** | **3.8** | **1.8** | **1.7** | **0.8** | **3.9** | **2.0** | **1.6** | **0.9** |
| | **GIR** | **7.6** | **8.1** | **8.9** | **12.2** | **7.2** | **3.6** | **3.2** | **2.3** | **62.1** | **62.6** | **40.0** | **33.8** |
| GUS vs GIR | overhead | **-117.3** | **-109.4** | **-99.8** | **-128.6** | **44.3** | **52.3** | **44.5** | **41.1** | **-584.4** | **-319.2** | **-212.0** | **-208.1** |
| | conf.interval | **1.0** | **2.2** | **2.5** | **4.4** | **1.3** | **0.8** | **0.7** | **0.6** | **22.9** | **23.7** | **15.1** | **12.9** |
| T-test | reject(T-test$<\alpha$)? | **yes** | **yes** | **yes** | **yes** | **yes** | **yes** | **yes** | **yes** | **yes** | **yes** | **yes** | **yes** |

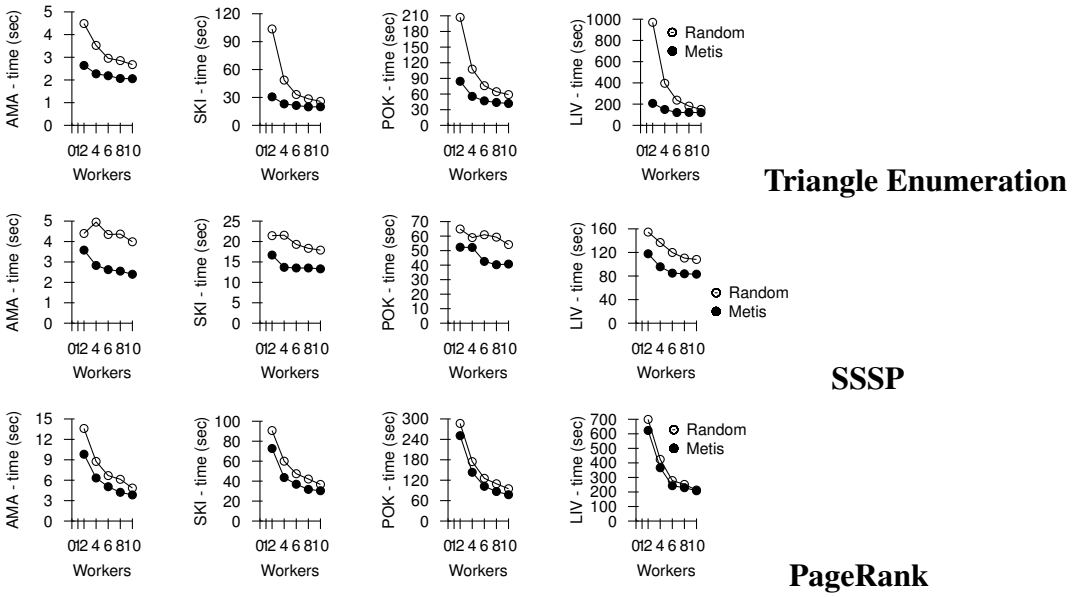



**Figure 8. Gust (Random *versus* Metis Partitioning)**

The exceptions are the experimental cases of *SSSP* with *POK* and *LiveJournal*, where Giraph shows better execution times than Gust (and also GraphX), despite its additional memory requirements. For *LiveJournal*, the difference is more significant.

The experimental cases with the large cluster configurations aim at evaluating the behavior of the frameworks when the available processing resources are increased. Until 20 nodes, it seems that the execution times of GraphX approach Gust for *Triangle Enumeration*, *SSSP* and *PageRank*. However, such a tendency fails for 30 and 40 nodes, giving an evidence that Gust presents better scalability under such a circunstance.

Table 3 presents statistical summaries for *LiveJournal* experimental cases, including *T-test* tests between corresponding *test-cases* that compare Gust to GraphX and Giraph, respectively. It checks whether their average execution times may be considered different (*yes*) or not (*no*) with 95% of confidence. The *T-test* tests confirms that Gust outperforms Giraph and GraphX for almost all the experimental cases, with 95% of confidence. Another important measure is the overhead confidence interval (CI). Negative overheads means that Gust outperforms other systems.

The performance of Gust with a balanced graph partitioning performed by Metis [Karypis and Kumar 1995] is evaluated in Figure 8. It evidences how load balancing affects the performance of Gust, notablying by improving it.

## 6. Conclusion

This paper introduces Gust, a framework for large-scale parallel processing of graphs on top of HPC Shelf, a component-oriented platform for cloud-based HPC services. Component-orientation makes it possible to combine building blocks for implementing different graph processing models, such as Pregel and its variations. In fact, Gust components may be combined in other ways than the known ones. The performance evaluation reported in this paper shows the performance of Gust is competitive with the per-

formance of Giraph and GraphX, two well-known state-of-the-art big graph processing frameworks. This is a significant result, since the optimization of algorithms and data structures have not been a prioritary concern when implementing the components of the first prototype of Gust on top of HPC Shelf. For attending the objectives of the paper, the performance evaluation has considered only single cluster execution. However, it must be noticed that multi-cluster execution, due to HPC Shelf, is an important distinguishing feature of Gust, as exemplified in Section 4 by presenting the archicteture of some parallel computing systems. For future works, besides to continue working on improving the performance of Gust, it is planned to work on fault tolerance requirements.

## References

Aridhi, S., Montresor, A., and Velegrakis, Y. (2016). BLADYG: A Novel Block-Centric Framework for the Analysis of Large Dynamic Graphs. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, HPGP'16, pages 39–42, New York, NY, USA. ACM.

Besta, M., Podstawski, M., Groner, L., Solomonik, E., and Hoefler, T. (2017). To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'17)*. ACM.

Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D. (2012). The HaLoop Approach to Large-Scale Iterative Data Analysis. *The VLDB Journal*, 21(2):169–190.

Carvalho Junior, F. H. and Rezende, C. A. (2013). A Case Study on Expressiveness and Performance of Component-Oriented Parallel Programming. *Journal of Parallel and Distributed Computing*, 73(5):557–569.

Carvalho Junior, F. H., Rezende, C. A., Silva, J. C., Al Alam, W. G., and de Alencar, J. M. U. (2016). Contextual abstraction in a type system for component-based high performance computing platforms. *Science of Computer Programming*, 132:96–128.

Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113.

Doekemeijer, N. and Varbanescu, A. L. (2014). A Survey of Parallel Graph Processing Frameworks. Technical Report PDS-2014-003, Delft University of Technology.

Elnikety, E., Elsayed, T., and Ramadan, H. E. (2011). iHadoop: Asynchronous Iterations for MapReduce. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 81–90, Washington, DC, USA. IEEE Computer Society.

Foundation, A. S. (2008). Hadoop project. http://hadoop.apache.org.

Foundation, A. S. (2011). Giraph project. http://giraph.apache.org.

Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012). PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA. USENIX Association.

Han, M. and Daudjee, K. (2015). Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *Proceedings of the VLDB Endowment*, 8(9):950–961.

Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. (2007). Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72.

Karypis, G. and Kumar, V. (1995). Multilevel Graph Partitioning Schemes. In *Proceedings of the 24th International Conference on Parallel Processing, III*, pages 113–122. CRC Press.

Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data`.

Li, R., Hu, H., Li, H., Wu, Y., and Yang, J. (2016). MapReduce Parallel Programming Model: A State-of-the-Art Survey. *International Journal of Parallel Programming*, 44(4):832–866.

Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. (2012). Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727.

Lumsdaine, A., Gregor, D., Hendrickson, B., and Berry, J. (2007). Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(01):5–20.

Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: A System for Large-scale Graph Processing. In *2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD'10, pages 135–146, New York, USA. ACM.

McCune, R. R., Weninger, T., and Madey, G. (2015). Thinking Like a Vertex: a Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *ACM Computing Surveys*, (2).

Naveh, B. et al (2003). JGraphT. http://jgrapht.org, [Online; acessado 17-Maio-2017].

Plimpton, S. J. and Devine, K. D. (2011). MapReduce in MPI for Large-scale graph algorithms. *Parallel Computing*, 37(9):610–632.

Post, D. E. and Votta, L. G. (2005). Computational Science Demands a New Paradigm. *Physics Today*, 58(1):35–41.

Salihoglu, S. and Widom, J. (2013). GPS: A Graph Processing System. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA. ACM.

Sarkar, V., Williams, C., and Ebcioğlu, K. (2004). Application Development Productivity Challenges for High-End Computing. In *Workshop on Productivity and Performance in High-End Computing (PPHEC'2004)*, pages 14–18.

Silva, J. C. and Carvalho Junior, F. H. (2016). A Platform of Scientific Workflows for Orchestration of Parallel Components in a Cloud of High Performance Computing Applications. In *Lecture Notes in Computer Science*, volume 9889, pages 156–170. Springer.

Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J. (2013). From "Think Like a Vertex" to "Think Like a Graph". *Proceedings of the VLDB Endowment (PVLDB)*, 7(3):193–204.

Valiant, L. G. (1990). A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111.

van der Steen, A. J. (2006). Issues in Computational Frameworks. *Concurrency and Computation: Practice and Experience*, 18(2):141–150.

Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. (2013). GraphX: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA. ACM.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA. USENIX Association.

# Assessing the Computation and Communication Overhead of Linux Containers for HPC Applications

**Guilherme Rezende Alles, Alexandre Carissimi, Lucas Mello Schnorr**

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{gralles,asc,schnorr}@inf.ufrgs.br

***Abstract.*** *Virtualization technology provides features that are desirable for high-performance computing (HPC), such as enhanced reproducibility for scientific experiments and flexible execution environment customization. This paper explores the performance implications of applying Operating System (OS) containers in HPC applications. Docker and Singularity are compared to a native baseline with no virtualization, using synthetic workloads and an earthquake simulator called Ondes3D as benchmarks. Our evaluation using up to 256 cores indicate that (a) Singularity containers have minor performance overhead, (b) Docker containers do suffer from increased network latency, and (c) performance gains are attainable with an optimized container on top of a regular OS.*

## 1. Introduction

One fundamental aspect of scientific computing research is that, in order for it to be credible, it should be reproducible. This means that the researcher has the responsibility of providing its readers with the resources necessary to run experiments and obtain similar results to the ones provided by the research. Although the concept of reproducibility is well known in the scientific community, it is very hard to put it to practice because of the large number of variables that comprise an execution environment. Additionally, HPC resources are usually managed under strict usage policies, many of which do not allow the user to customize the environment with an arbitrary software stack (e.g. compilers, libraries or device drivers), thus making it very difficult to obtain a uniform execution platform for a given experiment across different clusters.

Although being intuitively difficult to deal with, these problems can be solved through the use of traditional hardware virtualization, in which users can run their own operating system and execution environment on top of a hypervisor, ensuring that the software stack can be configured to the exact requirements for a given experiment. The problem with this approach is that, when considering HPC applications, the performance overhead introduced by the hypervisor and an entire virtual machine is too high for this solution to be considered feasible. As an alternative to virtual machines, container technologies have gained a lot of attention in the software development industry. This is mainly because containers offer the benefits of virtualization at a fraction of the performance overhead introduced by virtual machines [Chung et al. 2016]. By using containers to isolate and package software, users are able to create reproducible, configurable environments for their applications. This diminishes the hassle of managing application dependencies and software stack differences when executing experiments across different

Linux environments. The problem is that the performance implications of applying Operating System (OS) containers in parallel applications remains relatively unexplored in the HPC community.

This paper compares and contrast two container technologies, Docker and Singularity, with respect to their performance overhead and ability to solve common HPC workflow problems. We will consider the same experiments running on a native environment as the baseline for performance comparison using multiple nodes (up to 64 hosts, 256 cores). The main contributions of this paper are:

- We employ three cases: the NAS-EP [Bailey et al. 1991], the earthquake simulator Ondes3D [Dupros et al. 2010] and a in-house MPI-based Ping-Pong application to evaluate network latency (Section 4.1);
- We demonstrate that Singularity containers have minor computation overhead because of extra work during initialization; and that this overhead is absorbed for longer runs independent of the number of MPI ranks (Section 4.2);
- Docker containers with network virtualization (using the Docker Swarm) suffer from increased network latency that strongly penalizes communication-bound applications such as Ondes3D (Section 4.3);
- Performance gains are attainable with an optimized container on top of a regular OS such as our comparison of a Singularity container based on Alpine Linux that reduces the Ondes3D execution time of up to ≈5% in parallel runs (Section 4.4).

Section 2 presents some basic concepts regarding container technologies, Docker, and Singularity. Section 3 discusses related work in the field of containers and virtualization in high-performance computing contexts, and motivate our work. Section 4 presents the experimental design and benchmarks, and a detailed description of the main contributions of this paper. Section 5 concludes the paper with main achievements and detail some future work. The companion material of this work, including the source code, analysis scripts, and raw measurements, is publicly available at `https://github.com/guilhermealles/hpc-containers/`.

## 2. Basic Concepts

Operating System (OS) Containers [Soltesz et al. 2007] are a mean of achieving virtualization without relying on software to emulate hardware resources. Therefore, containers are known as software level virtualization for Linux systems, and they orchestrate features (*cgroups* and *namespaces*) that are native to the Linux kernel to isolate the resources managed by the OS. As of result, software that runs inside of a container can have its own file system, process tree, user space and network stack, giving it the impression of being executed on an isolated environment. Containers present a theoretically negligible overhead penalty when compared to an application running natively on the host operating system. This happens because the Linux kernel already uses *cgroups* and *namespaces* to manage its resources internally, even when there are not multiple containers on a single machine. Considering this approach, a non-virtualized Linux environment can be seen itself as a single container running on top of the Linux kernel, which means that there is no additional software layer in a container that should insert execution overhead. The core APIs and functionality used to create containers are not new, and have been present in the Linux kernel for more than a decade. Containers become mainstream after a long

time especially because of how difficult it is for an end user to interact with these kernel APIs directly. Conversely, containers only became popular when software (such as LXC, OpenVZ, Systemd-nspawn, Docker, rocket container runtime – rkt, and Singularity) was created to interact with the kernel and mediate the creation of containers. These container management platforms also introduced new features which are very desirable for many workflows (including software development and HPC), such as the ability to encapsulate an entire environment in an image that can be distributed and reproduced on top of different hardware, improving reproducibility and dependency management. Among all alternatives, we describe below two of them: Docker and Singularity since they are the more prominent and widely used in the OS and HPC communities.

**Docker** [Merkel 2014] is a very popular container system for software development and service deployment. Every major cloud infrastructure provider (such as AWS, Google Cloud Platform, and Microsoft Azure) supports Docker as a platform for executing software, and companies all over the world rely on it to deploy its services. Docker implements a virtualization model that, by default, isolates as many aspects of the underlying operating system as possible. As a result, a Docker container has many aspects that resemble a traditional virtual machine: it has its own network stack, user space, and file system. By virtualizing the network stack, Docker relies on a virtual controller that uses Network Address Translation (NAT) to correlate multiple containers to the host's IP address. This approach forces the user to explicitly specify which ports of the container should be exposed to the host operating system, allowing the user to have a finer control over network communication on the container.

Additionally, the user space is also separated between container and host. This means that there is a new root user inside the container, which is controlled by the user who starts it. This turn customization easier, for example to install libraries and packages and make modifications to the virtualized operating system. On the other hand, it also presents a security concern on shared environments, because a user can mount the root directory from the host operating system as a volume in the container, thus granting access to all the files in the host machine. Docker mitigates this issue by requiring root privileges in the host operating system for a user to create containers. Although efficient, this limitation imposes a barrier in adopting Docker as a standard container platform for shared environments in which not every user is granted with root privileges.

**Singularity** [Kurtzer et al. 2017] is a container system developed for scientific research and high-performance computing applications. Contrary to Docker, Singularity does not aim to create completely isolated environments. It relies on a more open model, with the objective of providing integration with existing tools installed on the host operating system. Consequently, the only namespace that is isolated between the host and a Singularity container is the file system. Other namespaces remain untouched by default. Thus, the network stack, process tree, and user space are the same between container and host, which lead to the container being seen as a process which is executed in the host operating system. This feature is very important for two reasons. First, Singularity containers can be started and killed by any tool used to manage processes, such as *mpirun* or even SLURM. Second, because the user space is untouched, the user that executes processes inside the container is the same as the one which started the container, which means that regular users can start a container without needing root access in the host OS.

## 3. Related Work and Motivation

Experiments to measure and evaluate the performance of virtualized environments for HPC have already been done in the past. One particular study compared the performance of Docker containers to traditional virtual machines for single-node applications, concluding that the former has a considerably lower overhead when compared to the latter [Chung et al. 2016]. The same conclusions were drawn when considering experiments that run on multiple physical nodes [Higgins et al. 2015] and with more complex application signatures that are common in HPC, such as load imbalance and communication with other processes [Beserra et al. 2015]. Additional work has also shown that there is some additional overhead when comparing the execution time of applications on top of containers to applications in the native environment (with no virtualization) [Ruiz et al. 2015].

An investigation work proposed a model of MPI cluster in a distributed environment [Nguyen and Bein 2017]. In this study, Docker containers are connected through an orchestrator called Docker Swarm, which is responsible for assigning names and providing network connectivity between the containers, leveraging Docker's overlay networking capabilities. Performance analysis, however, is absent from this study, obscuring the conclusion of whether such an approach is viable in a real-world scenario. Furthermore, it has been shown that the performance of network operations can be affected by the use of Docker containers, especially in latency-sensitive scenarios [Felter et al. 2015].

Singularity [Kurtzer et al. 2017] is a container system designed for scientific research workflows, and it strives to solve some drawbacks of using Docker in HPC. Author argues that Docker is not designed for shared, multi-user environments (as discussed in Section 2), something very common in supercomputers. As a consequence, it is very hard to find HPC centers that allow users to execute Docker containers. Singularity, on the other hand, solves these problems to make HPC containers more accessible to the scientific community. Consequently, Singularity containers are already accepted and used in many supercomputers around the world. Additionally, a performance analysis of applications running on top of Singularity containers has also been carried out [Le and Paz 2017]. It concludes that while some overhead does exist, the reported values are negligible for most use cases.

**Motivation:** The goal of this work is to study the drawbacks and improvements that occur by applying virtualization techniques to high-performance computing workflows. As concluded by previous work, using virtual machines is unfeasible because of the overheads that comes along with this strategy. Thus, our goal is to measure the performance impact of applying container-based virtualization to these HPC workloads. We present an analysis covering both synthetic benchmarks and a real application comparing the performance implications of Docker and Singularity, two major container systems, and using a traditional approach (with no virtualization) as baseline. Furthermore, we intend to demonstrate that virtualization techniques can be used in HPC without the massive overhead of traditional virtual machines. Next section details our results toward these goals.

## 4. Results and Evaluation of the Performance Overhead

Results are based on measurements obtained from experiments with multiple compute nodes of the Grid5000 platform [Balouek et al. 2013], in a controlled setup. In what follows, we present (a) the software/hardware stack adopted across all experiments with

three cases (Native, Docker, Singularity) and three benchmarks (NAS-EP, Ondes3D, Ping-Pong); (b) the computation overhead analysis with a comparison between docker, singularity, and native; (c) a verification of the increased communication latency leading to bad application performance; and (d) a comprehensive analysis to verify how performance gains can be used solely in applying an optimized container on top of an optimized OS.

### 4.1. Software/Hardware Environment, Benchmarks, and Workload Details

The Grid5000 is a platform used for scientific experiments in parallel computing, HPC, and computer science. It provides its users with many clusters that can be reserved for exclusive use for a limited time. We executed the experiments in the Grid5000's `graphene` cluster (at Nancy - France), which contains 131 nodes, each one equipped with 16GB of DDR3 memory and a quad-core Intel Xeon X3340 (Lynnfield, 2.53GHz), and interconnected by a 1 Gigabit Ethernet and a 20 Gbps Infiniband network. We used up to 64 compute nodes for our tests using exclusively the 1 Gigabit Ethernet because of limitations in the container configuration. In all experiments, each node received a maximum of 4 MPI processes due to the 4-core availability of processor cores. All compute nodes are initially deployed (using `kadeploy3` [Jeanvoine et al. 2013]) with the default Debian9 OS image, before laying the Docker or Singularity environment on top of it.

To ensure consistency between the container environments against the Native case, the same Debian9 Linux distribution was used for such environments in both Docker and Singularity containers. We have used a previously proposed [Nguyen and Bein 2017] multi-node container infrastructure for Docker where physical nodes are connected using the Docker Swarm utility. This tool is responsible for spawning containers on all the nodes and connecting them via an overlay network, so that every container (which will execute an MPI process) can be addressed by the MPI middleware. The multi-node container infrastructure for Singularity is similar to the one with native processes. Because Singularity containers share the network stack with its host, there is no need for a virtual network between the containers. Therefore, processes in Singularity containers communicate through the physical network.

Three parallel applications are used to evaluate the performance in the OS options (Native, Docker and Singularity): NAS-EP, Ondes3D, and Ping-Pong, detailed as follows. The NAS Embarrassingly Parallel – **NAS-EP** – is part of the NAS Parallel Benchmarks (NPB) [Bailey et al. 1991]. NAS-EP generates independent Gaussian random numbers using the polar method, being considered a CPU-bound case with parallel speedup close to ideal since communication takes place in the beginning and end of the execution. EP is executed with the class B workload using one to four hosts (4 to 16 cores) in preliminary tests. **Ondes3D** [Dupros et al. 2010] is developed at the BRGM (French Geological Survey) as an implementation of the finite-differences method (FDM) to simulate the propagation of seismic waves in three-dimensional media. As previously observed [Tesser et al. 2017], its signature contains characteristics such as load imbalance and frequent asynchronous small-message communications among MPI ranks. Two workloads have been used to run Ondes3D: the default test case without a geological model (synthetic earthquake) and a real Mw 6.3 earthquake that arose in Liguria (north-western Italy) in 1887 [Aochi et al. 2011] with 300 timesteps, which has been used as workload for our tests. So, this real-world application is also evaluated to verify if it is impacted by OS containers. Finally, an in-house **Ping-Pong** benchmark developed with MPI (see the

companion material for the source code) was used to assess the bandwidth and latency performance when introducing the container's virtual environment. This evaluation is conducted between two nodes that exchange MPI messages, with message sizes varying from 1Byte to 1MByte.

We generate two randomized full factorial designs [Jain 1991] to drive experiments and collect measurements for NAS-EP and Ondes3D. The first design targets a smaller scale test using up to four nodes, with 1, 4, 8, and 16 processes; the second design uses 64 nodes, with 64, 128, 192, and 256 processes. The first batch uses NAS-EP executed with the Class B workload (identified by NAS-EP/ClassB) and Ondes3D with the default test case (Ondes3D/Default). The second batch of experiments considers the Ondes3D application using the Ligurian workload (Ondes3D/Ligurian). The Ping-Pong application has been used in a separated batch since it uses only two compute nodes of the graphene cluster. Messages size corresponding to powers of two from 1B to 1MBytes (21 data points) has been sequentially measured. All reported makespan and ping-pong measurements are averages from 10 to 30 replications of each experiment parameter configuration; error bars are calculated considering a confidence level of 99.7% assuming a Gaussian distribution.

### 4.2. Computation Overhead Analysis

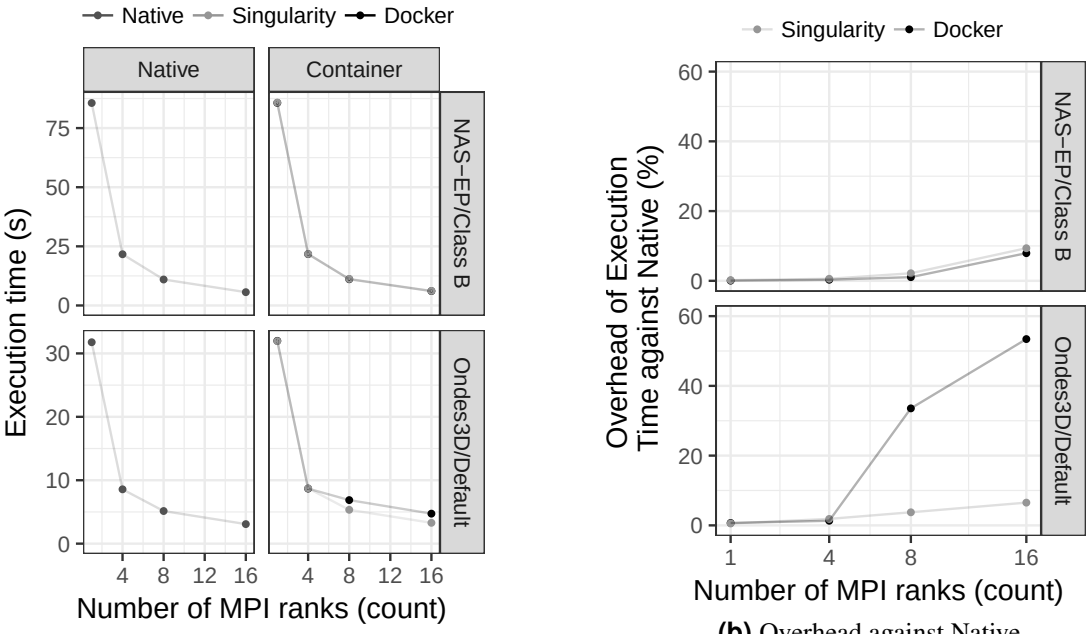
We present the results of the computation overhead for the small case scenarios (up to 16 cores) of NAS-EP/ClassB and Ondes3D/Default, then the larger scenario with the Ondes3D/Ligurian case, using 64 nodes and 256 cores.

#### Small case (4 nodes, 16 cores) with NAS-EP/ClassB and Ondes3D/Default

Figure 1a shows the makespans of the NAS-EP/ClassB (the top facets in the first row) and the Ondes3D/Default (bottom), with respect to the number of MPI ranks for the Native (left facets) and the Containers (right) – Singularity and Docker. Figure 1b depicts the execution time overhead with respect to the number of MPI ranks, calculated for each container environment against the native runs, also for both applications.

For the **NAS-EP/ClassB** case, Figure 1 (top facets) shows that the virtualized approaches perform very close to each other and to the native baseline. For 16 MPI ranks, the Docker overhead is of 8% while the Singularity imposes a slightly higher overhead of 9%. Although limited, both indicate an alarming increasing trend. This difference in execution time can be related to the time needed to spin up the containers and should increase as the number of containers (and MPI ranks) increases. However, since these runs were short – less than 7s for 16 processes in NAS-EP/ClassB (see Figure 1a) – such overhead may be absorbed with longer CPU-bound runs that make a limited use of the communications. For the **Ondes3D/Default** case (bottom facets of Figures 1a and 1b), we observe that the performance on the three environments is similar for 1 and 4 MPI ranks. However, the Docker performance degrades when going up to 8 and 16 ranks with execution time overhead of $\approx$33% for 8 MPI ranks and $\approx$53% for 16. This behavior surfaces exactly when more physical nodes are added to the experiment, which indicates that the network communication might be impacting the performance of Docker containers. This hypothesis is further supported by the virtual network (Docker Swarm)

that is required to provide connectivity between Docker containers. Such a virtual network does not exist in the other two environments. Although the Singularity container poses some overhead (≈6% for 16 ranks), we believe it has the same reason for the NAS-EP/ClassB case, so unrelated to the network.




**(a)** Execution Time

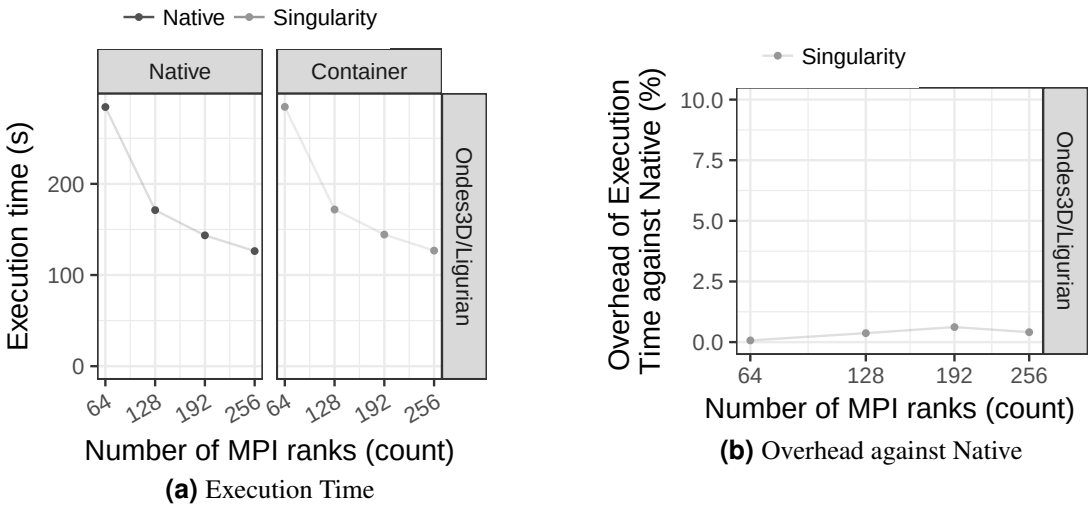
**(b)** Overhead against Native

**Figure 1. (a) Execution time as a function of the number of MPI ranks for the three environments (Native, Docker, and Singularity) and the applications (NAS-EP/ClassB, and Ondes3D/Default), and (b), the execution overhead of the container environments against the native environment with respect to the number of MPI ranks.**

### Large case (64 nodes, 256 cores) with Ondes3D/Ligurian

Figure 2 shows a large-scale simulation of the Ligurian earthquake on Ondes3D. This experiment was conducted to put Singularity in a highly-distributed computing scenario, and its main objective is to assess the aggregated overhead of spawning a large number of containers across multiple nodes. Unfortunately, the container infrastructure for Docker using its overlay network and Docker Swarm as an orchestrator failed to spawn containers in such a high number of nodes, and thus Docker was excluded from this test case. As the plot indicates (see Figure 2a), there is no observable difference in execution time between the two approaches (Singularity and Native), which indicates that the additional cost of executing applications in a Singularity environment is negligible even when spawning a high number of containers. The Figure 2b shows the computed overhead as a function of the number of MPI ranks, revealing a minor overhead of less than ≈1% in all cases. We believe the overhead is minor because of the longer run (more than 100s), so any initialization time imposed by the container is more easily absorbed by the run. Surprisingly, the overhead is smaller with 256 ranks, breaking the upward trend from 64 to 192. This is probably due to the diminishing returns on speedup as the number of cores increase,

which can mask the container initialization time with other overheads that are the constant in both environments, such as network communication.




**(a)** Execution Time

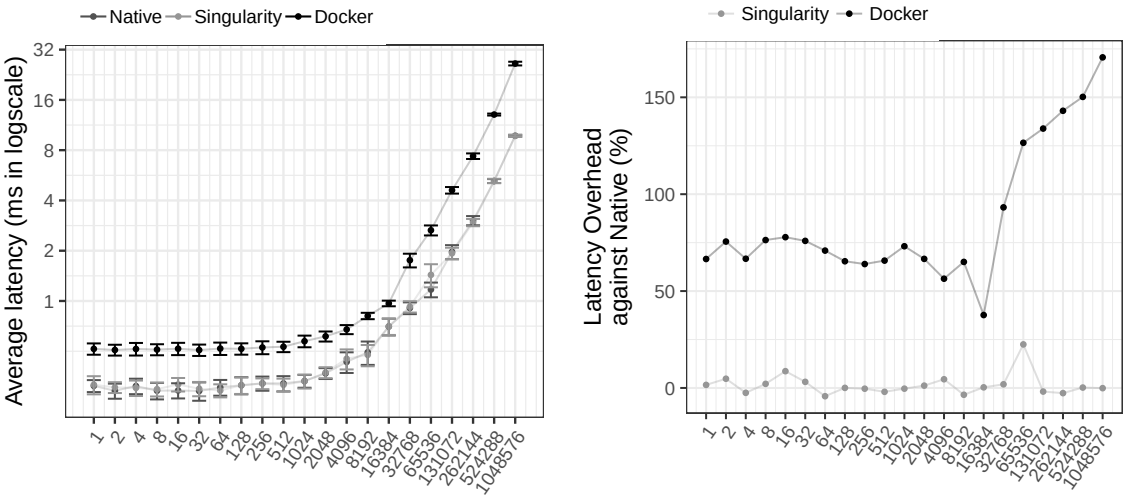
**(b)** Overhead against Native

**Figure 2. (a) Execution time as a function of the number of MPI ranks for the three environments (Native and Singularity) with the Ondes3D/Ligurian application, and (b), the Singularity overhead with respect to the number of MPI ranks.**

### 4.3. Verification of Increased Communication Latency

The results obtained with Ondes3D/Default using the Docker environment (see previous Subsection) led us to design an experiment to demonstrate that the bad performance is caused by network issues. Figure 3 presents the Ping Pong benchmark which was used to measure the communication latency from the application point of view. Figure 3a depicts the average latency (on the logarithmic scale Y axis) between two nodes for the three environments (differentiated by color) as a function of the message size (on X, also log scale). From these results, we can see that the Docker network latency is much higher when compared to both the native and singularity environments, therefore with poorer performance. This evidence confirms that, as observed in the Ondes3D/Default experiment, the virtual network (Docker Swarm) used by Docker introduces significant overhead to communication. Singularity containers, on the other hand, use the same network stack as the host operating system, resulting in non-observable performance differences since most of the average latency is within the confidence interval of native measurements. The Figure 3b shows the latency overhead of each container environment against the native physical interconnection. We can see that the overhead imposed by Singularity in the communication latency remains stable no matter the message size, which is something desirable. In some cases the Singularity overhead is negative, meaning that average latency measured within Singularity is smaller than the average with the native OS. This is just an artifact since we show (see Figure 3a), that confidence intervals of Singularity and Native overlap, indicating no statistical difference. The case for Docker is much worse because (a) the overhead is $\approx$75% against native, and (b) it dramatically increases after the message size 32KBytes. This indicates the low scalability of the approach, especially for those applications with larger message sizes, but also impacting applications that mostly used smaller messages. For instance, in the case of Ondes3D previously studied, ranks exchange multiple small

messages according to the domain decomposition. Even if most messages are exchanged asynchronously, the latency impact on the application is easily observed (see Figure 1b).

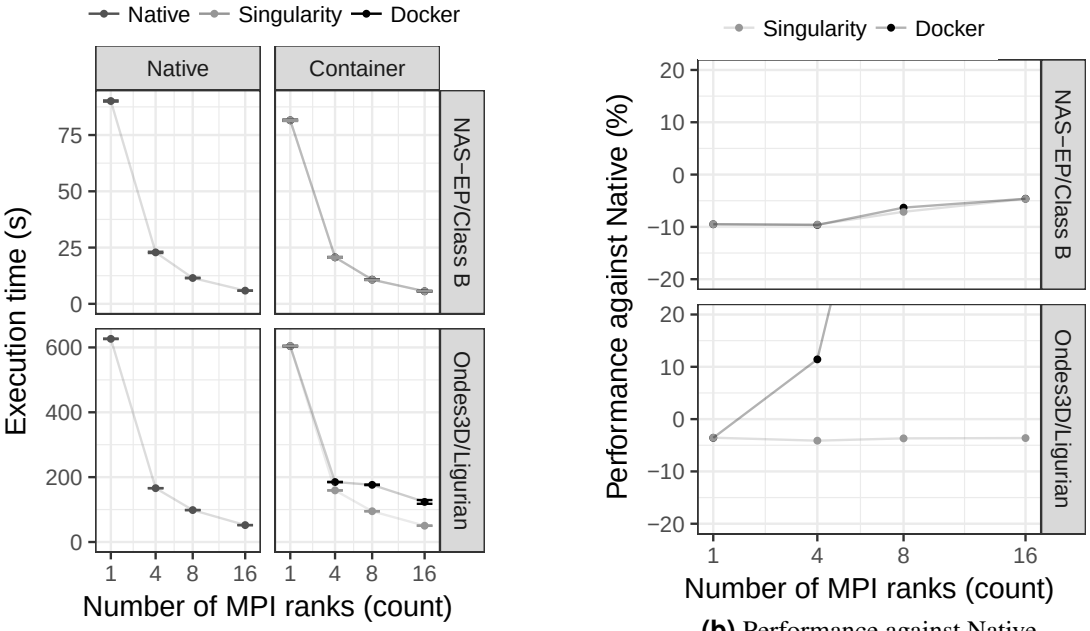



**(a)** Average Latency (ms) **(b)** Latency Overhead

**Figure 3. (a) Average network latency (on Y log scale) measured with the Ping Pong benchmark for the three environments (color) as a function of message size (on X log scale), and (b), the same but showing the derived latency overhead against native (on Y).**

### 4.4. Performance Gains with an Optimized Container based on Alpine Linux

To illustrate the advantages in flexibility for environment configuration, we also conducted an experiment running an Alpine Linux image on the container environments (Singularity and Docker). The Alpine Linux is a lightweight Linux distribution that strives for efficiency and isolation. It is based on Busybox [Wells 2000] and provides an alternate set of standard libraries that can yield better performance for some applications. Installing a completely different Linux distribution on multiple hosts of a cluster for a single experiment is generally a very hard task, sometimes even considered unfeasible (especially in a shared cluster environment). However, this task can be easily done when using containers. Figure 4 shows how Docker and Singularity (running the Alpine Linux distribution) compare to the native operating system (running Debian) both in terms of average execution time (Figure 4a) and performance difference against the native host (Figure 4b) as a function of the number of MPI ranks. These results show that, by modifying the execution environment, it is possible for the virtualized execution to outperform the native one. In the Ondes3D/Ligurian case, Singularity/Alpine is ≈5% faster than native, while in the NAS-EP/ClassB, both Docker and Singularity running Alpine are from ≈5% (with 16 cores) to ≈10% (sequential) faster than the native host when equipped with Debian9. Such results are not so surprising but are still unconventional. This experiment shows that using a fine-tuned, HPC-tailored container in experiments can bring performance advantages as well as a reproducible environment.




**(a)** Execution Time

**(b)** Performance against Native

**Figure 4. (a) Execution time for the NAS-EP/ClassB benchmark and Ondes3D/Ligurian with containers running Alpine Linux while the host is running Debian9, and (b), performance difference of Alpine Linux containers (Singularity and Docker) against the Debian9 Native environment. Negative percentages in (b) indicate the performance gains of the containers.**

## 5. Conclusion

In this paper, we assess the performance implications of the adoption of Linux Containers for HPC applications with different workloads. While containers provide similar features as hardware level virtualization with a theoretically negligible performance overhead, making them suitable for high-performance applications has to be evaluated prior using in production environments. Therefore, we compared two container technologies, Docker and Singularity, against a native environment running with no virtualization.

The results for the proposed tests indicate that containers introduce very little (if any) computation overhead in CPU-bound applications, for both Docker and Singularity. This can be verified by the lack of a clear performance difference on the EP-NAS/ClassB Benchmark among the container and native environments. Although we observed penalties up to $\approx$9%, they are completely absorbed if the applications last longer. Communication overhead, on the other hand, has been observed in Docker containers. This is mainly because the Docker architecture requires the containers to be connected through an overlay network in order for them to have connectivity across multiple hosts (which was needed for the MPI cluster). This overhead was observed in both the Ping Pong test case as well as the Ondes3D application, which is known to require frequent communication between MPI processes. Singularity is free from such overheads. Additionally, we conducted experiments that leveraged the potential flexibility that a virtualized workflow provides. Because containers allow users to fine-tune the execution environment more easily, it was possible to use a different Linux distribution without having root access to the host operating system. This approach yielded better performance than the native exe-

cution, which means that it is possible to use these fine-tuning capabilities to considerably enhance the performance of HPC applications.

With our experiments, we can conclude that Linux containers are a suitable option for running HPC applications in a virtualized environment, without the drawbacks of traditional hardware-level virtualization. In our tests, we concluded that Singularity containers are the most suitable option both in terms of system administration (for not granting every user that starts a container root access to the system) and in terms of performance (for not imposing an overlay network that is a potential bottleneck).

As future work, we plan to investigate HPC applications within containers that make use of low-latency Infiniband networks and multi-GPU systems. We also intend to verify in further details if the computation signature of HPC codes are the same outside and inside the container.

## Acknowledgements

We thank these projects for supporting this investigation: FAPERGS GreenCloud (16/488-9), the FAPERGS MultiGPU (16/354-8), the CNPq 447311/2014-0, the CAPES/Brafitec EcoSud 182/15, and the CAPES/Cofecub 899/18. Experiments were carried out at the Grid'5000 platform (`https://www.grid5000.fr`), with support from Inria, CNRS, RENATER and several other french organizations. The companion material is hosted by GitHub for which we are also grateful.

## References

[Aochi et al. 2011] Aochi, H., Ducellier, A., Dupros, F., Terrier, M., and Lambert, J. (2011). Investigation of historical earthquake by seismic wave propagation simulation: Source parameters of the 1887 m6. 3 ligurian, north-western italy, earthquake. In L'Association Française du Génie Parasismique (AFPS), editor, *8ème colloque AFPS, Vers une maitrise durable du risque sismique*.

[Bailey et al. 1991] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., et al. (1991). The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73.

[Balouek et al. 2013] Balouek, D., Carpen Amarie, A., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., Richard, O., Pérez, C., Quesnel, F., Rohr, C., and Sarzyniec, L. (2013). Adding virtualization capabilities to the Grid'5000 testbed. In Ivanov, I. I., van Sinderen, M., Leymann, F., and Shan, T., editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing.

[Beserra et al. 2015] Beserra, D., Moreno, E. D., Endo, P. T., Barreto, J., Sadok, D., and Fernandes, S. (2015). Performance analysis of lxc for hpc environments. In *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 358–363.

[Chung et al. 2016] Chung, M. T., Quang-Hung, N., Nguyen, M. T., and Thoai, N. (2016). Using docker in high performance computing applications. In *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, pages 52–57.

[Dupros et al. 2010] Dupros, F., Martin, F. D., Foerster, E., Komatitsch, D., and Roman, J. (2010). High-performance finite-element simulations of seismic wave propagation in three-dimensional nonlinear inelastic geological media. *Par. Comput*, 36(5):308 – 325.

[Felter et al. 2015] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172.

[Higgins et al. 2015] Higgins, J., Holmes, V., and Venters, C. (2015). Orchestrating docker containers in the hpc environment. In Kunkel, J. M. and Ludwig, T., editors, *High Performance Computing*, pages 506–513, Cham. Springer International Publishing.

[Jain 1991] Jain, R. (1991). *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley.

[Jeanvoine et al. 2013] Jeanvoine, E., Sarzyniec, L., and Nussbaum, L. (2013). Kadeploy3: Efficient and scalable operating system provisioning for clusters. *USENIX; login:*, 38(1):38–44.

[Kurtzer et al. 2017] Kurtzer, G. M., Sochat, V., and Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20.

[Le and Paz 2017] Le, E. and Paz, D. (2017). Performance analysis of applications using singularity container on sdsc comet. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, pages 66:1–66:4, New York, NY, USA. ACM.

[Merkel 2014] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

[Nguyen and Bein 2017] Nguyen, N. and Bein, D. (2017). Distributed mpi cluster with docker swarm mode. In *IEEE 7th Annual Comp. and Comm. Workshop and Conf.*

[Ruiz et al. 2015] Ruiz, C., Jeanvoine, E., and Nussbaum, L. (2015). Performance evaluation of containers for hpc. In Hunold, S., Costan, A., Giménez, D., Iosup, A., Ricci, L., Gómez Requena, M. E., Scarano, V., Varbanescu, A. L., Scott, S. L., Lankes, S., Weidendorfer, J., and Alexander, M., editors, *Euro-Par 2015: Parallel Processing Workshops*, pages 813–824, Cham. Springer International Publishing.

[Soltesz et al. 2007] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. (2007). Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287.

[Tesser et al. 2017] Tesser, R. K., Schnorr, L. M., Legrand, A., Dupros, F., and Navaux, P. O. A. (2017). Using simulation to evaluate and tune the performance of dynamic load balancing of an over-decomposed geophysics application. In *European Conference on Parallel Processing*, pages 192–205. Springer.

[Wells 2000] Wells, N. (2000). Busybox: A swiss army knife for linux. *Linux J.*, 2000(78es).

# Meta-Análise de Artigos Científicos Segundo Critérios Estatísticos: Um estudo de caso no WSCAD*

**Alessander Osorio, Marina Dias, Gerson Geraldo H. Cavalheiro**

[1]Programa de Pós-Graduação em Computação
Universidade Federal de Pelotas
Pelotas – RS – Brasil

{alessander.osorio,mldias,gerson.cavalheiro}@inf.ufpel.br

***Abstract.*** *This paper presents the systematization of the results of the meta-analysis of the publications of the 18 editions of the WSCAD according to the categories: statistics, metrics and tests. The goal is not to invalidate the results, but to alert to how they are described and presented.From a sample of 426 publications, 93 % referred at least one of the terms searched, showing that there is some care in the demonstration of results, even inadequate or incomplete given the low occurrence of only 3 % of statistical tests confirmed by a second sample of 30 articles. It is necessary not only to focus on the development of the research object itself, but also on the results demonstrations.*

***Resumo.*** *Este artigo apresenta a sistematização dos resultados da meta-análise das publicações das 18 edições do WSCAD segundo as categorias: estatística, métricas e testes. O objetivo não é invalidar os resultados, mas alertar para a forma como são descritos e apresentados. Da amostra de 426 publicações analisadas, 93% fizeram referência a pelo menos um dos termos pesquisados, mostrando que existe a preocupação na demonstração dos resultados, mesmo que inadequada ou incompleta dada a baixa ocorrência de apenas 3% de testes estatísticos confirmada por uma segunda amostra de 30 artigos. É preciso não apenas colocar foco no desenvolvimento do objeto de pesquisa em si, mas também na aferição e apresentação dos resultados.*

## 1. Introdução

A apresentação de uma nova técnica ou algoritmo usualmente é acompanhada de uma análise de desempenho. Deve-se atentar para que o estudo de desempenho seja realizado de forma a que o ganho, caso exista, possa ser atestado. Se observa que, em muitos casos, pesquisadores dedicam grande quantidade de tempo para execução dos experimentos mas, em contrapartida, limitam-se a apresentar dados de desempenho sem realizar um estudo estatístico que os valide.

O estudo estatístico a ser realizado, como considerado neste artigo, é aquele estudo em que é realizada a análise da consistência e coerência dos dados de desempenho obtidos. Esta etapa deve preceder a inferência de comportamentos, interpretações sobre os resultados coletados e as conclusões obtidas. Portanto, a efetiva realização do estudo

*O presente trabalho foi realizado com apoio do Programa Nacional de Cooperação Acadêmica da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – CAPES/Brasil.

estatístico permite associar confiabilidade aos resultados apresentados em qualquer documento de divulgação científica.

Neste trabalho é realizada uma meta-análise qualitativa, por meio de processo automatizado de mineração de dados, dos artigos dos últimos 18 anos do Simpósio de Sistemas Computacionais de Alto Desempenho (WSCAD) para extrair a síntese sobre os métodos estatísticos e métricas utilizados nestas publicações. O objetivo do trabalho é fazer um levantamento de como o estudo estatístico está sendo caracterizado nos artigos deste simpósio, com vistas a trazer indicativos de como qualificar suas submissões nos próximos anos.

Além de apresentar o levantamento da apresentação dos estudos estatísticos nos artigos do WSCAD, o presente artigo também contribui por caracterizar métodos estatísticos relevantes para avaliação de desempenho em processamento de alto desempenho e indicar técnicas estatísticas aplicáveis nesta área.

Este trabalho esta dividido em 6 seções. Na Seção 2 são caracterizados trabalhos relacionados ao estudo apresentado neste artigo. Os critérios de análise quantitativa são expostos na Seção 3 e a discussão sobre a metodologia utilizada é apresentada na Seção 4. Os resultados da coleta realizadas são discutidos na Seção 5. A Seção 6 conclui o trabalho.

## 2. Trabalhos Relacionados

Após uma pesquisa bibliográfica sobre o tema, foram encontrados trabalhos relacionados desenvolvidos ao longo dos anos. Estes trabalhos dizem respeito a outros trabalhos que também tiveram como objetivo a identificação de metodologias de validação de resultados em artigos científicos da grande área de Computação. Nesta seção, os trabalhos são organizados em ordem cronológica, de forma que a evolução das pesquisas ao longo dos anos possa ser visualizada e comparada com os resultados obtidos neste trabalho.

[Prechelt 1994] avaliou 190 artigos publicados em 1993 e 1994 e evidenciou que apenas 1/3 dos artigos não tinham comparação quantitativa com técnicas previamente conhecidas.[Tichy et al. 1995] analisou 400 artigos para determinar se cientistas da computação apoiam seus resultados com avaliação experimental. O estudo descobriu que 40% dos artigos não possuía qualquer tipo de avaliação. [Wainer et al. 2009] replicou a pesquisa de [Tichy et al. 1995] analisando 147 artigos publicados no ano de 2005 concluindo que 33% dos artigos encontram-se na mesma situação. O trabalho de [Tedre and Moisseinen 2014] reforça essa afirmação e cita como causa a falta de experiência da comunidade na correta análise dos dados a fim de produzir evidências estatísticas que comprovem os resultados.

Mais recentemente, o trabalho de [Adler et al. 2015] investigou 183 artigos do IPIN (International Conference on Indoor Positioning and Indoor Navigation) e concluiu que embora em muitas das publicações houvesse alguma preocupação na avaliação dos resultados, a qualidade da descrição dos métodos de análise era pobre. Apenas 35% relatam de maneira clara não só a metodologia do experimento em si, mas o que efetivamente os resultados querem estatisticamente dizer.

Considerando os resultados nos trabalhos mencionados acima, nosso objetivo é avaliar as publicações do Simpósio de Sistemas Computacionais de Alto Desempenho

(WSCAD) quanto à forma como são descritas em seus artigos as análises estatísticas de desempenho de maneira que possa ser atestado o ganho, se houver.

## 3. Critérios de Análise Quantitativa

A pesquisa em Ciência da Computação, envolve na maioria dos casos, o desenvolvimento de um modelo, aplicação, algoritmo ou sistema computacional novo [Wainer et al. 2007], que ainda não foi totalmente estudado à exaustão como objeto de pesquisa. Dentro deste processo, o objeto de pesquisa é comparado a seus pares para efetiva avaliação de desempenho da solução proposta. Esta avaliação deve ser feita pela análise quantitativa dos resultados sumarizados, obtidos pela utilização de *dados sintéticos* e técnicas estatísticas de comparação de conjuntos de medidas [Wainer et al. 2007, Bukh 1992].

Dados sintéticos, obtidos por *workloads*, *benchmarks*, simulações e competições, são classificados em três categorias. A primeira é utilizada para avaliar o tempo de resposta de uma solução, a segunda para avaliar se uma solução consegue obter o resultado (eficácia) e a terceira para avaliar a qualidade da resposta da solução (eficiência) [Wainer et al. 2007].

Os experimentos realizados em uma solução precisam ter efetiva significância estatística, de acordo com o tipo de medição realizada e o teste estatístico correto para analisar essa medição. Os tipos de medida são classificados em categóricas ou nominais, ordinais, intervaláveis e de razão [Wainer et al. 2007]. Testes estatísticos são procedimentos usados para testar a hipótese nula, cujo pressuposto é de que não há diferença ou relação entre os grupos de dados ou eventos testados no objeto da pesquisa e que as diferenças encontradas se devem ao acaso, bem como a hipótese alternativa, na qual o pressuposto é que existem diferenças estatisticamente significantes entre as medidas.

Calculando a probabilidade da hipótese nula ser verdadeira ou não, por meio do teste adequado ao tipo de medição realizada, chega-se ao chamado valor-p ou *p-value*. Quando o nível de significância representado por este valor é inferior a um terminado indicador, sendo 0,05 (5%) o valor mais empregado, rejeita-se a hipótese nula e aceita-se hipótese alternativa, de que realmente existe a diferença e esta não foi encontrada ao acaso [Wainer et al. 2007] [Dean et al. 1999]. Ainda que o teste de hipóteses seja útil, quando se comparam valores obtidos em experimentos diferentes o teste de hipóteses não é suficiente. É necessário saber o quanto esses valores efetivamente diferem, para isso utiliza-se o chamado intervalo de confiança. Em [Montgomery 2017], este intervalo de confiança é definido em, pelo menos, 95%, o qual representa o maior e o menor valores assumíveis garantindo um p-valor de 0,05 [Wainer et al. 2007]. O intervalo de confiança não se sobrepõe, ou invalida, a medida de desvio padrão. Este último corresponde à indicação do quanto os dados do experimento podem variar em relação a média e é utilizado como parâmetro em alguns testes.

Os testes indicados, e, por consequência os mais utilizados, para comparação de até dois conjuntos de medições e obtenção do valor-p são: Teste T, Teste T Pareado, Teste U de *Mann-Whitney* ou *Wilcoxon rank-sum test*, *Wilcoxon signed-rank test*, Chi-quadrado e Teste Exato de *Fisher*. Para comparações múltiplas, com mais de dois conjuntos de valores, são: Teste ANOVA e *Kruskal-Wallis* [Wainer et al. 2007].

Sabendo que o estudo estatístico deve ser aplicado sobre uma coleção de $n$ amostras de desempenho coletadas, o problema que resta é como definir o valor de $n$ para

um determinado experimento. O Teorema do Limite Central é o resultado mais importante em estatística, do qual muitos métodos estatísticos comummente usados se baseiam para terem validade [Navidi 2012]. Este teorema diz que se for extraída uma amostra suficientemente grande o comportamento das médias tende a ser uma distribuição normal [Navidi 2012, Bukh 1992] ou gaussiana [Neto et al. 2010]. A distribuição normal (ou gaussiana), é o modelo estatístico que melhor representa o comportamento natural de um experimento, onde uma variável aleatória pode assumir qualquer valor dentro de um intervalo definido [Neto et al. 2010].

Amostra aqui refere-se às medições dos objetos de pesquisa, número de repetições ou iterações realizadas nos testes ou experimentos. Dependendo do tipo do objeto de pesquisa exitem cálculos específicos para o tamanho da amostra, porém, o teorema do limite central sugere que para a maioria dos casos uma amostra de tamanho 30 ou mais é suficientemente grande para que a aproximação normal seja adequada [Navidi 2012].

Para cada objeto de pesquisa existem critérios de medições aplicáveis. Segundo [Fortier and Michel 2003] a mensuração de desempenho pode ser classificada como medidas orientadas ao sistema ou ao usuário. As medições orientadas ao sistema transitam tipicamente no entorno da taxa de transferência (*Throughput*) e utilização. Taxa de transferência é definida como uma média por intervalo de tempo, sejam tarefas, processos ou dados. Utilização é a medida do intervalo de tempo em que um determinado recurso computacional está ocupado. Já as medições orientadas ao usuário compreendem o tempo de resposta (*response time*) e *turnaround time*.

Dentro desse conceito, é possivel perceber que métricas especializadas como *Reaction Time, Strech Fator*, MIPS (*Millions of Instructions Per Second*), MFLOPS (*Millions of Floating-Point Operations Per Second*), PPS (*packets per second*), BPS (*bits per second*), TPS (*Transactions Per Second*), *Nominal Capacity, Bandwidth, Usable Capacity, Efficiency, Idle Time, Reliability, Availability, Downtime, Uptime,* MTTF (*Mean Time To Failure*), *Cost/Performance Ratio* [Bukh 1992] [Fortier and Michel 2003] estão inseridas dentro destas duas generalizações.

A termologia das métricas é bastante extensa e variada. Pode também variar seu uso conforme o entendimento dos autores das publicações ou conforme a região. No entanto [Bukh 1992] e [Fortier and Michel 2003] são elucidativos tanto na definição, quanto no uso de cada métrica supra mencionada.

## 4. Metodologia

Para minerar os dados, todos os trabalhos publicados dos anos de 2000 a 2017 foram salvos localmente, numerados e separados por pasta segundo o ano de publicação. Trabalhos no formato resumo e aqueles escritos em língua estrangeira não fizeram parte da amostra das 426 publicações analisadas. Isto se deve ao fato de que, dada a natureza do formato resumo, certos detalhes da análise dos dados do objeto estudado poderiam ser suprimidos o que certamente geraria um viés. Bem como trabalhos em língua estrangeira aumentaria consideravelmente o número de termos de pesquisa se sua diversidade.

Em seguida, por meio de processo automatizado via software (NVivo[1]), foram pesquisadas citações de termos, utilizados em análise estatística bem como métricas e

[1] http://www.software.com.br/p/qsr-nvivo

testes utilizados para aferição de resultados, assim categorizados neste artigo. Estas categorias referem-se às boas práticas na coleta de dados e análise de resultados em pesquisa científica, conforme Tabelas 1, 2 e 3.

**Tabela 1. Termos estatísticos selecionados para coleta de dados**

| Descrição | Chave de Pesquisa |
|---|---|
| Amostra (AM) | amostra |
| Desvio Padrão (DP) | ”desvio padrão” |
| Distribuição Normal (DN) | ”distribuição normal” |
| Frequência (FR) | ”frequência”OR ”frequência” |
| Gaussiana (GA) | gaussiana |
| Intervalo de Confiança (IC) | ”intervalo de confiança” |
| Média (ME) | ”média” |
| Num. Execuções (NE) | ”número de execuções” |
| Num. Iterações (NI) | ”numero de iterações” |
| Teste/Experimento (TE) | teste OR experimento OR simulação |
| Variância (VR) | variância |

**Tabela 2. Métricas selecionadas para coleta de dados**

| Descrição | Chave de Pesquisa |
|---|---|
| Bandwidth (BW) | ”bandwidth OR ”largura de banda” |
| BPS (BP) | ”bits por segundo”OR bps” |
| Capacidade Nominal (CN) | ”nominal capacity”or ”capacidade nominal” |
| Capacidade Utilizável (CU) | ”usable capacity”or ”capacidade utilizável” |
| Confiabilidade (CO) | Reliability OR Confiabilidade |
| Cost/Perfornace Ratio (CP) | ”cost ratio”OR ”performance ratio” |
| Disponibilidade (DI) | availability OR disponibilidade |
| Downtime/Uptime (DU) | downtime OR uptime |
| Eficiência/Acurácia | (EA)eficiência OR eficácia OR accuracy |
| Fator de Estiramento | (FE)”strech factor”OR ”fator de estriamento” |
| Tempo Ocioso | (TO)”Idle time”OR ”tempo ocisoso” |
| MFLOPS (MF) | MFLOPS |
| MIPS (MI) | MIPS |
| MTTF (MT) | MTTF |
| PPS (PP) | PPS |
| Speed up (SU) | ”speedup OR speed-up OR ”speed up” |
| Tempo de Reação (TR) | reaction time or tempo de reação |
| TPS (TP) | TPS |

**Tabela 3. Testes estatísticos selecionados para coleta de dados**

| Descrição | Chave de Pesquisa |
|---|---|
| P-Valor (PV) | ”p-valor OR p-value OR ”valor p” |
| Teste ANOVA (AN) | anova |
| Teste Chi-quadrado (CH) | chi-quadrado OR qui-quadrado |
| Teste de Wilcoxon (TC) | ”wilcoxon signed-rank” |
| Teste Exato de Fisher (FI) | ”teste exato de fisher”or ”fisher” |
| Teste Kruskal-Wallis (KR) | kruskal-wallis |
| Teste T (TT) | ”teste t”OR ”teste-t”OR ”teste de student”OR ”Student” |
| Teste U (TU) | ”teste U”OR ”mann-whitney”OR ”wilcoxon rank-sum” |

Após a tabulação, os dados foram sumarizados por ano conforme a categoria do termo. As Tabelas 4, 5 e 6 mostram os resultados obtidos na pesquisa de termos estatísticos, métricas e testes respectivamente. Note que somente termos que obtiveram resultados são sumarizados, aqueles onde não houve ocorrência foram suprimidos. Ainda na Tabela 4, os resultados do termo frequência foram suprimidos devido ao viés que os resultados obtiveram. Frequência é citada como unidade de medida do *clock* de processadores. Os resultados dos termos distribuição normal e gaussiana foram sumarizados juntos por se tratarem do mesmo objeto.

**Tabela 4. Citações de Termos Estatísticos por ano**

| Ano | n | AM | DP | DN | IC | ME | NE | NI | TE | VR |
|---|---|---|---|---|---|---|---|---|---|---|
| 2000 | 7 | - | - | - | - | 2 | - | 1 | 4 | - |
| 2001 | 34 | 4 | 1 | - | - | 10 | 1 | 1 | 16 | 1 |
| 2002 | 35 | 1 | 2 | - | - | 9 | 1 | 2 | 20 | - |
| 2003 | 40 | 2 | 3 | - | - | 11 | 1 | 2 | 21 | - |
| 2004 | 53 | - | 4 | - | - | 19 | 1 | 4 | 24 | 1 |
| 2005 | 58 | 3 | 6 | - | 1 | 17 | - | 2 | 28 | 1 |
| 2006 | 38 | - | 2 | 1 | - | 7 | 1 | 3 | 24 | - |
| 2007 | 28 | - | 1 | - | - | 7 | 1 | 1 | 17 | 1 |
| 2008 | 41 | 1 | 3 | 1 | 1 | 12 | - | 1 | 22 | - |
| 2009 | 18 | - | - | - | - | 4 | - | - | 14 | - |
| 2010 | 19 | 2 | - | 1 | - | 5 | - | - | 11 | - |
| 2011 | 8 | - | - | - | - | 3 | - | - | 5 | - |
| 2012 | 33 | - | 1 | 1 | 1 | 7 | 1 | - | 21 | 1 |
| 2013 | 31 | 2 | 2 | 2 | - | 11 | - | - | 14 | - |
| 2014 | 33 | - | - | - | - | 3 | - | 2 | 27 | 1 |
| 2015 | 16 | - | 1 | 1 | 1 | 1 | - | 1 | 11 | - |
| 2016 | 23 | 1 | - | 1 | 1 | 2 | - | 1 | 16 | 1 |
| 2017 | 13 | 2 | 1 | - | - | 2 | - | - | 8 | - |

**Tabela 5. Citações de Métricas por ano**

| Ano | n | Disp. | BW | BP | CP | DU | EA | ID | MF | MI | PP | CO | SU |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2000 | 17 | 5 | 1 | - | - | - | 6 | 1 | - | 1 | - | 1 | 2 |
| 2001 | 29 | 7 | 3 | - | - | - | 9 | - | - | 1 | - | 2 | 7 |
| 2002 | 31 | 5 | 1 | - | - | - | 12 | - | - | 3 | - | 4 | 6 |
| 2003 | 38 | 11 | 2 | - | - | - | 8 | - | 3 | 2 | - | 5 | 7 |
| 2004 | 38 | 10 | - | - | - | - | 14 | - | - | 1 | - | 6 | 7 |
| 2005 | 36 | 10 | 1 | - | - | - | 9 | 1 | - | 1 | - | 4 | 10 |
| 2006 | 15 | 3 | 2 | - | - | - | 4 | - | - | 1 | - | - | 5 |
| 2007 | 17 | 3 | 1 | - | - | - | 3 | 1 | - | 2 | - | - | 7 |
| 2008 | 33 | 6 | 5 | - | - | - | 7 | - | 1 | 3 | - | 3 | 8 |
| 2009 | 23 | 6 | 1 | 1 | - | - | 4 | 1 | 1 | 1 | - | 1 | 7 |
| 2010 | 21 | 5 | - | - | - | 1 | 2 | 1 | - | 5 | - | 2 | 5 |
| 2011 | 8 | 3 | - | - | - | - | 1 | - | - | 1 | - | - | 3 |
| 2012 | 24 | 9 | 2 | 1 | - | - | 4 | - | - | 4 | - | - | 4 |
| 2013 | 31 | 5 | - | 1 | - | 3 | 5 | 1 | - | 3 | - | 2 | 11 |
| 2014 | 32 | 6 | - | - | - | - | 5 | 1 | - | 3 | - | 1 | 16 |
| 2015 | 17 | 1 | - | - | - | - | 3 | 1 | 1 | 2 | 1 | - | 8 |
| 2016 | 28 | 6 | 3 | - | 1 | - | 1 | 4 | 1 | 1 | - | 1 | 10 |
| 2017 | 22 | 5 | 1 | - | - | 1 | - | 2 | - | 1 | - | 1 | 11 |

**Tabela 6. Citações de Testes por ano**

| Ano | n | PV | TT |
|---|---|---|---|
| 2001 | 1 | 1 | - |
| 2007 | 4 | 1 | 3 |
| 2008 | 1 | - | 1 |
| 2009 | 1 | - | 1 |
| 2012 | 2 | - | 2 |
| 2015 | 1 | - | 1 |
| 2016 | 1 | - | 1 |

A Tabela 7 apresenta a sumarização dos resultados das Tabelas 4, 5 e 6, bem como a distribuição de citações por categoria do termo e ano. Nela é possível visualizar o número de artigos total no ano, o número de artigos que contém pelo menos uma ocorrência do termo. Também é identificado o número de ocorrências dentro dos artigos separados por categoria de termo.

**Tabela 7. Distribuição de citações por tipo do termo e ano**

| Ano | n | Estatística | | Métricas | | Testes | |
|---|---|---|---|---|---|---|---|
| | - | Art. | Cit. | Art. | Cit. | Art. | Cit. |
| 2000 | 9 | 5 | 7 | 7 | 17 | - | - |
| 2001 | 23 | 20 | 34 | 16 | 29 | 1 | 1 |
| 2002 | 27 | 22 | 35 | 19 | 31 | - | - |
| 2003 | 32 | 28 | 40 | 23 | 38 | - | - |
| 2004 | 33 | 30 | 53 | 21 | 38 | - | - |
| 2005 | 34 | 31 | 58 | 26 | 36 | - | - |
| 2006 | 28 | 24 | 38 | 12 | 15 | - | - |
| 2007 | 21 | 17 | 28 | 12 | 17 | 2 | 4 |
| 2008 | 28 | 24 | 41 | 17 | 33 | 1 | 1 |
| 2009 | 23 | 14 | 18 | 16 | 23 | 1 | 1 |
| 2010 | 20 | 12 | 19 | 16 | 21 | - | - |
| 2011 | 6 | 5 | 8 | 5 | 8 | - | - |
| 2012 | 28 | 22 | 33 | 17 | 24 | 2 | 2 |
| 2013 | 20 | 17 | 31 | 16 | 31 | - | - |
| 2014 | 39 | 29 | 33 | 23 | 32 | - | - |
| 2015 | 15 | 12 | 16 | 11 | 17 | 1 | 1 |
| 2016 | 21 | 16 | 23 | 17 | 28 | 1 | 1 |
| 2017 | 19 | 9 | 13 | 15 | 22 | - | - |

## 5. Discussão

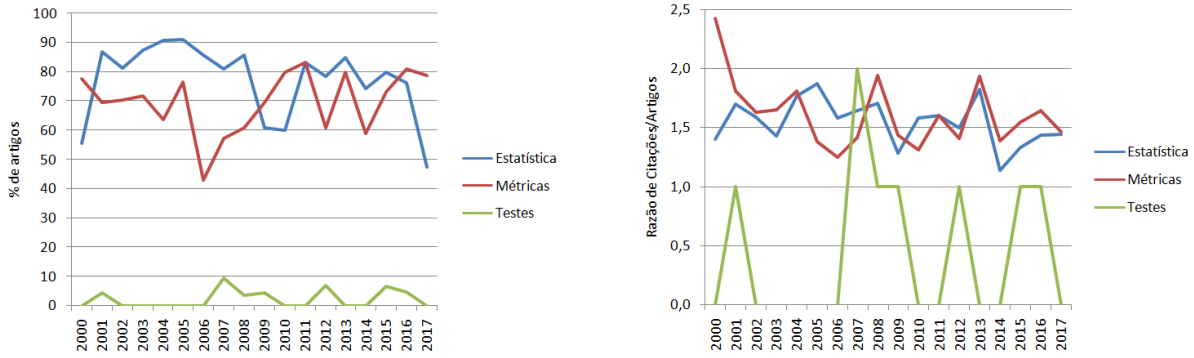
O objetivo aqui não é desmerecer ou invalidar os resultados obtidos nos experimentos realizados, uma vez que a prática permite identificar resultados coerentes com o esperado. O que se espera é verificar qual o cuidado dos autores ao publicarem esses resultados assim como a evolução da pesquisa científica uma vez que se tratam de quase 20 anos de WSCAD. Assume-se que todos os possíveis erros assim como as questões para evitá-los já foram levados em conta [Bukh 1992].

Nesta análise, foi utilizada a proporção de artigos com citações em relação ao total de artigos no ano e não o número de artigos para possibilitar a comparação entre os anos, uma vez que o número de artigos por ano não é o mesmo tendo grande variabilidade Figura 1(a). Do total de 426 artigos analisados, 79% citaram termos estatísticos, 67% métricas e apenas 2% testes estatísticos.

A razão foi obtida por meio dos dados Tabela 7, dividindo o número de artigos com citação pelo número de citações. A média de citações das três categorias foi de 1,46 citações por artigo. Individualmente observa-se que para termos estatísticos (1,57) e métricas (1,59) os valores ficaram acima da média geral em quase todos os anos.

Dos testes pesquisados apenas o Teste T ou de *Student* obteve resultado com 9 ocorrências, e, sendo que das duas ocorrências do nível de significância estatística (p-valor), nenhuma delas foi de artigos com ocorrência do Teste T.

Devido à baixa ocorrência de testes estatísticos nos resultados, em torno de 2%, resolveu-se investigar mais a fundo este dado. Para isso foi calculada uma segunda amostra dos 426 trabalhos, desconsiderando os 9 trabalhos onde já se encontrou a ocorrências




(a) Proporção de artigos com citações por ano

(b) Razão de citações por artigo ano

**Figura 1. Gráficos dos Resultados Encontrados**

destes termos, com intervalo de confiança de 95% e erro amostral de 5%, chegando a um total de 30 artigos sorteados aleatoriamente.

A segunda amostra de artigos foi encaminhada a dois revisores, que leram e analisaram a metodologia e análise dos resultados destes artigos segundo alguns critérios. Experimentação, se foi realizada ou não; Método amostral referindo-se ao método utilizado para encontrar o tamanho da amostra, pela utilização de *benchmarks*, estimação ou cálculo; Tamanho da amostra representado pelo número de experimentos realizados, repetições, instruções, jobs e afins; Se fica evidente o tamanho da amostra e se seu tamanho é adequado; Utilização de métricas; Se houve comparação com outra técnica e esta foi demonstrada adequadamente de alguma maneira.

Dos artigos analisados na segunda amostra, *nenhum* evidenciou igualidade ou deficiência em seus resultados, o pior caso apenas aponta ”indícios de melhora no desempenho”. Desta amostra, 24 deles realizaram experimentação, 6 são modelos teóricos ou memoriais descritivos de implementações de sistemas computacionais sem análise numérica de resultados. Daqueles que realizaram experimentação 10 usaram *benchmarks*, 13 estimaram o tamanho da amostra e 1 não citou números.

Os tamanhos de amostra foram considerados adequados para todos aqueles que usaram *benchmarks*, uma vez que se tratam de grandes coleções de registros e fica implícito que para cada registro de entrada há uma medição e registro de saída, também por serem um método amplamente aceito na comunidade científica [Wainer et al. 2007]. Daqueles trabalhos onde houve estimação da amostra (13) em 11 deles o tamanho da amostra foi considerado adequado, dado o tamanho amostral e em 2 foi considerado inadequado por estarem abaixo do mínimo estipulado pelo TLC [Navidi 2012]. É preciso estar atento, pois um *benchmark* pode representar a carga de entrada a que um experimento é submetido e não o montante de dados coletados, olhando por este aspecto apenas 8 trabalhos tiveram amostras adequadas com poder estatístico de análise especificado no texto.

Todos os artigos onde houve experimento (24) houve a utilização de métricas, deste total 15 compararam seus resultados a outras técnicas e 9 não o fizeram. Confirmando os resultados para utilização de métricas da primeira amostra em 70%.

Na segunda amostra apenas em 3 artigos a demonstração dos resultados foi embasada em critérios científicos (métodos estatísticos e modelos matemáticos), porém apenas 1 artigo utilizou minimamente métodos estatísticos (p-valor e DP) para confirmarem seus resultados. Percentualmente os valores para este critério de avaliação da primeira e segunda amostras são similares 2% e 3% respectivamente.

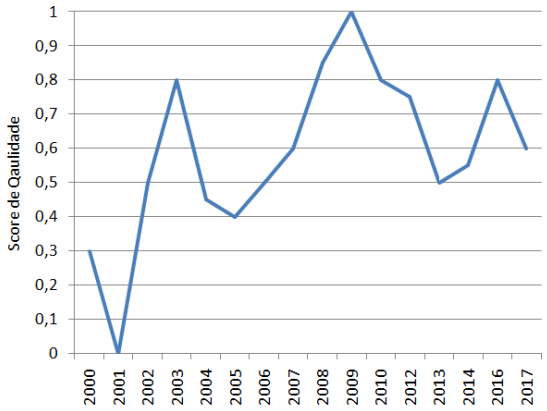



**Figura 2. Score de Qualidade por Ano**

Para calcular a evolução do WSCAD em termos de qualidade das publicações, a cada resultado positivo das categorias analisadas na segunda amostra foi atribuído peso 1 e peso zero para resultados negativos ou inexistentes. Calculados os somatórios de cada ano e as médias chegou-se em um *score* que compreende valores entre 0 e 1. A Figura 2 mostra a linha evolutiva do *score* de qualidade das publicações, evidenciando o amadurecimento do evento ao longo do tempo. Embora existam alternâncias na linha, a tendência da curva é ascendente com a maioria dos anos acima da média.

## 6. Conclusão

Neste artigo foi apresentada uma pesquisa sobre a ocorrência de termos estatísticos, métricas e testes estatísticos utilizados para comprovação dos resultados em pesquisa científica. Foram analisadas as publicações de todas as edições do WSCAD numa amostra total de 426 artigos.

Da amostra analisada 398 publicações fizeram referência a pelo menos um dos termos pesquisados, correspondendo a 93% do total. Isso mostra que existe a preocupação na realização pesquisa e na demonstração dos resultados, mesmo que inadequada ou incompleta dada a ocorrência de apenas 3% de testes estatísticos confirmada por uma segunda amostra de 30 artigos. É preciso não apenas colocar foco no desenvolvimento do objeto de pesquisa em si, mas também na aferição e apresentação dos resultados.

Quase a totalidade dos artigos apenas apresenta as medições da técnica implementada porém o questionamento é inevitável: apenas simples mensuração da métrica é suficiente para comparação entre técnicas similares, desconsiderando fatores de erro e considerando que elas foram executadas dentro dos mesmos padrões?

A pesquisa aqui descrita serve de alerta à comunidade científica. Há claramente a necessidade de atenção nos pontos destacados aqui em pesquisas. Os achados deste trabalho devem promover a reflexão sobre os cursos de graduação e pós-graduação e a neces-

sidade da inclusão da metodologia de análise estatística de dados aplicada à computação nas disciplinas básicas de formação. Neste sentido, a contribuição do WSCAD poderia se dar na direção de indicar na sua chamada de trabalhos a necessidade de que artigos apresentando análise de desempenho venham acompanhados de validação estatística de seus resultados ao mesmo tempo em que solicita aos revisores destes artigos que observem se tal estudo foi devidamente apresentado.

## Referências

Adler, S., Schmitt, S., Wolter, K., and Kyas, M. (2015). A survey of experimental evaluation in indoor localization research. In *Indoor Positioning and Indoor Navigation (IPIN), 2015 International Conference on*, pages 1–10. IEEE.

Bukh, P. N. D. (1992). The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling.

Dean, A., Voss, D., Draguljić, D., et al. (1999). *Design and analysis of experiments*, volume 1. Springer.

Fortier, P. and Michel, H. (2003). *Computer systems performance evaluation and prediction*. Elsevier.

Montgomery, D. C. (2017). *Design and analysis of experiments*. John Wiley & Sons.

Navidi, W. (2012). *Probabilidade e estatística para ciências exatas*. AMGH.

Neto, B. B., Scarminio, I. S., and Bruns, R. E. (2010). *Como Fazer Experimentos-: Pesquisa e Desenvolvimento na Ciência e na Indústria*. Bookman.

Prechelt, L. (1994). A quantitative study of experimental evaluations of neural network learning algorithms: Current research practice. *IEEE Transactions on Neural Networks*, 6.

Tedre, M. and Moisseinen, N. (2014). Experiments in computing: A survey. *The Scientific World Journal*, 2014.

Tichy, W. F., Lukowicz, P., Prechelt, L., and Heinz, E. A. (1995). Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28(1):9–18.

Wainer, J., Barsottini, C. G. N., Lacerda, D., and de Marco, L. R. M. (2009). Empirical evaluation in computer science research published by ACM. *Information and Software Technology*, 51(6):1081–1085.

Wainer, J. et al. (2007). Métodos de pesquisa quantitativa e qualitativa para a ciência da computação. *Atualização em informática*, 1:221–262.

# Otimização Computacional do Modelo BRASIL-SR

**Jefferson Gonçalves de Souza[1], Celso Luiz Mendes[1], Rodrigo Santos Costa[2]**

Instituto Nacional de Pesquisas Espaciais (INPE) – São José dos Campos – SP – Brasil

Laboratório Associado de Computação e Matemática Aplicada - LAC[1]

Centro de Ciência do Sistema Terrestre – CCST[2]

jefferson.souza@inpe.br, celso.mendes@inpe.br, rodrigo.costa@inpe.br

***Abstract.*** *This paper describes the improvement of computational performance of the BRASIL-SR model. This computational code, which is used for the estimate of incident surface solar radiation, was optimized with an application of OpenMP directives and code modifications for the implementation of data output (I/O) operations using NetCDF format files. After these implementations, it was verified that the speedup of the code was further improved with the I/O optimizations, with a more stable efficiency, which shows a better use of the processors in the underlying supercomputer.*

***Resumo.*** *Este artigo apresenta a melhoria de desempenho computacional do modelo BRASIL-SR. Este código computacional, utilizado para a estimativa da irradiação solar incidente na superficie, foi otimizado com a aplicação de diretivas de OpenMP e modificações do código para implementação de operações de entrada e saída de dados (E/S) utilizando arquivos em formato NetCDF. Após estas implementações, verificou-se que o ganho de desempenho do código foi ainda maior com um novo tratamento de E/S, gerando uma eficiência mais estável, o que mostra uma melhor utilização dos processadores no supercomputador empregado.*

## 1. Introdução

Diante da preocupação em atender a demanda energética mundial sem aumentar as emissões de gases do efeito estufa, diversos países têm buscado soluções relacionadas com a inserção de fontes limpas em suas matrizes energéticas. A geração solar tem crescido vertiginosamente no Brasil, principalmente depois dos déficits hídricos registrados nos últimos anos em função da variabilidade climática. Condições de estiagem afetam a geração de energia elétrica, já que os reservatórios podem vir a atingir níveis muito baixos; isso também levanta questões relativas ao conflito do uso da água, que também é utilizada no abastecimento e na agricultura. Há uma necessidade iminente da diversificação da matriz elétrica brasileira, o que minimizaria problemas relacionados à segurança hídrica, energética e alimentar. Também é importante que esta diversificação ocorra no sentido da inserção de fontes renováveis, com baixa emissão de gases.

O potencial solar brasileiro é considerado elevado, uma característica de regiões tropicais. Além disso, a baixa variabilidade intra-anual permite que as regiões brasileiras com menor capacidade de geração produzam mais energia que as regiões de maior potencial da Alemanha, por exemplo.

Antes da implantação de empreendimentos solares, é necessário conduzir estudos de avaliação de viabilidade e potencial do recurso. Utilizar dados de estações meteorológicas é a fonte mais segura para se obter o conhecimento do potencial local de energia solar, entretanto, pesquisas indicam que os erros de interpolação de dados observados em estações de superfície afastadas em mais de 30km entre si, são superiores aos erros de estimativas produzidas por modelos de transferência radiativa, como pode ser observado na Figura 1. Assim, a utilização de códigos computacionais que realizam estimativas precisas a respeito do potencial solar no Brasil é o melhor modo de se obter dados confiáveis.

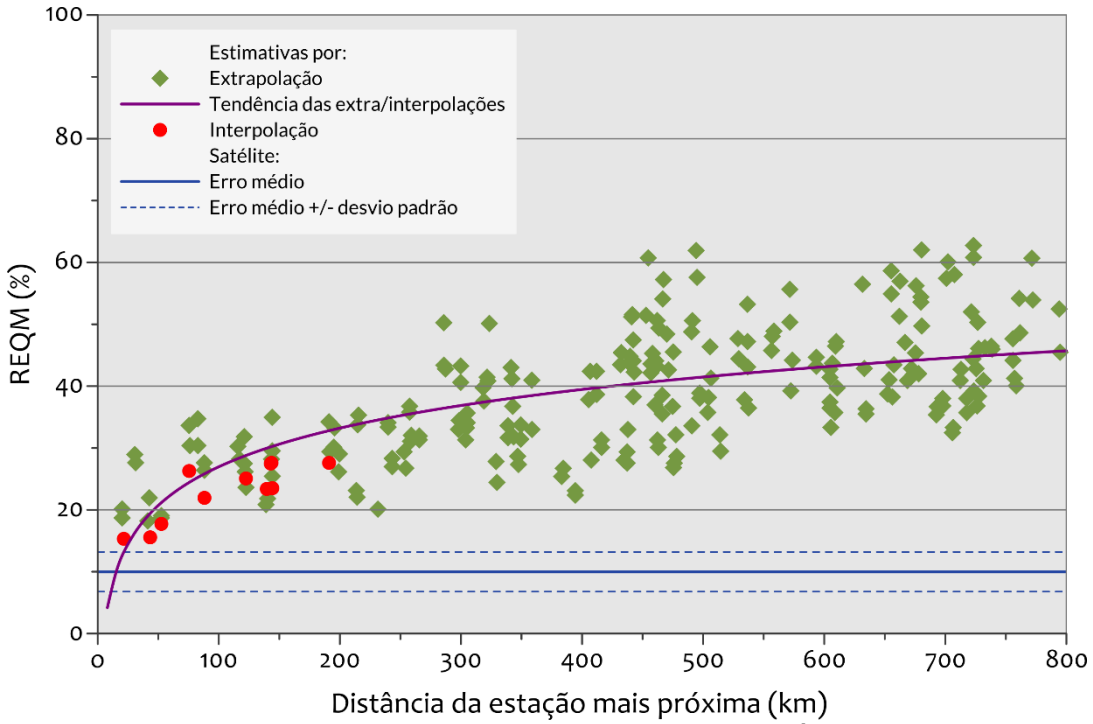



**Figura 1. Incerteza típica nos dados interpolados de irradiância solar e dados obtidos através de modelos satelitais.**
**Fonte: Martins, 2011**

Assim, a utilização de códigos computacionais (modelos) que realizam estimativas precisas a respeito do potencial solar é o melhor modo de se obter dados confiáveis. Vários modelos foram desenvolvidos de modo a obter estimativas da irradiação solar como (Dave e Canosa, 1974) que utilizaram harmônicos esféricos, (Liou, 1976) utilizando ordenadas discretas, (Lenoble, 1985) com as equações de Monte Carlo e Diferenças Finitas. Uma alternativa aos modelos citados é a utilização de métodos que geram resultados aproximados como o método de dois fluxos (Two-Stream) (Lenoble, 1985).

O modelo utilizado neste trabalho, BRASIL-SR, é um modelo físico para obtenção de estimativas da radiação solar incidente na superfície, que combina a utilização da aproximação de dois fluxos na solução da equação de transferência radiativa com o uso de informações climatológicas e parâmetros determinados a partir de imagens de satélites. Contudo, com um custo de execução serial de quase 27 horas em computadores atuais, o modelo se torna inviável para, por exemplo, realizar previsões de curtíssimo prazo ou mesmo para realização de estimativas diárias da irradiação solar. Este tipo de informação tem uso em diversos setores, tais como no planejamento de operações de sistemas

fotovoltaicos visando um melhor aproveitamento do recurso, gerenciamento de despacho em linhas de transmissão e realização de manutenção preventiva em painéis solares com uma menor perda na geração de energia, sendo inclusive de uso estratégico.

Assim, neste trabalho, buscou-se a melhoria do desempenho computacional relacionado ao tempo de processamento do modelo de transferência radiativa BRASIL-SR, que devido ao seu alto custo computacional, possui um elevado tempo de processamento em processadores convencionais. Na primeira fase deste estudo, foi realizada uma análise do desempenho da versão original do modelo – que é executada em modo serial – com o objetivo de verificar quais são os processos mais custosos computacionalmente. Posteriormente, foi utilizada a inserção das diretivas de OpenMP, permitindo a criação e o gerenciamento automático de *threads*, nas regiões importantes do código, alteração dos dados de entrada e saída (E/S), para o formato NetCDF, e a utilização da técnica de blocagem para tentar reduzir ainda mais o tempo de processamento.

## 2. Metodologia

O modelo BRASIL-SR foi desenvolvido com base no modelo alemão IGMK Forschungszentrum (Stuhlmann, 1990) e posteriormente adaptado para as condições brasileiras pelo LABREN/CCST/INPE e a Universidade Federal de Santa Catarina – UFSC (Pereira, 1996), sendo o principal recurso deste grupo a quantificação do recurso energético solar. Neste trabalho, foram utilizadas 665 imagens do mês de abril de 2016, com resolução temporal de 30 minutos e resolução espacial de 0.03º em longitude e 0.05º de latitude, o que corresponde aproximadamente a 3km x 5km do ponto nadir do satélite.

O código, contendo mais de 8300 linhas, está escrito em linguagem Fortran 90 e foi separado em 7 módulos a fim de facilitar qualquer tipo de aperfeiçoamento ou alteração de algum cálculo do modelo. No módulo 1 é calculada a transmitância em céu claro, no módulo 2 a transmitância em céu parcialmente nublado, no módulo 3 é realizado o cálculo das Irradiações Global, Difusa e Direta, no módulo 4 da Irradiação Par, no módulo 5 a integral diária da irradiação Par, no módulo 6 a integral diária das Irradiações Global, Direta, Difusa e Plano Inclinado e no módulo 7 as médias mensais das irradiações Global, Direta, Difusa, Par e Plano Inclinado (Pereira, 2017).

O modelo está sendo executado no supercomputador Santos Dumont, localizado em Petrópolis, RJ, no Laboratório Nacional de Computação Cientifica – LNCC. Para a compilação dos códigos do modelo é necessário carregar as variáveis de ambiente e a inclusão no path dos programas e bibliotecas de forma modular utilizando o aplicativo "*module*". Foi utilizada a biblioteca NetCDF 4.4.1 e o compilador Intel ifort 16.0.2 em ambiente computacional RedHat Linux 6.4 (LNCC, 2017).

Deste supercomputador, será utilizado os processadores Intel Xeon com 12 núcleos por unidade de processamento e cada nó computacional têm duas unidades de processamento tendo assim um total de 24 núcleos por nó. No supercomputador há um total de 12.096 núcleos multi-core, não sendo contabilizados os nós computacionais com os processadores Intel Xeon Phi e GPU NVidia. A capacidade de armazenamento principal é de 1,7 Pbytes, com sistema secundário de 640 Tbytes, e sistema operacional Linux (LNCC, 2018).

O modelo, em sua forma original, realiza suas operações de E/S em formato binário e texto. Também foi avaliada a implementação destas operações a partir de arquivos em formato NetCDF (Network Commom Data Form) que é uma especificação desenvolvida pela fundação UNIDATA e oferecida sob a forma de bibliotecas utilitárias, visando o armazenamento de grandes volumes de dados. Além disso, um dos objetivos do formato é o acesso eficiente a pequenos subconjuntos de grandes conjuntos de dados, fazendo uso do acesso direto em vez de acesso sequencial, sendo muito mais eficiente quando a ordem em que os dados são lidos é diferente da ordem em que foram gravados (UNIDATA, 2018).

A paralelização do modelo foi realizada utilizando as diretivas de OpenMP (Open Multi-Processing), que é uma interface de programação multi processo de memória compartilhada em múltiplas plataformas (OpenMP, 2018). Esta forma de paralelização foi escolhida devido a dois fatores principais: (a) permite a paralelização gradativa do código sequencial original, através de alterações sucessivas e independentes em trechos do código; e (b) grande popularização atual de processadores multi-núcleo (multi-core), com custo moderado, que podem ser facilmente empregados pelas instituições interessadas em executar o modelo no futuro próximo. Uma paralelização mais ampla, para ambientes de memória distribuída, através do uso de um paradigma de troca de mensagens (por exemplo com MPI), é uma possibilidade viável para desenvolvimentos futuros.

O OpenMP foi desenvolvido pelo grupo OpenMP Architecture Review Board (ARB) – que é formado pelos maiores fabricantes de programas e componentes eletrônicos do mundo (OpenMP, 2017). O OpenMP trabalha com arquitetura *multicore* e *multithread*, conseguindo assim executar simultaneamente duas ou mais tarefas utilizando *threads*. Cada *thread* possui sua própria pilha de execução que compartilha o mesmo espaço de memória com outros *threads* do mesmo processo. Por utilizar diretivas, chamadas de sentinelas, que identificam a área que será paralelizada, o OpenMP se torna uma ferramenta muito importante quando se tem um código complexo onde há a necessidade de paralelização, com diretivas simples existem poucas mudanças necessárias no código, fazendo com que até códigos mais antigos tenham uma implementação com o OpenMP de maneira mais fácil do que utilizando MPI.

Para se paralelizar um código com OpenMP o início da área a ser paralelizada começa com uma diretiva do tipo *!$omp parallel*, como pode ser observado abaixo.

```
!$omp parallel
        write(*,*)"Olá"
!$omp end parallel
```

A diretiva *!$omp private()* é importante quando se precisa especificar uma variável que terá seu uso especifico em cada *thread*. Assim, durante a execução a variável, caso seja um índice de um laço, terá um valor (posição) em cada *thread*, garantindo que o OpenMP não utilize uma posição que está sendo utilizada em outro *thread*.

Quando no código existe uma região onde há a necessidade de ser executada por um *thread* por vez, podemos utilizar a diretiva *!$omp critical*, assim quando um *thread* estiver executando a região critica os outros *threads* irão bloquear a execução quando alcançarem essa região, e cada uma executará a região de acordo com a ordem de chegada.

A paralelização, neste trabalho, foi realizada em cada um dos 7 módulos que compõem o modelo. O primeiro módulo é o mais custo computacionalmente tanto na versão original como na versão com dados em NetCDF, utilizando apenas 1 thread, o tempo de processamento é maior que 60 mil segundos. E o módulo 7 é o que tem menor tempo de processando, um pouco mais de 10 segundos nas duas versões do modelo.

Nos dois primeiros módulos a paralelização ocorreu no laço principal que é o responsável por chamar todas as funções que calculam as transmitâncias. Os módulos 3 e 4 foi paralelizado a região onde são chamadas as funções responsáveis por calcular as irradiações. E nos módulos 5, 6 e 7 foi paralelizado a região onde são realizadas as integrações diárias e as médias mensais das irradiações solares.

Para realizar a paralelização dos módulos foi utilizado a diretiva *!$omp parallel do private(i,j)* como pode ser observado na Figura 2. A otimização dos laços *DO* foi realizada invertendo para que a linha seja percorrida e junto os valores próximos possam ser acessados e trazidos para a memória *cache* melhorando assim o acesso a memória e melhorando o tempo de processamento.

```
  !$omp parallel do private(I,J)
   DO J=1,1180
    DO I=1,1784
      IF(XALT(I,J).GT.-98.0) THEN
            if (DMACC(I,J) < SSIW(I,J))then
           SSIW(I,J) = DMACC(I,J)
            endif
  CALL TRANSMIT(XLAT(J),XALB(I,J),XUMI(I,J),XALT(I,J),SSIW(I,J),TCLEAR(I,J),TDIR(I,J))
  CALL TRANSMIT(XLAT(J),XALB(I,J),XUMI(I,J),XALT(I,J),SSIW(I,J),TCLOUD(I,J),TTDIR)
      ELSE
       TCLEAR(I,J)  = -1.0
       TCLOUD(I,J) = -1.0
       XTDIR(I,J)     = -1.0
      ENDIF
      END DO
END DO
    !$omp end parallel do
```

**Figura 2. Código do módulo 1 com a diretiva de OpenMP**

Como os arquivos de E/S estão em formatos binário e texto fazendo com que a leitura e escrita consumam um tempo de processamento grande, foi realizada a conversão desses arquivos para NetCDF, melhorando assim o tempo de processamento e facilitando a visualização dos dados dos arquivos, já que os arquivos em NetCDF contém um *dataset* com informações de dimensão, variáveis e atributos, todas possuindo um nome e um número de identificação pelos quais podem ser referenciadas. Na Figura 3 podemos observar como é o *dataset* do NetCDF e como os dados ficam disposto nos arquivos em NetCDF. No *dataset* é definido o tamanho das matrizes e o tipo de informação que existe dentro do NetCDF.

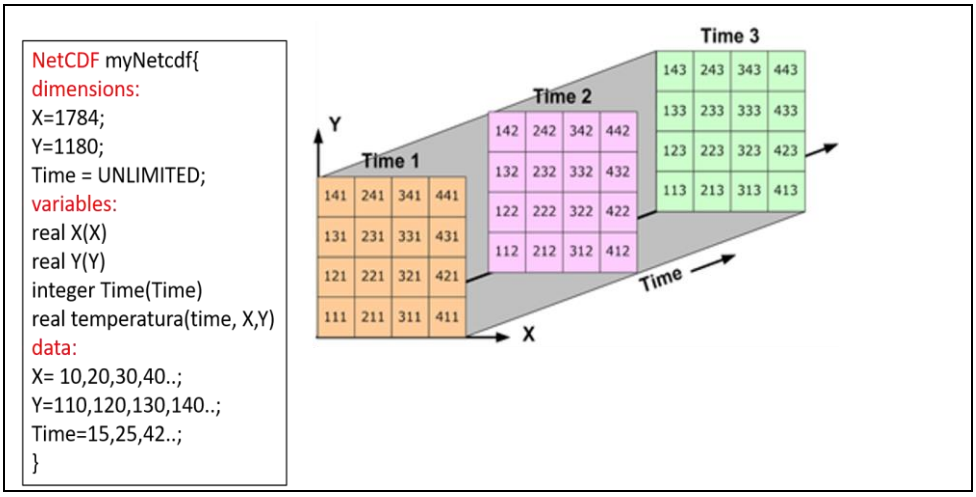



**Figura 3. Dataset do NetCDF e disposição dos dados**

Neste trabalho foi utilizada a técnica de blocagem, para tentar melhorar ainda mais os resultados de tempo de execução do modelo BRASIL-SR. A técnica de blocagem se baseia em agrupar os elementos adjacentes da matriz, onde as colunas adjacentes serão lidas ao mesmo tempo. Esse método é eficiente desde que a quantidade de dado lido seja capaz de ficar armazenado em *cache*. Assim, ao criar um bloco, deve se garantir que o bloco seja o maior possível minimizando o *overhead* dos novos laços e ser pequeno o bastante para ser armazenado na *cache*.

Com a blocagem, pequenos trechos são criados e alocados na memória *cache*. Os *strideI* e *strideJ* são os responsáveis pelo tamanho do bloco a ser obtido e que será armazenado na *cache* e a função *min* é utilizada para verificar se o laço chegou ao final.

Para realizar os testes com blocagem no modelo, tendo que a resolução das imagens de satélite é de 1784x1180, foi escolhido o máximo divisor comum entre eles, gerando assim matrizes quadradas de tamanho 4x4, como pode ser observado na Figura 4.

```
ni = 1784
nj = 1180
stridei = 4
stridej = 4
 do kj=1,nj,stridej
  do ki=1,ni,stridei
   do J=kj,min(nj,kj+stridej-1)
    do I=ki,min(ni,ki+stridei-1)
  CALL TRANSMIT(XLAT(J),XALB(I,J),XUMI(I,J),XALT(I,J),SSIW(I,J),TCLEAR(I,J),TDIR(I,J))
  CALL TRANSMIT(XLAT(J),XALB(I,J),XUMI(I,J),XALT(I,J),SSIW(I,J),TCLOUD(I,J),TTDIR)
         ELSE
            TCLEAR(I,J)  = -1.0
            TCLOUD(I,J) = -1.0
            XTDIR(I,J)     = -1.0
         ENDIF
       enddo
      enddo
     enddo
    enddo
```

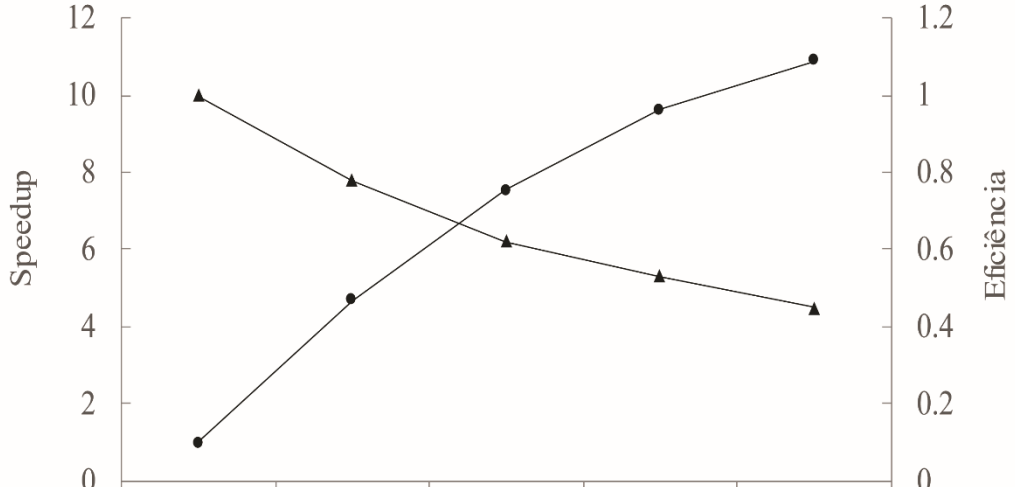
**Figura 4. Código do modelo utilizando blocagem**

Para avaliar o desempenho do modelo BRASIL-SR são utilizadas duas medidas, o ganho de desempenho e a eficiência. O ganho de desempenho é a relação entre o tempo da execução sequencial e o tempo da execução paralela. A equação para se obter essa medida é $S_p = T_i/T_p$ onde o $T_i$ é o tempo da execução sequencial e o $T_p$ é o tempo da execução paralela. A eficiência é a fração de tempo que os processadores estão sendo utilizados em processamento e é calculada levando em conta o número de processadores $p$ e o ganho de desempenho, através da equação $n_p = S_p/p$, sendo que a eficiência ideal é 1. (Pacheco,1996)

## 3. Resultados e Discussões

Depois do processo de inserção das diretivas de OpenMP, foi verificado a existência de uma chamada de arquivo nos dois primeiros módulos. Foi inserida a diretiva *!$omp critical* para que aquela região fosse executada com apenas um *thread* por vez. Visto que a região estava ficando serial com a diretiva, o seu conteúdo foi alocado em um vetor e passado por parâmetro pelas funções.

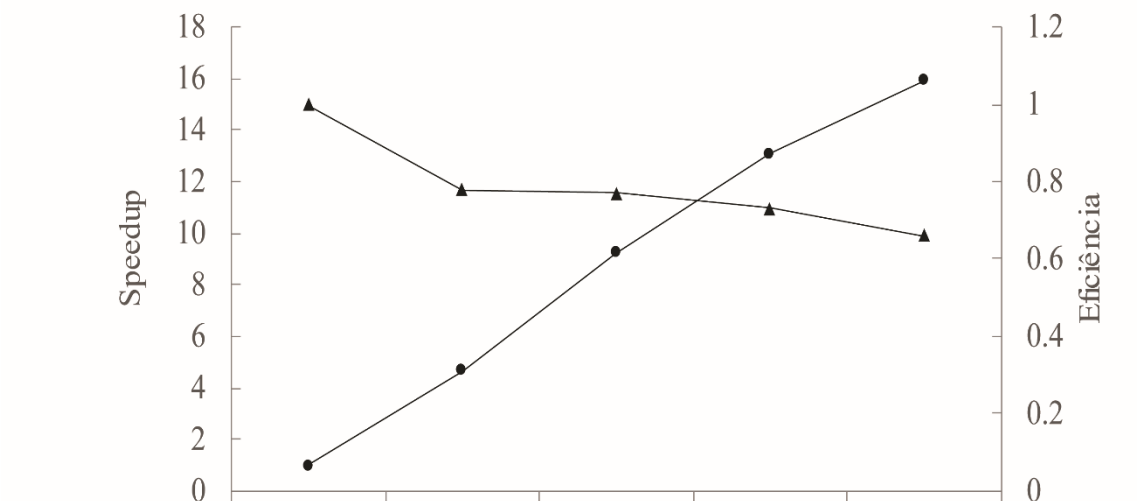
Na Figura 6, podemos observar que o tempo de processamento melhorou à medida que a quantidade de *threads* foi aumentando no modelo BRASIL-SR original. Com um ganho de desempenho máximo de 10,9 e uma eficiência moderada (0,45), nota-se que o modelo não tem uma eficiência paralela muito boa. Isto é devido, em parte, à influência dos trechos seriais originais do modelo que não puderam ser cobertos pelas diretivas OpenMP inseridas, e assim limitaram o ganho de desempenho da paralelização, ou ainda aos efeitos dos processos de leitura/escrita de dados.




| | 1Thread | 6Threads | 12Threads | 18Threads | 24Threads |
|---|---|---|---|---|---|
| Speedup | 1 | 4.7 | 7.54 | 9.63 | 10.9 |
| Eficiência | 1 | 0.78 | 0.62 | 0.53 | 0.45 |
| Tempo (s) | 96637 | 20540 | 12811 | 10036 | 8863 |

**Figura 6. Tempo (segundos), Ganho de desempenho e Eficiência do Modelo BRASIL-SR**

Com a versão do modelo contendo os dados em NetCDF, o tempo total de processamento diminuiu ainda mais. Na Figura 7, podemos observar a melhora do desempenho e a eficiência, até 12 *threads*, se mantendo constante e a eficiência com 24 *threads* melhorando em relação ao modelo original.




| | 1Thread | 6Threads | 12Threads | 18Threads | 24Threads |
|---|---|---|---|---|---|
| Speedup | 1 | 4.66 | 9.25 | 13.08 | 15.9 |
| Eficiência | 1 | 0.78 | 0.77 | 0.73 | 0.66 |
| Tempo (s) | 90700 | 19469 | 9808 | 6935 | 5703 |

**Figura 7. Tempo (segundos), Ganho de Desempenho e Eficiência do Modelo BRASIL-SR utilizando dados de E/S em formato NetCDF**

Em comparação com o modelo original, podemos observar que o ganho de desempenho e a eficiência no modelo com dados em NetCDF se mostraram melhores.

Com os testes realizados, podemos afirmar que o modelo com E/S em formato NetCDF e utilizando 24 *threads* tem uma melhora de 37% no tempo comparado com sua versão serial, e melhora de 41% se comparado com o código original serial. O tempo de processamento de E/S sem NetCDF e com 24 threads é de 4444 segundos e com NetCDF é de 2728 segundos, mostrando uma melhora em 61% no tempo de E/S. O ganho de desempenho e eficiência também melhoraram com essa alteração. A possível razão para esta melhora é que os arquivos em NetCDF têm acesso direto podendo ser recuperados sem passar por outros dados na sequência, sendo importante quando a execução do modelo é realizada utilizando mais de um *thread*. Além das imagens de satélite e dos dados climatológicos de entrada, os dados de saída somam 4100 imagens.

A técnica de blocagem foi utilizada no modelo original e na versão com dados em NetCDF, os tempos obtidos ficaram muito próximos ou maiores em relação aos tempos obtidos nas Figuras 7 e 8. Neste teste foi definindo valores de *stride* igual 4, mas o modelo não apresentou melhoras, (veja Tabela 1). A utilização de diferentes valores de *stride* para a blocagem é um caminho a ser potencialmente explorado em estudos futuros.

**Tabela 1. Tempo (segundos) da execução com a técnica de blocagem**

| | 1T | 6T | 12T | 18T | 24T |
|---|---|---|---|---|---|
| **Modelo Original** | **96742** | **20591** | **12168** | **10519** | **9427** |
| **Modelo NetCDF** | **91135** | **19869** | **9246** | **7168** | **6131** |

## 4. Conclusão

Neste artigo, foram apresentados resultados iniciais da paralelização do modelo de radiação solar BRASIL-SR, através de OpenMP. Este paradigma de paralelização foi escolhido, principalmente, para permitir aos usuários atuais do modelo a exploração de processadores multo-core, amplamente disponíveis atualmente. Com isto, a utilização do modelo poderia atender satisfatoriamente aos requisitos operacionais de tempo de execução hoje existentes. A extensão deste trabalho para ambientes de memória distribuída, via paralelização com MPI, é uma possibilidade para trabalhos futuros.

A paralelização do modelo BRASIL-SR, utilizando OpenMP, mostrou uma redução considerável no seu tempo de processamento com 24 *threads.* Posteriormente, serão utilizados mais *threads*, num nó computacional com maior número de núcleos, a fim de avaliar mais extensivamente a escalabilidade do desempenho e eficiência do modelo. Além disso, a implementação relativa às operações de E/S em formato NetCDF mostrou uma melhora adicional, com uma melhor eficiência paralela.

Nos testes realizados com blocagem, em que não obtivemos uma melhora no tempo de processamento, mostra-se necessária uma maior exploração com diferentes valores de *stride*, de modo a variar a quantidade de dados que são efetivamente armazenados em *cache*. Simultaneamente, deve ser conduzida uma concreta medição das taxas de acerto/erro em *cache*, de modo a caracterizar precisamente quais valores de *stride* são mais vantajosos.

Esses testes apresentados são parte do trabalho que está sendo realizado para a redução do tempo de processamento do modelo BRASIL-SR, e uma implementação futura contemplará a exploração da técnica de vetorização de laços, dado que boa parte dos cálculos realizados no modelo são efetuados dentro de laços

## 5. Referências

Dave, J. V.; Canosa, Z. A direct solution of the radiative transfer equation: application to atmospheric models with arbitrary vertical non-homogenities. Journal of Atmospheric Science, 31,1089-1101, 1974.

Laboratório Nacional de Computação Científica (LNCC), “Manual Supercomputador Santos Dumont”, Disponível: http://sdumont.lncc.br/support_manual.php?pg=support# Acesso: Janeiro/2018

Lenoble, J. Radiative transfer in scattering and absorbing atmospheres: standard computational procedures. Virginia: A. Deepak Publishing, 1985.

Liou, K. N. On the absorption, reflection and transmission of solar radiation in cloudy atmospheres. Journal of Atmospheric Science, 33,798-805, 1976.

Martins, F. R., Pereira, E. B., "Estudo Comparativo da confiabilidade de estimativas de irradiação solar para o sudeste brasileiro obtidas a partir de dados de satélite e por interpolação/extrapolação de dados de superfície". Revista Brasileira de Geofísica, São Paulo, v.29, p.265-276, 2011

OpenMP, "Parallel Programming in Fortran 95 using OpenMP", Disponível: https://www.openmp.org/wp-content/uploads/OpenMP-WelcomeGuide.pdf Acesso: Janeiro/2017

OpenMP, "Parallel Programming in Fortran 95 using OpenMP", Disponível: https://www.openmp.org/wp-content/uploads/F95_OpenMPv1_v2.pdf Acesso: Janeiro/2018

Pacheco, Peter S., "Parallel Programming With MPI". Morgam Kaufmann Publisher Inc. San Francisco, CA, USA, p.418, 1996, ISBN 1-55860-339-5

Pereira, E.B., Martins, F.R, Gonçalves, A.R., Costa, R.S., Lima, J.L., Rüther, R., Abreu, S.L., Tiepolo, G.M., Pereira, S.V. and Souza, J.G., "Atlas Brasileiro de Energia Solar", 2.ed. São José dos Campos: INPE, 2017. 88p. ISBN 978-85-17-00090-4, 2017

Pereira, E.B., Abreu, S.L., Stuhlmann, R., Rieland, M. and Colle, S. "Survey of The Incident Solar Radiation In Brazil by use of Meteosat Satellite Data", Solar Energy, Phoenix, v. 57, n.2, p.125-132, 1996.

Stuhlmann, R., Rieland, M. and Raschke, E. "An Improvement of the IGMK model to derive total and difuse solar radiation at the surface from satellite data", J. Applied Meteorology, v29, n. 7, p.586-603, 1990.

UNIDATA, "NetCDF 4.6.1", Disponível: https://www.unidata.ucar.edu/software/NetCDF/docs/NetCDF_introduction.html Acesso: Janeiro/2018

# Otimizando uma Aplicação de Geofísica com Mapeamento de Threads e Dados*

**Matheus S. Serpa[1], Eduardo H. M. Cruz[2], Jairo Panetta[3], Philippe O. A. Navaux[1]**

[1] Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970, Porto Alegre – RS – Brasil

{msserpa, navaux}@inf.ufrgs.br

[2]Instituto Federal do Paraná (IFPR) Paranavaí – PR – Brasil

eduardo.cruz@ifpr.edu.br

[3]Divisão de Ciência da Computação – Instituto Tecnológico de Aeronáutica (ITA)
São José dos Campos – SP – Brasil

jairo.panetta@gmail.com

***Abstract.*** *Many software mechanisms for geophysics exploration in Oil & Gas industries are based on wave propagation simulation. To perform such simulations, state-of-art HPC architectures are employed, generating results faster at each generation. It is therefore of paramount importance to optimize their operation on modern processors. In this paper, we study the impact of thread and data mapping strategies, revealing that, with smart mapping policies, one can indeed significantly speed up these simulations on modern architectures. We reduced the execution time by up to 33.5% and 58.6% on Intel's multi-core Xeon and many-core accelerator Xeon Phi Knights Landing, respectively.*

***Resumo.*** *Muitos mecanismos de software para exploração de geofísica nas indústrias de petróleo e gás são baseados na simulação. Para executar tais simulações, arquiteturas HPC de última geração são utilizadas, gerando resultados mais rápidos a cada geração. Por conseguinte, é de suma importância otimizar a sua operação em processadores modernos. Neste artigo, estudamos o impacto de estratégias de mapeamento de threads e dados, revelando que, com políticas de mapeamento inteligentes, pode-se acelerar significativamente essas simulações em arquiteturas modernas. Reduzimos o tempo de execução em até 33,5% e 58,6% no processador Intel Xeon e no acelerador Intel Xeon Phi Knights Landing, respectivamente.*

## 1. Introdução

A geofísica de exploração continua a ser fundamental para o mundo moderno, buscando acompanhar a demanda por recursos energéticos. Este esforço resulta em altos custos de perfuração. Assim, as indústrias de petróleo e gás dependem de *softwares* focados em computação de alto desempenho (HPC) para reduzir riscos e tornar a perfuração economicamente viável.

*Este trabalho foi parcialmente financiado pela Intel sobre o projeto Modern Code e Petrobras 2016/00133-9.

Normalmente, essas aplicações são aceleradas através de ambientes baseados em *Unidades Gráficas de Processamento* (GPUs). No entanto, o uso de sistemas baseados em *Unidades Centrais de Processamento* (CPUs) também pode ser interessante por sua flexibilidade, capacidade de memória e escalabilidade. Com os novos processadores *many-core*, as CPUs podem ser uma escolha melhor para grandes cargas de trabalho paralelas, que anteriormente eram dominadas por GPUs [Witten et al. 2016].

Os processadores *many-core* geralmente são definidos como processadores construídos com um grande número de núcleos com menor poder de processamento. Eles oferecem maior vazão para aplicações paralelas, aliado a uma melhor eficiência energética. As arquiteturas *many-core* estão presentes em três dos dez melhores computadores da lista Top-500 em Junho de 2018 [J. Dongarra and Strohmaier 2018]. O segundo computador mais poderoso do mundo é o Sunway TaihuLight, que usa 40960 processadores *manycore* SW26010, cada um com 260 núcleos.

Com centenas de *threads* executando simultaneamente no mesmo sistema *many-core*, o *overhead* relacionado à sincronização, por exemplo, na manutenção da coerência de *cache*, representa um impacto ainda maior no desempenho. *Threads* em aplicações *multithreaded*, como aquelas que implementam algoritmos de geofísica, compartilham dados, forçando-os a serem movidos através das interconexões *inter-chip* ou *intra-chip*, dependendo da hierarquia da memória e da localização das *threads* [Diener et al. 2016].

*Threads* que se comunicam intensamente tem latência de comunicação menores se colocadas próximas no *chip*. Por outro lado, a largura de banda também é compartilhada, o que significa que as *threads* competirão por isso. Nos sistemas NUMA (*Non-uniform memory access*) os dados presentes na memória principal podem ser locais ou remotos, dependendo do nó onde estão armazenados e da CPU que executa a *thread* que está acessando. Ler dados de um banco de memória remoto resulta em latências mais altas e mais tráfego nas interconexões.

Este artigo pretende mostrar que uma aplicação de geofísica pode ser acelerada graças ao mapeamento de *threads* e dados. Nós avaliamos as políticas de mapeamento de *threads* e dados mais comuns e verificamos seu desempenho em dois tipos de CPUs: Intel Xeon e Intel Xeon Phi Knights Landing. O artigo está organizado da seguinte forma. A Seção 2 discute os trabalhos relacionados. A Seção 3 mostra o potencial da aplicação geofísica para o mapeamento. A Seção 4 apresenta o potencial do mapeamento nas arquiteturas discutidas no artigo. A Seção 5 descreve as arquiteturas, a aplicação geofísica e as técnicas que avaliamos. A Seção 6 fornece uma avaliação de desempenho de diferentes estratégias de mapeamento e, finalmente, a Seção 7 apresenta conclusões e trabalhos futuros.

## 2. Trabalhos relacionados

Arquiteturas recentes, incluindo aceleradores e coprocessadores, provaram ser adequadas para geofísica, simulações de hidrodinâmica e fluxo magnético, superando os processadores de propósito geral. Kukreja et al. [Kukreja et al. 2016] gera automaticamente um código estêncil altamente otimizado para várias arquiteturas, enquanto Niu et al. [Niu et al. 2014] sugere o uso de um modelo de desempenho para reduzir o consumo de recursos. Caballero et al. [Caballero et al. 2015] estudou o efeito de diferentes otimizações nas equações de propagação de ondas elásticas.

Vários trabalhos identificam o mapeamento de *threads* e dados como forma efetiva de melhorar o desempenho de aplicações paralelas e propõem novos métodos para executar o mapeamento de forma mais eficiente. Mazouz et al. [Mazouz et al. 2011] analisa a aceleração do conjunto de *benchmarks* SPEC OMP em diferentes máquinas que utilizam várias estratégias de mapeamento de *threads*, alcançando um desempenho significativamente melhor do que o padrão do sistema operacional em uma máquina NUMA. No entanto, não foram avaliadas aplicações de geofísica.

Em Cruz et al. [Cruz et al. 2016], é apresentado um método que usa o tempo que uma entrada permanece na TLB (*translation lookaside buffer*) e as *threads* que acessam essa página como uma métrica para executar o mapeamento de *threads* e dados, obtendo excelentes melhorias de desempenho com baixa sobrecarga. Diener et al. [Diener et al. 2016] propõe o *kMAF*, um *framework* implementado diretamente no *kernel* que usa faltas na tabela de páginas para perfilar seu padrão de acesso à memória e melhorar o mapeamento em tempo de execução. He et al. [He et al. 2016] apresenta o NestedMP, uma extensão para o OpenMP que permite que o programador forneça informações sobre a estrutura da árvore de tarefas para o *runtime*, que então executa um mapeamento de *threads*. Esses trabalhos propõem mecanismos para aprimorar o mapeamento de *threads* e dados, mas não levam em conta a memória MCDRAM do Xeon Phi Knights Landing.

Liu et al. [Liu et al. 2015] utiliza *profiling* para determinar o mapeamento de *threads* na arquitetura *Knights Corner* que depende da localização do diretório de *tags* distribuídas, conseguindo reduções significativas na latência de comunicação. Tousimojarad e Vanderbauwhede [Tousimojarad and Vanderbauwhede 2014] mostram que o mapeamento de *threads* padrão do Linux é ineficiente quando o número de *threads* é grande como em um processador *many-core* e apresentam uma nova política de mapeamento de *threads* que usa a quantidade de tempo que cada núcleo faz de trabalho útil para encontrar o melhor núcleo para alocar cada *thread*. Os autores não realizaram experimentos em arquiteturas *many-core*, como o Xeon Phi Knights Landing.

Em comparação com os trabalhos relacionados, este artigo analisa as implicações de desempenho das técnicas de mapeamento de *threads* e dados em um algoritmo de geofísica para as arquiteturas Intel Xeon e Intel Xeon Phi Knights Landing.

## 3. Potencial de mapeamento para uma aplicação geofísica

A aplicação geofísica é usada para modelagem da propagação de ondas em meio anisotrópico. Este tipo de aplicação paralela executa muitos acessos à dados devido a característica do padrão *stencil*. Assim, o mapeamento de *threads* e dados é uma técnica útil para otimizar o desempenho dessas aplicações. Nesta seção, analisamos como o mapeamento pode melhorar o desempenho da aplicação geofísica.

### 3.1. Comportamento dos acessos à memória para o mapeamento de dados

Para o mapeamento de dados, precisamos saber a quantidade de acessos à memória de cada *thread* (ou nó NUMA) para cada página. Com esta informação, cada página pode ser mapeada para o nó NUMA que executa a maioria dos acessos à memória para eles. O objetivo desta técnica é reduzir a quantidade de acessos à memória remota, que são mais custosos do que os acessos à memória local.

Para analisar a quantidade de acessos a dados privados e compartilhados, usamos a métrica chamada *nível de exclusividade* [Diener et al. 2014]. O nível de exclusividade de uma aplicação é maior quando a quantidade de acessos a dados privados é maior que a quantidade de acessos a dados compartilhados. O nível de exclusividade de uma página é calculado usando a Equação 1, onde $Acessos(p,t)$ é a quantidade de acessos à memória de uma *thread* $t$ para a página $p$, $Acessos(p)$ é a quantidade de acessos à memória de cada *thread* para a página $p$ e $N_T$ é o número total de *threads*. A menor exclusividade que uma página pode ter é quando todas as *threads* acessam a página a mesma quantidade de vezes. A maior exclusividade de uma página é quando uma página é acessada apenas por uma *thread*, ou seja, a página é privada.

$$ExcPag(p) = 100 \cdot \frac{max(Acessos(p))}{\sum_{t=1}^{N_T} Acessos(p,t)} \tag{1}$$

No contexto deste trabalho, avaliamos uma aplicação geofísica, a qual é apresentada em mais detalhes posteriormente. Na Figura 1a, mostramos a quantidade de páginas de acordo com seu nível de exclusividade na aplicação. O eixo *X* da Figura é o tamanho de *N* de uma entrada $N \times N \times N$ utilizada. As aplicações com as maiores quantidades de páginas com alta exclusividade são aqueles com o maior potencial de melhorias de desempenho com o mapeamento de dados considerando a localidade. Portanto, mapeando essas páginas para o nó NUMA mais próximo das *threads* que mais acessam, podemos reduzir a quantidade de acessos à memória remota para estas *páginas*. Apenas as páginas que contêm o nível de exclusividade mais baixo não são afetadas pelo mapeamento considerando a localidade, de modo que o mapeamento considerando o balanceamento de carga é mais apropriado neste caso. No entanto, como podemos observar na Figura 1a, muitas páginas têm um alto nível de exclusividade.

Embora a Figura 1a mostre uma visão detalhada das páginas, pode ser difícil ter uma visão geral de toda a aplicação. Para calcular o nível de exclusividade de toda a aplicação, podemos usar a Equação 2 ou a Equação 3 [Diener et al. 2014]. A Equação 2 calcula o nível médio de exclusividade de todas as páginas da aplicação. A Equação 3 também calcula o nível de exclusividade médio de todas as páginas, mas ponderado para a quantidade de acessos à memória para cada página. As páginas com mais acessos à memória têm maior influência no nível de exclusividade da aplicação. A Equação 3 fornece uma melhor visão geral da aplicação, porque a quantidade de acessos à memória para cada página pode variar significativamente.

$$ExclPags = \frac{\sum_{p=1}^{N_P} ExcPag(p)}{N_P} \tag{2}$$

$$ExclAcessos = \frac{\sum_{p=1}^{N_P} \sum_{t=1}^{N_T} Acessos(p,t) \cdot ExcPag(p)}{\sum_{p=1}^{N_P} \sum_{t=1}^{N_T} Acessos(p,t)} \tag{3}$$

O nível de exclusividade da aplicação geofísica é mostrado na Figura 1b. Mostramos o nível de exclusividade usando ambas Equações 2 e 3. As aplicações com níveis de exclusividade elevados tendem a se beneficiar mais com o mapeamento de dados. Isso

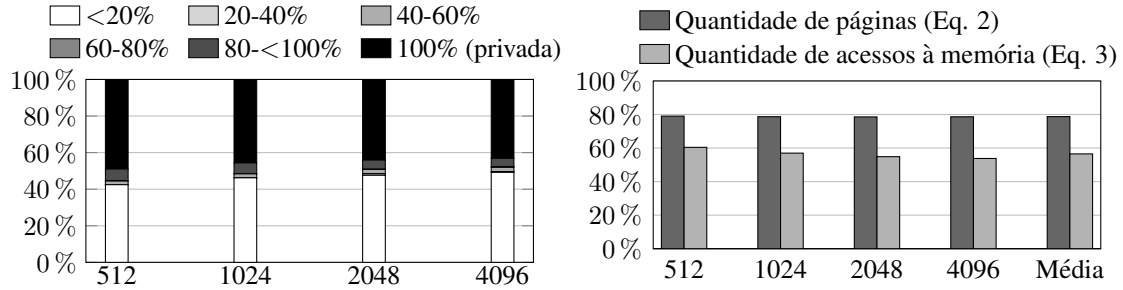



**Figura 1: Exclusividade da aplicação geofísica.**

ocorre porque, se uma página com alta exclusividade é mapeada para o nó NUMA das *threads* que mais acessam a página, a quantidade de acessos à memória remota é reduzida. Por outro lado, se uma página tiver um nível de exclusividade baixo, a quantidade de acessos à memória remota seria semelhante, independentemente do nó NUMA usado para armazenar a página.

### 3.2. Padrão de compartilhamento para mapeamento de *threads*

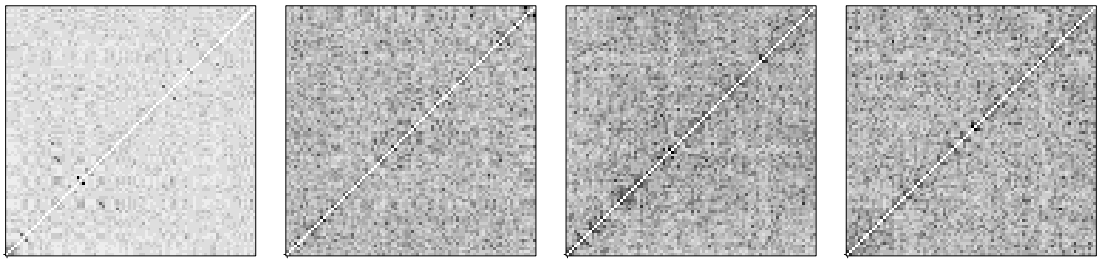
Para o mapeamento de *threads*, a informação mais importante é a quantidade de acessos a dados compartilhados e entre quais *threads* os dados são compartilhados. O endereço de memória dos dados não é relevante. Para analisar a localidade da memória no contexto do mapeamento de *threads*, devemos investigar como as *threads* de uma aplicação compartilham dados. Esta análise é baseada em matrizes de padrão de compartilhamento [Cruz et al. 2016]. As matrizes de compartilhamento da aplicação geofísica executando no Xeon são apresentadas na Figura 2, onde os eixos representam IDs das *threads* e células mostram a quantidade de acessos a dados compartilhados para cada par de *threads*, onde células mais escuras indicam mais acessos.

As matrizes de compartilhamento foram geradas usando dados coletados por instrumentação da aplicação usando a ferramenta Pin [Bach et al. 2010]. O código de instrumentação monitora todos os acessos à memória, acompanhando quais *threads* acessam cada bloco de memória. As matrizes de compartilhamento têm diferentes padrões de acordo com o tamanho da entrada da aplicação geofísica. Para o tamanho 512, a comunicação é menor se comparada com os outros tamanhos. Para os outros, existem padrões irregulares e all-to-all de comunicação.

## 4. Impacto da topologia da arquitetura no mapeamento

A diferença na localidade de memória entre os núcleos afeta o desempenho do compartilhamento de dados. As *threads* que acessam uma grande quantidade de dados compartilhados devem ser mapeadas para núcleos próximos uns dos outros na hierarquia de memória, enquanto os dados devem ser mapeados para o nó NUMA executando as *threads* que os acessam [Diener et al. 2016]. Desta forma, a *localidade* dos acessos à memória é melhorada, o que aumenta o desempenho e a eficiência energética. Idealmente, o mapeamento de *threads* e dados deve ser realizado em conjunto [Diener et al. 2016].

Para ilustrar como a hierarquia de memória influencia a localidade da memória, as Figuras 3a e 3b mostram as arquiteturas Broadwell e Knights Landing, onde exis-



**(a) 512** **(b) 1024** **(c) 2048** **(d) 4096**

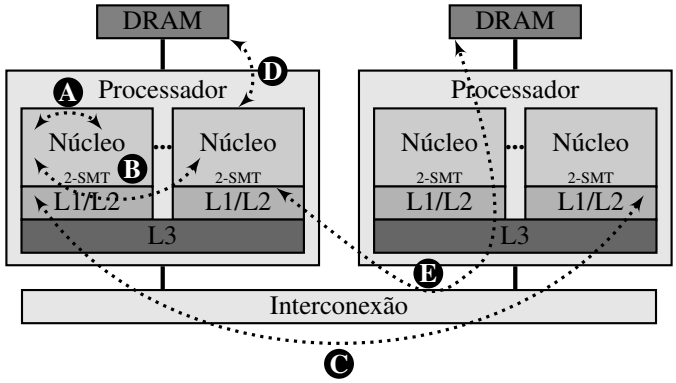
**Figura 2: Matrizes de compartilhamento da aplicação geofísica. Os eixos representam IDs de *threads*. As células mostram a quantidade de acessos aos dados compartilhados para cada par de threads. As células mais sombrias indicam mais acessos.**

tem três possibilidades de compartilhamento entre as *threads*. *Threads* executando no mesmo núcleo no Broadwell (Figura 3a Ⓐ) podem compartilhar dados através das *caches* rápidas L1 ou L2 e ter o melhor desempenho do compartilhamento. No Knights Landing (Figura 3b Ⓐ) as *threads* compartilham dados através das *caches* L1. *Threads* que são executadas em diferentes núcleos no Broadwell (Figura 3a Ⓑ) têm que compartilhar dados através da *cache* L3, a qual é mais lenta, mas ainda podem se beneficiar da rápida interconexão *intra-chip*. No Knights Landing (Figura 3b Ⓑ), a diferença é que elas compartilham dados através da *cache* L2. Quando as *threads* que executam no Broadwell compartilham dados entre processadores físicos em diferentes nós NUMA (Figura 3a Ⓒ), elas precisam usar a interconexão *inter-chip*, a qual é lenta. Isso também acontece no Knights Landing quando *threads* compartilham dados em diferentes *tiles* (Figura 3b Ⓒ). Portanto, o desempenho do compartilhamento no caso Ⓒ é o mais lento em ambas as arquiteturas.

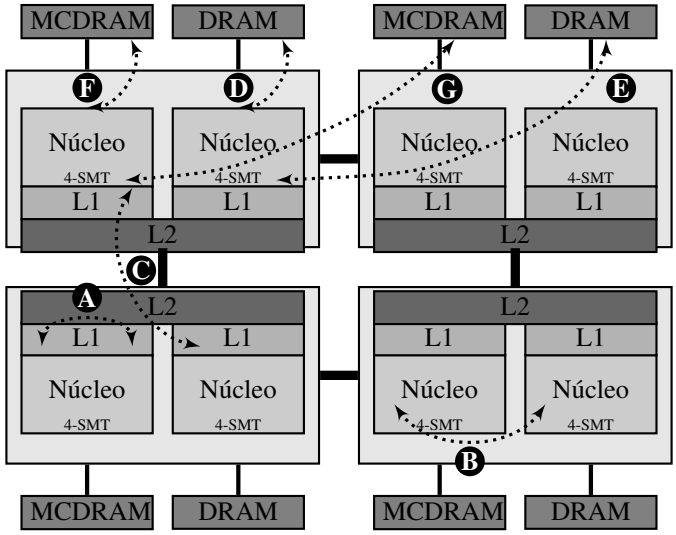
Nos sistemas NUMA, o tempo para acessar a memória principal depende do núcleo que solicitou o acesso à memória e o nó NUMA que contém a página de memória de destino [Diener et al. 2016]. Se o núcleo e a página de memória de destino pertencerem ao mesmo nó, temos um acesso à memória *local*, como nas Figuras 3a Ⓓ e 3b Ⓓ. Por outro lado, se o núcleo e a página de memória de destino pertencerem a nós NUMA diferentes, temos um acesso à memória *remota*, como nas Figuras 3a Ⓔ e 3b Ⓔ. No Knights Landing, existem bancos de memória MCDRAM, que são mais rápidos que os bancos de memória DRAM. Portanto, se o núcleo e o MCDRAM pertencem ao mesmo nó, temos um acesso MCDRAM local, como na Figura 3b Ⓕ. Caso contrário, temos um acesso MCDRAM remoto, como na Figura 3b Ⓖ. Os acessos à memória local são mais rápidos do que os acessos à memória remota. Ao mapear as *threads* e os dados da aplicação de forma a aumentar o número de acessos à memória local em comparação aos acessos remotos à memória, a latência média da memória principal é reduzida.

## 5. Metodologia de avaliação

Nesta seção, mostramos como avaliamos o mapeamento com a aplicação geofísica. Apresentamos as arquiteturas, a aplicação geofísica e as políticas de mapeamento avaliadas.




**(a) Arquitetura Broadwell (Xeon).**




**(b) Arquitetura Knights Landing (KNL).**

**Figura 3: Hierarquia de memória das arquiteturas.**

### 5.1. Arquiteturas *multi-core* e *many-core*

Utilizamos dois ambientes para analisar o impacto do mapeamento de *threads* e dados. (1) Utilizamos uma arquitetura Broadwell de 2 nós, onde cada nó consiste em um processador Intel Xeon E5-2699 v4 de 22 núcleos. Cada núcleo suporta *2-way Simultaneous Multithreading* (SMT) e possui *caches* privadas L1 e L2, enquanto a *cache* L3 é compartilhada entre todos os núcleos do processador. Nós nos referimos a este sistema como *Xeon*. (2) Utilizamos um Intel Xeon Phi 7250 de 68 núcleos da arquitetura Knights Landing. Ele suporta um *4-way SMT*, onde cada núcleo possui uma *cache* privada L1 e uma L2 compartilhada. Ele também possui uma memória MCDRAM de 16 GB que é mais rápida do que a DRAM. Nós nos referimos a este sistema como *Knights Landing*.

Cada experimento foi executado 30 vezes com o número de *cores* virtuais de cada arquitetura (88 e 272, respectivamente). Os gráficos mostram os valores médios de tempo de execução e os intervalos de confiança de 95% segundo a distribuição t de Student [Ott and Longnecker 2015].

### 5.2. Aplicação geofísica

Ao decorrer do projeto Petrobras 2016/00133-9, um programa de modelagem da propagação de ondas em meio anisotrópico foi escrito. A dedução das equações diferenciais parciais, tal como condições de fronteira e estabilidade, podem ser encontradas em Fletcher et al. [Fletcher et al. 2009]. A discretização foi feita utilizando diferenças finitas. O código foi escrito em linguagem *C* e paralelizado com *OpenMP*.

A modelagem simula a coleta de dados em um levantamento sísmico. De tempos em tempos, equipamentos acoplados ao navio emitem ondas que refletem e refratam em mudanças de meio no subsolo. Eventualmente essas ondas voltam à superfície do mar, sendo coletadas por microfones específicos acoplados a cabos rebocados pelo navio. O conjunto de sinais recebidos por cada fone ao longo do tempo constitui um traço sísmico. Para cada emissão de ondas, gravam-se os traços sísmicos de todos os fones do cabo. O navio continua trafegando e emitindo sinais ao longo do tempo.

### 5.3. Políticas de mapeamento de dados e *threads*

O objetivo do mapeamento é melhorar o uso de recursos, organizando *threads* e dados de acordo com uma política fixa, onde cada abordagem pode aprimorar aspectos diferentes. Algumas técnicas se concentram em melhorar a localidade, reduzir as faltas de *cache*, acessos à memória remota e tráfego em interconexões *inter-chip*, enquanto outros procuram distribuir a carga de forma uniforme entre os núcleos e controladores de memória.

As seguintes políticas de mapeamento de *threads* foram avaliadas:
**Mapeamento de *threads* - Padrão (*baseline*)** O mapeamento de *threads* padrão do Linux, focado no balanceamento de carga.
**Mapeamento de *threads* - *Compact*** O mapeamento de *threads Compact* organiza *threads* vizinhas para núcleos mais próximos de acordo com a hierarquia de memória.
**Mapeamento de *threads* - *Scatter*** O mapeamento de *threads Scatter* distribui as *threads* o mais uniformemente possível em todo o sistema, o que é o oposto do *Compact*.
**Mapeamento de *threads* - *RoundRobin*** O mapeamento de *threads RoundRobin* distribui as *threads* para os núcleos em uma ordem de 0 para o número de núcleos menos 1.

As seguintes políticas de mapeamento de dados foram avaliadas:
**Mapeamento de Dados - Padrão (*baseline*)** O mapeamento de dados padrão do Linux, a política de mapeamento de dados *first-touch*, onde a página é mapeada para o nó NUMA do primeiro núcleo que acessou a página.
**Mapeamento de Dados - NUMA Balancing** O mapeamento de dados *NUMA Balancing* [Corbet 2012] migra páginas ao longo da execução para o nó NUMA da última *thread* que acessou a página, que é detectada através da introdução de faltas de página.
**Mapeamento de Dados - Interleave** O mapeamento de dados *interleave* distribui páginas consecutivas para nós NUMA consecutivos. Na arquitetura Knights Landing, devido a essa ter nós DRAM e MCDRAM, também avaliamos as políticas *interleave DRAM* e *interleave MCDRAM*, que distribuem páginas apenas para nós DRAM ou MCDRAM.

Combinamos algumas técnicas de mapeamento de *threads* e dados para mostrar que, em conjunto, melhoram o desempenho ainda mais.

## 6. Resultados de Desempenho

Esta seção apresenta os resultados de diferentes técnicas de mapeamento de *threads* e dados na aplicação geofísica, seguindo a metodologia mostrada na Seção 5. Apresentamos os resultados do mapeamento de *threads*, mapeamento de dados e mapeamento de *threads* e dados para o Xeon e Knights Landing.

### 6.1. Mapeamento para Máquina Xeon

A Figura 4 mostra os resultados de mapeamento de *threads* para a máquina Xeon. Mostramos a redução do tempo de execução para as técnicas RoundRobin, Compact e Scatter em comparação com o mapeamento de *threads* padrão do Linux. Nestes experimentos, executamos as aplicações usando 88 *threads*, que é o número padrão de *threads* no Xeon.

Usando o mapeamento de *threads* sozinho, os melhores resultados foram alcançados para o tamanho 1024, reduzindo o tempo de execução em 4,5% usando um mapeamento de *threads scatter*. A média das reduções no tempo de execução foram 2,7% para *round robin*, 2,8% para *compact* e 2,8% para *scatter*. Para os tamanhos menores *scatter* obteve o melhor desempenho e para os maiores *round robin*. Em aplicações com padrões irregulares de acessos à memória, o mapeamento de *threads* pode melhorar o desempenho reduzindo o número de migrações de *threads* não necessárias.

A Figura 5 mostra os resultados de mapeamento de dados para a máquina Xeon. Analisamos as técnicas *interleave* e *NUMA Balancing* em comparação com o mapeamento de dados *first-touch*. O desempenho do mapeamento de dados foi melhor do que o mapeamento de *threads*. Isso acontece porque, mesmo que as aplicações não compartilhem muitos dados, cada *thread* ainda precisa acessar seus dados privados. A melhor redução no tempo de execução foi de 31,3% para o mapeamento de dados *interleave* para o tamanho 512. Essa aplicação tem um número significante de páginas compartilhadas, as quais quando distribuídas entre os nós pela técnica *interleave*, melhoram o balanceamento de carga entre os controladores de memória. O mapeamento *NUMA Balancing* tem os piores resultados devido à alta sobrecarga da migração das páginas, uma vez que as *threads* podem ser migradas para outros nós NUMA ao longo da execução.

A Figura 6 mostra os resultados para o mapeamento de *threads* e dados juntos na máquina Xeon. Combinamos ambas as técnicas porque, na maioria das aplicações, a eficácia do mapeamento de dados depende do mapeamento de *threads*. Mostramos todas as possibilidades de combinação de mapeamento de *threads* e dados. As melhores reduções foram de 33,5% para um mapeamento de dados *interleave* com qualquer um dos mapeamentos de *threads*. O mapeamento de dados *interleave* sozinho foi 18% pior pois o mapeamento de *threads* padrão migra *threads* durante a execução, diferente das políticas *round robin*, *compact* e *scatter* que fixam as *threads* em um determinado núcleo.

### 6.2. Mapeamento para Máquina Knights Landing

A Figura 7 mostra os resultados do mapeamento de *threads* para a máquina Knights Landing. As técnicas são as mesmas que as analisadas na máquina Xeon. Para os experimentos nesta seção, executamos a aplicação usando 272 *threads*, o que é o padrão na arquitetura Knights Landing. O melhor resultado para o mapeamento de *threads* foi uma redução de 2% no tempo de execução para o tamanho 512 usando um mapeamento de *threads round robin*. Para os outros tamanhos de entrada, o mapeamento de *threads* sozinho teve uma influência pequena no desempenho.

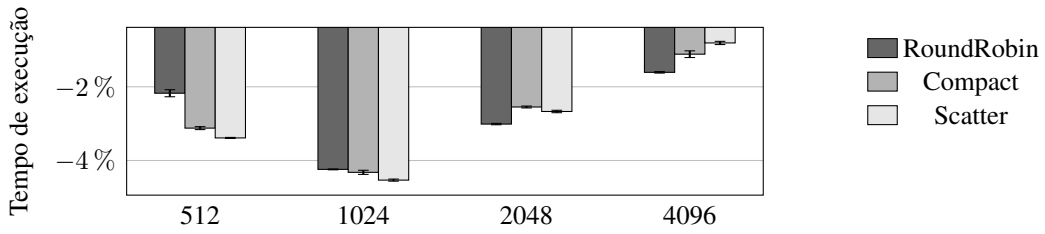



**Figura 4: Mapeamento de *threads* no Xeon.**

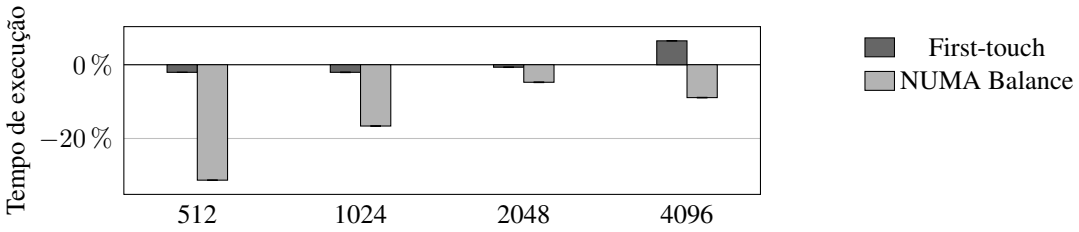



**Figura 5: Mapeamento de dados no Xeon.**

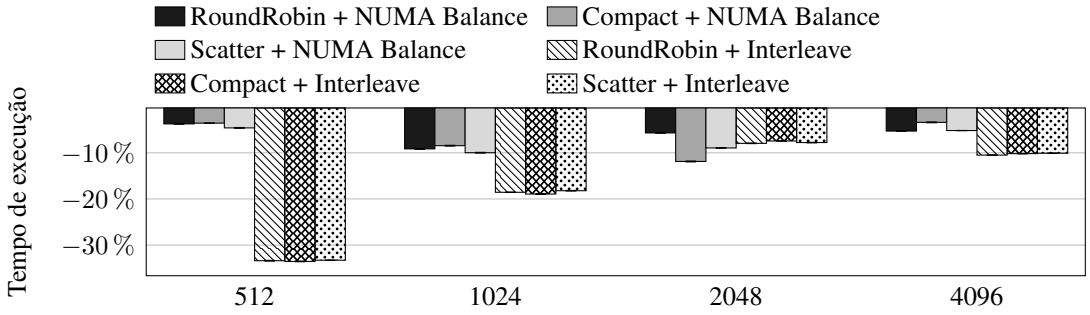



**Figura 6: Mapeamento de *threads* e dados no Xeon.**

A Figura 8 mostra os resultados do mapeamento de dados para a máquina Knights Landing. Para esta máquina, analisamos as mesmas técnicas que no Xeon mais *interleave* usando apenas DRAM ou MCDRAM. Também incluímos resultados usando o MCDRAM configurado no modo de memória cache. A diferença é que, no modo *cache*, a memória MCDRAM funciona como uma cache para a DRAM. A melhor redução foi de 57,4% para o tamanho 2048 com um mapeamento *interleave com MCDRAM*.

A Figura 9 mostra os resultados do mapeamento de *threads* e dados para a máquina Knights Landing. Mostramos um subconjunto das combinações possíveis, pois todas as combinações gerariam muitas barras. Selecionamos os mapeamentos de *threads round robin* e *scatter*, e combinamos com o mapeamento de dados *interleave com MCDRAM*. Também mostramos resultados usando MCDRAM como memória cache.

O melhor resultado foi uma redução de 58,6% para um mapeamento de *threads scatter* e o mapeamento de dados *interleave MCDRAM*. Para tamanhos maiores de 2048, os melhores resultados foram utilizando a memória MCDRAM como um último nível de *cache*. Podemos observar que, no Xeon e no Knights Landing, os maiores ganhos aconteceram geralmente com políticas que se concentram no balanceamento de carga, como o mapeamento de *threads scatter* e o mapeamento de dados *interleave*.

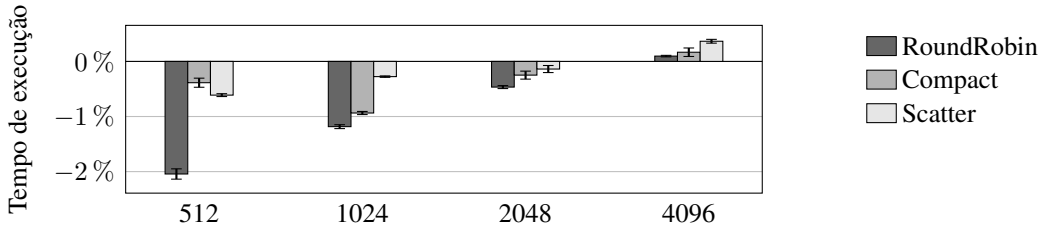



**Figura 7: Mapeamento de *threads* no Knights Landing.**

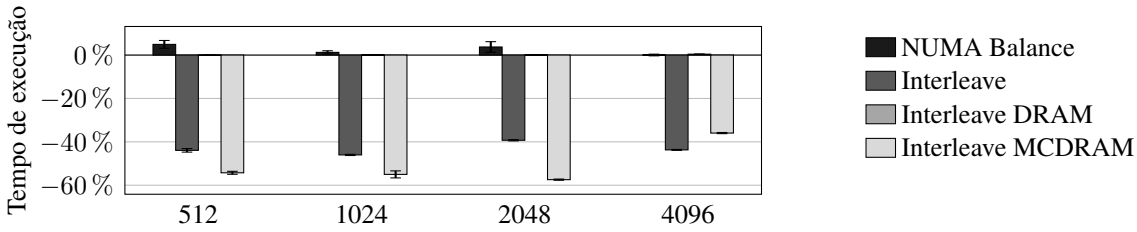



**Figura 8: Mapeamento de dados no Knights Landing.**

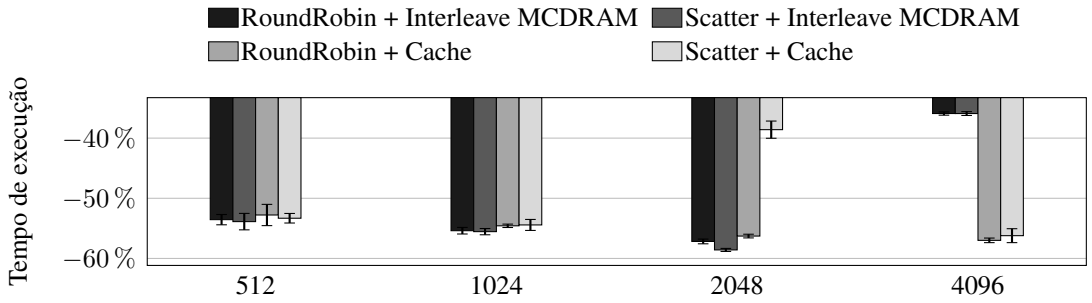



**Figura 9: Mapeamento de *threads* e dados no Knights Landing.**

## 7. Conclusão e Trabalhos Futuros

Avaliamos diferentes políticas de mapeamento de *threads* e dados para uma aplicação de geofísica. Os experimentos foram realizados em arquiteturas Intel Xeon e Intel Xeon Phi KNL. O tempo de execução foi reduzido em até 33,5% e 58,6%, respectivamente. Essas reduções foram alcançadas para o mapeamento de *threads scatter* com o mapeamento de dados *interleave*. Isso mostra que, para a aplicação geofísica, as políticas que se concentram no balanceamento de carga são melhores. Em ambos Intel Xeon e Intel Xeon Phi KNL, observamos que as políticas que se concentraram na melhoria da localização dos acessos à memória obtiveram melhores resultados. Para o Intel Xeon Phi KNL, mostramos que o uso da MCDRAM oferece melhorias no desempenho.

Como trabalho futuro, pretendemos modificar o código-fonte dos algoritmos para permitir uma granularidade de mapeamento de grão fino. Outras possibilidades incluem políticas adaptativas.

## Referências

Bach, M., Charney, M., Cohn, R., Demikhovsky, E., Devor, T., Hazelwood, K., Jaleel, A., Luk, C.-K., Lyons, G., Patil, H., and Tal, A. (2010). Analyzing Parallel Programs with

Pin. *IEEE Computer*, 43(3):34–41.

Caballero, D., Farrés, A., Duran, A., Hanzich, M., Fernández, S., and Martorell, X. (2015). Optimizing Fully Anisotropic Elastic Propagation on Intel Xeon Phi Coprocessors. In *2nd EAGE Workshop on HPC for Upstream*.

Corbet, J. (2012). Toward better NUMA scheduling.

Cruz, E. H., Diener, M., Alves, M. A., Pilla, L. L., and Navaux, P. O. (2016). Lapt: A locality-aware page table for thread and data mapping. *Parallel Computing*, 54:59–71.

Diener, M., Cruz, E. H., Alves, M. A., Navaux, P. O., Busse, A., and Heiss, H.-U. (2016). Kernel-based thread and data mapping for improved memory affinity. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2653–2666.

Diener, M., Cruz, E. H. M., Navaux, P. O. A., Busse, A., and Heiß, H.-U. (2014). kMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

Fletcher, R. P., Du, X., and Fowler, P. J. (2009). Reverse time migration in tilted transversely isotropic (tti) media. *Geophysics*, 74(6):WCA179–WCA187.

He, J., Chen, W., and Tang, Z. (2016). Nestedmp: Enabling cache-aware thread mapping for nested parallel shared memory applications. *Parallel Computing*, 51:56–66.

J. Dongarra, H. M. and Strohmaier, E. (2018). Top500 supercomputer: June 2018. `https://www.top500.org/lists/2018/06/`. [Acesso em: 10 set. 2018].

Kukreja, N., Louboutin, M., Vieira, F., Luporini, F., Lange, M., and Gorman, G. (2016). Devito: Automated fast finite difference computation. In *Intl. Workshop on Domain-Spec. Lang. and High-Level Frameworks for HPC*, WOLFHPC '16.

Liu, G., Schmidt, T., Dömer, R., Dingankar, A., and Kirkpatrick, D. (2015). Optimizing thread-to-core mapping on manycore platforms with distributed tag directories. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*.

Mazouz, A., Barthou, D., et al. (2011). Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 273–279. IEEE.

Niu, X., Jin, Q., Luk, W., and Weston, S. (2014). A Self-Aware Tuning and Self-Aware Evaluation Method for Finite-Difference Applications in Reconfigurable Systems. *ACM Trans. on Reconf. Technology and Systems*, 7(2).

Ott, R. L. and Longnecker, M. T. (2015). *An introduction to statistical methods and data analysis*. Nelson Education.

Tousimojarad, A. and Vanderbauwhede, W. (2014). An efficient thread mapping strategy for multiprogramming on manycore processors. *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Advances in Parallel Computing*, 25.

Witten, I. H., Frank, E., Hall, M. A., and Pal, C. J. (2016). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

# Aspectos de desempenho e consumo de energia ao migrar sistemas *big-data* para a nuvem

**Nestor D. O. Volpini**[1,2], **Guilherme M. Balzana**[1], **Dorgival Guedes**[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG

{nestor,guimaluf,dorgival}@dcc.ufmg.br

[2]Centro Federal de Educação
Tecnológica de Minas Gerais (CEFET-MG) – Divinópolis, MG

nestor@div.cefetmg.br

***Resumo.*** *As áreas de processamento de dados massivos (PDM, ou big-data) têm tirado proveito da disponibilidade de computação oferecida em nuvens. Aplicações processadas em ambientes virtualizados têm se tornado comum. Visto que datacenters são responsáveis por demandar algo próximo de* $2\%$ *da energia global, entender o impacto causado pela forma de virtualizar sobre as aplicações PDM se faz necessário, assim como acompanhar o custo disso sobre o consumo de energia. Nesse trabalho, foram avaliadas duas políticas de escalonamento de recursos para processar big-data em três tamanhos diferentes de máquinas virtuais. Os experimentos evidenciaram que o tempo de execução (desempenho) e o consumo de uma tarefa PDM podem apresentar relevantes discrepâncias ainda que contando com um mesmo conjunto de recursos.*

***Abstract.*** *Massive data processing areas (big-data) have taken advantage on the availability of cloud computing. Applications processed in virtualized environments have become usual. Since datacenters are responsible for demanding around of 2 % of the global energy, understand the impact of virtualization on big-data applications is necessary, as well its energy consumption. In this work, we evaluated two resources scheduling policies to process big-data in three different sizes of virtual machines. Our experiments showed that execution time (performance) and consumption of a big-data tasks can result in relevant discrepancies even with the same resources budget.*

## 1. Introdução

A demanda por processar grandes volumes de dados, também conhecidos como *big-data*, tem apresentado elevado crescimento nos âmbitos da ciência e dos negócios. Buscar informações significativas nesse tipo de conjunto de dados considerando seu volume, demanda uma infraestrutura elaborada de computadores que trabalhem em paralelo, para viabilizar soluções em tempo eficiente. A elaboração e o aprimoramento de algoritmos que sejam adequados para processar e extrair informação útil dessa massa de dados, bem como a configuração do ambiente distribuído onde se dá a execução desses algoritmos, tornam seu escalonamento não trivial.

Em paralelo a essa demanda, há uma crescente disponibilidade de nuvens computacionais que oferecem variados **pacotes** de recursos computacionais para processar essas

tarefas. Nesse contexto, o custo para se implantar tal ambiente é vinculado diretamente à sua utilização (*pay-per-use*), o que dispensa grandes investimentos do usuário em infraestrutura [Armbrust et al. 2010]. Dessa forma, para o provedor de serviços, o modelo de computação em nuvem estabelecido pela oferta de recursos virtualizados reduz custos e provê flexibilidade e escalabilidade em sua administração e disponibilização de serviços.

O processamento de dados massivos era executado originalmente em ambientes não virtualizados. Entretanto, o advento da computação em nuvem tem levado os usuários a mover ambientes de processamento *big-data* para a nuvem [Assunção et al. 2015]. Atualmente há uma tendência de fazer tal processamento em máquinas virtuais (VMs)[1] [Hashem et al. 2015]. Ao desacoplar o processamento de uma infraestrutura física específica e contratar um certo volume de recursos (*cores*, memória, discos, etc.), o usuário pode organizar esses recursos de diferentes maneiras. Por exemplo, pode-se contratar um *cluster* com 4 VMs de 16 *cores* e 32 GB de RAM cada, ou 16 VMs de 4 *cores* e 8 GB de RAM, entre outras configurações. Escolher qual a configuração mais adequada para uma determinada aplicação se torna um problema complexo, ainda pouco estudado. Sendo assim, o uso de diferentes combinações de máquinas virtuais, com configurações e características diversas, consolidadas em um servidor na nuvem, tem impacto direto no desempenho das aplicações, na taxa de utilização dos recursos e no consumo dos *datacenters*.

O consumo de energia é um fator significativo, já que apresenta um custo relevante para o provedor da nuvem. De acordo com o relatório publicado por Shehabi e outros [Shehabi et al. 2016], no ano de 2014, 1,8% de toda energia consumida apenas nos Estados Unidos foi utilizada por *datacenters*. Espera-se ainda um aumento de 4% nessa demanda entre os anos de 2014 a 2020. Além disso, a matriz de produção energética se sustenta emitindo preocupantes quantidades de carbono na atmosfera, de acordo com o NRDC (Natural Resources Council) [Whitney and Delforge 2014].

Neste trabalho avaliamos o desempenho de diferentes combinações de máquinas virtuais em um ambiente de nuvem, com seu consumo de energia sistematicamente monitorado ao executar aplicações. Como carga para os testes, dois algoritmos de mineração de dados, Twidd e Eclat, em implementações paralelas, foram testados sobre uma base de dados contendo *posts* do Twitter. Pretendemos com esse estudo, demonstrar que há diferenças no desempenho e também no consumo das aplicações *big-data* para formas distintas de alocação de máquinas virtuais em um ambiente de nuvem, ainda que contando com o mesmo conjunto total de recursos físicos. Este trabalho se baseia em um estudo anterior [Volpini et al. 2018], porém nossa análise utilizou novos testes onde os algoritmos, bem como a maneira de alocar os recursos virtualizados, foram reparametrizados para melhor refletir o comportamento em nuvens públicas. Além disso, como forma de avaliar o impacto ao se utilizar um ambiente virtualizado, incluímos testes executados sem virtualização.

## 2. Trabalhos relacionados

Muito tem sido estudado sobre economia de energia em *datacenters*, incluindo o processamento de dados de grande volume. Já se demonstrou que é possível economizar

[1] Abreviatura de uso frequente referenciando o termo em inglês: Virtual Machines

energia ao processá-los [Leverich and Kozyrakis 2010], o que disparou interesse sobre o tema [Mashayekhy et al. 2014].

Há trabalhos que propõem maior eficiência energética por meio de arquitetura específicas para cargas *big-data* [Gu et al. 2011], ou soluções como o GreenNebula [Berral et al. 2014] que, por meio fontes renováveis, produz e armazena sua própria energia e gerencia máquinas virtuais para que migrem sempre para onde esteja disponível desse tipo energia. Outros estudos mostram como a escolha da topologia de redes [Abts et al. 2010] em um *datacenter*, a decisão do processador [Ramanathan 2006] e o escalonamento da tensão e frequência de operação em função da carga [Kim et al. 2009] são formas viáveis de redução no consumo de energia.

Abordagens mais recentes utilizam de GPUs (*Graphic Processing Units*) para processar grande volumes de dados [Ferro et al. 2017] e apresentam bons resultados com relativo baixo consumo. Alguns estudos apesar de não tratarem de processamento de grandes volumes de dados falam sobre o consumo de VMs [Kansal et al. 2010], inclusive na forma de alocá-las visando redução de consumo [Dupont et al. 2012] e ainda assim preservar boa relação com o desempenho [Ye et al. 2010].

Neste trabalho nos aproximamos de um cenário real de processamento de grandes volumes de dados em ambiente virtualizado de nuvem e avaliamos o impacto sobre o tempo de conclusão das tarefas e o consumo de energia nas diversas formas de escalonar recursos.

## 3. Metodologia de trabalho

Nosso trabalho observou o desempenho da aplicação, o ambiente de execução e o gasto energético de dois algoritmos de mineração de padrões frequentes realizados em diferentes configurações e composições de máquinas virtuais em um mesmo conjunto de servidores físicos. Com o objetivo de simular um ambiente de processamento de dados massivos em nuvem, escalonamos de variados modos a execução dos algoritmos por meio de aplicações distribuídas processando em uma mesma base. Utilizamos também um *cluster* de monitoração para coleta sistemática de métricas dos experimentos.

### 3.1. Aplicações

Twidd é uma aplicação que implementa o algoritmo FPGrowth sobre a abstração de RDD's (*Resilient Distributed Datasets*) para resolver o problema de mineração de padrões frequentes. A partir de uma série de transações, cada uma composta por um conjunto de itens e um limiar, encontram-se todos os subconjuntos de itens que ocorrem no mínimo a quantidade de vezes especificada por esse limiar. Usamos esta aplicação devido ao elevado custo computacional inerente ao processo de geração dos subconjuntos de itens para cada transação, o que consequentemente impacta no uso da CPU, alterando o uso de energia [Dias et al. 2016].

A segunda aplicação executa Eclat, um algoritmo que trabalha sobre uma representação vertical da base de dados. Ao invés de um conjunto de itens para cada transação, temos conjuntos de identificadores de transações para cada item. Eclat é iterativo pela sua característica hierárquica durante a mineração e irregular pela combinatória envolvida na distribuição dos *itemsets* frequentes [Dias et al. 2016]. Dessa forma, os pro-

cedimentos realizados a cada iteração são os mesmos, mas os dados de entrada/saída variam e consequentemente alteram seus padrões de comunicação.

A base de dados utilizada possui 8 GB de um conjunto de *posts* do Twitter, coletados usando a API da plataforma. Para o processamento paralelo das aplicações citadas foram definidas 128 partições sobre o conjunto de dados, para evitar subutilizar o processamento disponível e também a fragmentação de contêineres.

### 3.2. Ambiente de processamento

As aplicações citadas foram executadas na plataforma Apache Spark v1.5.2, utilizando o sistema de arquivos Hadoop/HDFS v2.6.0 sobre Yarn. Para provisionamento e orquestração do ambiente de máquinas virtuais utilizamos OpenStack na versão Pike em um servidor físico Intel Core-i7 2600 3,4 GHz com 16 GB de RAM. Como hospedeiros de máquinas virtuais foram utilizados 6 servidores Intel Xeon E5-2620v4 2,1 GHz com 32 GB de RAM, duas interfaces de rede Gigabit Ethernet e um disco SATA de 2 TB, contando com o *hypervisor* QEMU-KVM 2.10.1 sobre GNU/Linux Ubuntu 16.04, kernel 4.4.0.

### 3.3. Arquitetura de monitoração

Todos os dados de utilização de CPU, disco, ocupação de memória, carga da máquina, tráfego de rede e consumo de energia, além dos *logs* de execução do Spark, Yarn e do HDFS foram registrados em um *cluster* de monitoração *Seshat*, composto por 3 servidores físicos Intel Core-i7 2600 3,4 GHz com 16 GB de RAM. *Seshat* é uma arquitetura de monitoramento escalável para análise, notificação, coleta de métricas e *logs* em ambientes de nuvem [Conceição et al. 2018].

Para monitorar parâmetros elétricos como potência ativa por tomada, tensão, corrente e o consumo de cada servidor utilizamos a unidade de distribuição de energia PDU (*Power Distribution Unit*) Raritan PX-2 da série 5000, com exatidão de 1% (norma ISO/IEC 62053-21). Todos os servidores físicos executando máquinas virtuais foram conectados de forma independente às tomadas do PDU. Para esse estudo a potência ativa consumida pelos servidores e ativos de rede são a grandeza de maior relevância.

Nosso ambiente de monitoração *Seshat* é composto por quatro camadas: (i) coleta, (ii) transporte, (iii) armazenamento e (iv) visualização de dados, e tem seu comportamento resumido a seguir:

(i) **coleta**
- agentes de monitoramento são instalados nos servidores físicos, extraem métricas e as enviam para um serviço de filas;
- um *daemon* se conecta via SNMP ao PDU e recolhe medidas de energia para cada uma de suas tomadas de força e as envia para um serviço de filas;

(ii) **transporte**
- o servidor de monitoramento consome o serviço de filas e encaminha as métricas coletadas para o serviço de armazenamento;

(iii) **armazenamento**
- um banco de dados de séries temporais armazena as métricas coletadas dos servidores e do PDU;

(iv) **visualização de dados**

- *dashboards* customizados recuperam as métricas do armazenamento e as exibem em gráficos, tabelas e indicadores.

Além do ambiente de monitoração utilizamos o recurso de History Server do Spark para que após executar as aplicações fosse possível rastrear detalhes das tarefas, como tempo gasto por etapa, resultados intermediários de processamento, parâmetros de uso da JVM, entre outros. Com esse conjunto de informações armazenadas e disponíveis, foi possível determinar a duração da execução das tarefas associada a seu consumo de energia. Dessa forma, foi caracterizado o perfil de consumo de energia para cada execução dentro das variações de alocação de máquinas virtuais na nuvem.

### 3.4. Abordagem do problema

Ao executar aplicações no ambiente Spark há um grande conjunto de parâmetros que podem ser variados. Sendo uma aplicação distribuída, é possível definir configurações para o mestre (também chamado de *driver*), que é responsável por coordenar e gerenciar tarefas e execuções, para seus escravos (normalmente conhecidos como executores), que efetivamente executam processamento e escolher a quantidade de partições sobre o conjunto de dados. Ao mover esse ambiente para a nuvem, podemos também variar a forma como os recursos da nuvem são distribuídos em máquinas virtuais (por exemplo um *cluster* com 4 VMs de 16 *cores*, ou com 16 VMs de 4 *cores*).

Em nossos experimentos um primeiro conjunto de testes permitiu definir uma base de valores de configuração a fim de adequar o ambiente de processamento de dados massivos para as aplicações de mineração de padrões, executados sobre máquinas virtuais. Observamos durante o *setup* significativa melhora no desempenho geral ao disparar VMs sem ocupar a totalidade da memória RAM dos servidores físicos. Desse modo, parte da RAM dos servidores ficou disponível para o *hypervisor*. Partindo desse ponto, nossos testes avaliaram o impacto ao alterar a disponibilidade de *cores* de processamento para diferentes quantidades de executores em duas políticas de ocupação física da nuvem, além de avaliar sua execução diretamente nos servidores físicos.

Estruturamos nosso ambiente de execução Spark contando com o Yarn como escalonador de recursos para processar nossa base de dados de 8 GB distribuídas em um sistema de arquivos HDFS, com 3 réplicas sempre que possível[2]. Para garantir carga (*stress*) significativa sobre a nuvem definimos o suporte mínimo dos algoritmos em 0,004.

Avaliamos duas políticas de escalonamento ao ocupar 6 servidores físicos, variando gradativamente a quantidade de *cores* oferecidos às VMs em agregados de 16 *cores*. Com objetivo de avaliar o impacto da granularidade da virtualização os mesmos testes foram realizados em VMs de 4, 8 e finalmente 16 *cores*. As VMs de 4 *cores* contaram com 7 GB de RAM, as de 8 *cores* com 14 GB e as de 16 *cores* tinham 28 GB de RAM. Com esses valores, sempre que o servidor físico foi ocupado por completo em *cores*, ainda pôde contar com memória livre para uso do *hypervisor*. Dessa forma, executamos 5 séries de experimentos[3], mais a execução de testes sem virtualização, com ocupação total nos valores de 16, 32, 48, 64, 80 e 96 *cores*.

A primeira rodada de testes ocupando fisicamente os servidores em Round Robin, dispara VMs com 4 *cores*. Inicialmente temos 4 VMs em 4 servidores físicos diferentes,

[2]algumas configurações dos experimentos não cabem 3 réplicas

[3]Para VMs de 16 *cores* a ocupação da nuvem na forma Gulosa e Round Robin é idêntica

totalizando 16 *cores* alocados. Na sequência, mais 4 VMs são disparadas, ocupando os 2 servidores que estavam vazios com uma VM cada e mais 2 VMs são atribuídas respectivamente a 2 dos servidores[4], de modo que as tarefas passam a contar com 8 VMs de 4 *cores*, totalizando 32 *cores* alocados. A ocupação do ambiente de servidores segue até que se tenha todos os 6 servidores ocupados com 4 VMs cada, totalizando 96 *cores* alocados compondo uma série de testes. Esse procedimento se repetiu para VMs com 8 *cores* em acréscimos de 2 VMs para a composição de mais uma série e finalmente testes com VMs de 16 *cores* contam com acréscimos de uma VM por vez, uniformizando os pontos em cada série, já que contam sempre com múltiplos de 16 *cores*.

Para a primeira rodada com escalonamento Guloso para VMs com 4 *cores* temos 4 VMs ocupando completamente um único servidor, totalizando 16 *cores* alocados. Na segunda rodada, mais 4 VMs são disparadas, ocupando completamente um segundo servidor, e assim por diante até ocupação total dos 6 servidores contando com 96 *cores* em uso. O procedimento se repetiu para VMs de 8 *cores*, com acréscimos de 2 VMs por vez, o que acrescenta o mesmo **pacote** de recursos da rodada anterior. Para o caso Guloso de VMs de 16 *cores* a forma de ocupar o ambiente acaba sendo idêntica à que conta com VMs de 16 *cores* em Round Robin, uma vez que só será possível colocar uma VM por servidor.

Além das rodadas que ocupam a infraestrutura por VMs com 16 *cores* executamos as mesmas aplicações diretamente nos servidores físicos, ocupando um por rodada. Então nesses dois casos sempre foi necessário pelo menos 2 servidores, um mestre e um executor, mantendo assim uma ocupação mínima de 32 *cores*.

Durante os testes, as execuções realizadas que contaram com apenas 16 *cores* apresentaram tempo de conclusão das tarefas muito elevado, chegando a demorar até uma ordem de grandeza a mais que os resultados de menor tempo. Isso se deve à elevada restrição de recursos ofertados para as aplicações. Portanto, mantivemos os pontos de 16 *cores* em nossos testes em todas as séries, porém os descartamos das análises gráficas por dificultarem (comprimindo) a visualização dos demais resultados que pretendemos discutir. Os resultados detalhados dos testes estão apresentados na próxima seção.

## 4. Testes e discussão de resultados

Em nosso trabalho avaliamos principalmente dois parâmetros: **desempenho** das aplicações dado pelo tempo para concluir a tarefa em segundos e o **consumo de energia**, em watt-hora. Todos os valores apresentados nos experimentos contam com confiança de 95% sobre as médias. Os valores absolutos das confianças encontrados não foram superiores a 5,8% de suas respectivas médias, sendo majoritariamente inferiores a 2%. Para evitar excesso de informações nos gráficos, os intervalos de confiança dos experimentos foram suprimidos. Os gráficos, mantidos em mesma escala, foram desmembrados para facilitar avaliações, comparações e a compreensão dos resultados.

Ressaltamos que o consumo de energia possui forte relação com o tempo de execução da tarefa. Portanto, mesmo que executadas em um maior número de máquinas, físicas ou virtuais, há uma expectativa de que tarefas que contem com mais recursos sejam concluídas mais rapidamente e por isso apresentem a tendência de consumir menos.

[4]que podem inclusive ser os mesmos que não tinham VMs anteriormente

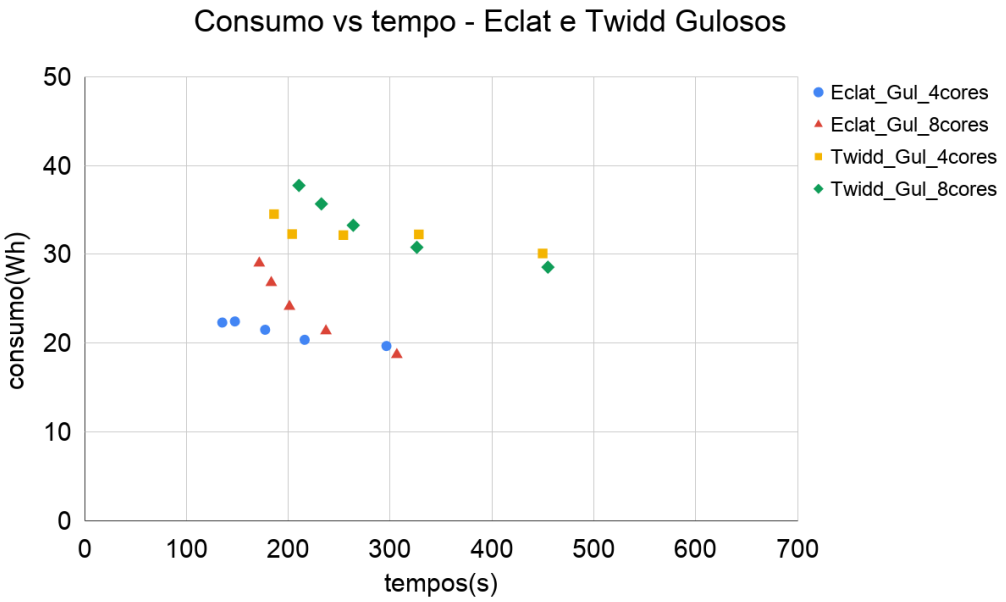



**Figura 1. Gráfico de dispersão do consumo *vs* tempo de conclusão da execução das aplicações em disponibilidades de 96, 80, 64, 48 e 32 *cores* na política de escalonamento Guloso**

Isso se deve ao fato de que o consumo da infraestrutura em repouso, ou seja, sem carga é significativo. Seguindo a metodologia apresentada na seção anterior, apresentamos os resultados como gráficos de dispersão do consumo *versus* o tempo de sua execução.

Os resultados obtidos para os algoritmos submetidos à política de ocupação Gulosa são mostrados na Figura 1. Já a Figura 2 mostra o comportamento das mesmas aplicações executadas em VMs distribuídas em Round Robin e finalmente a Figura 3 apresenta a ocupação da infraestrutura com VMs de 16 *cores*, que para fins de comparação está acompanhada pela execução dos algoritmos sem virtualização (*bare metal*).

Cada ponto pertencente a uma série apresentada, corresponde a uma determinada quantidade de *cores* ocupados. Tarefas com menor tempo correspondem a mais *cores* utilizados, não havendo exceção para nenhum dos casos. Assim, para cada série apresentada o ponto de cada série mais à esquerda no gráfico corresponde a 96 *cores* ocupados, seguido imediatamente à direita, na mesma série, pelo ponto de 80 *cores* e assim por diante de modo que mais a direita de cada série apresentada está o ponto correspondente a 32 *cores*.

Apesar de serem algoritmos diferentes, o resultado dos algoritmos Eclat e Twidd não se misturam na visualização gráfica, o que facilita nossa análise.

A Figura 1 apresenta os resultados obtidos distribuindo VMs de forma Gulosa, ao ocupar a nuvem. Observamos que para ambos os algoritmos, a medida que a oferta de recursos para a execução aumenta, sua conclusão responde de modo significativamente mais rápido. O impacto no consumo de energia apresenta um acréscimo sutil para VMs de 4 *cores* e uma elevação mais acentuada para VMs de 8 *cores*. Isso se deve ao fato de que a cada rodada um novo servidor é ocupado e passa a compor o consumo energético em paralelo com a redução do tempo de conclusão da execução, o que tem efeito antagônico.

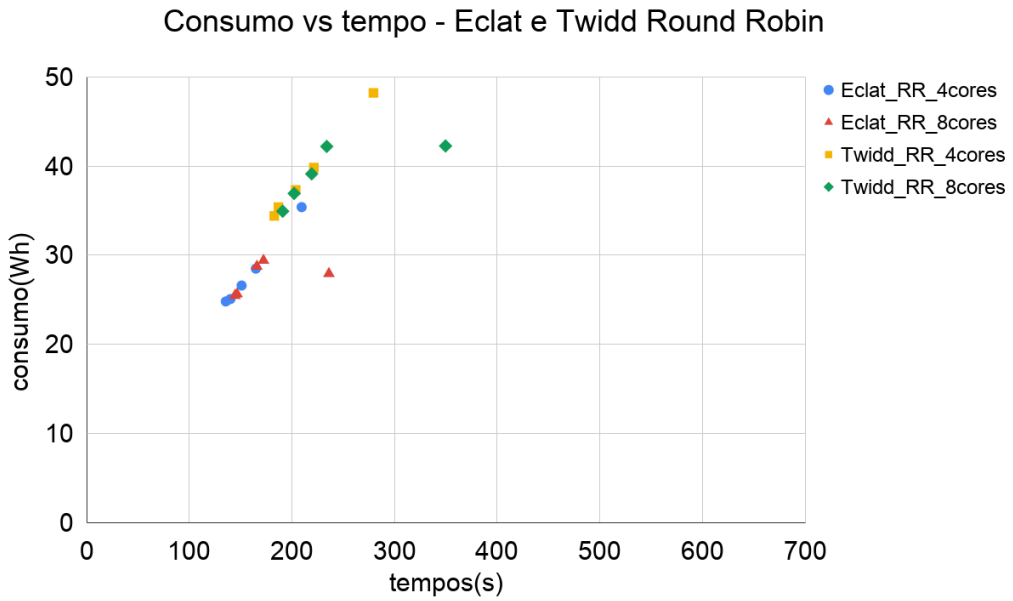



**Figura 2. Gráfico de dispersão do consumo *vs* tempo de conclusão da execução das aplicações em disponibilidades de 96, 80, 64, 48 e 32 *cores* na política de escalonamento Round Robin**

Portanto o resultado que predomina se sobressai no gráfico.

A perceptível inclinação na política Gulosa em VMs de 8 *cores* para ambos os algoritmos pode ser explicada: apesar do rigor pelo qual os recursos foram atribuídos a todas as configurações, as VMs de 4 *cores* contam com mais nós de processamento para executar a mesma tarefa. Como resultado disso há um acréscimo no grau de paralelismo da tarefa, além de melhor utilização dos recursos. Como exemplo, se uma VM de 8 *cores* entra em *garbage collection*, o que ocorre com frequência, mais *cores* ficam suspensos da tarefa, prejudicando sua conclusão de forma mais significativa que em VMs de 4 *cores*. Portanto, em termos gerais, contar com mais *cores* para resolver tarefas acelera sua solução mas também contribui significativamente no consumo de energia, além de ocupar mais recursos na nuvem. Não se pode perder de vista que quanto mais recurso da nuvem é alocado, inevitavelmente se eleva o custo.

Nossa experimentação mostrou que na ocupação Gulosa, VMs com 4 *cores* apresentam melhor desempenho relativo a um mesmo **pacote** de recursos, mas nem sempre o menor consumo de energia, ainda que em patamares bem próximos.

No gráfico da Figura 2 a ocupação da nuvem por VMs em Round Robin mostra que todas as configurações apresentam praticamente o mesmo comportamento. Se comparadas caso a caso com as execuções de mesmo recurso no escalonamento Guloso, Round Robin tipicamente se apresenta mais veloz porém com consumo de energia mais elevado. Em Round Robin mais servidores físicos são demandados em processamento o que acaba elevando o *clock* de suas CPUs. Como consequência há um aumento significativo no consumo. É interessante observar que não há diferenças muito expressivas nessa política entre usar VMs de 4 ou de 8 *cores*. Exceções a esse comportamento são os pontos que se destacam mais à direita, correspondendo à utilização de VMs com 8 *cores*, que contam com 32 *cores* distribuídos em um mestre e três escravos, do qual somente 24 *cores*

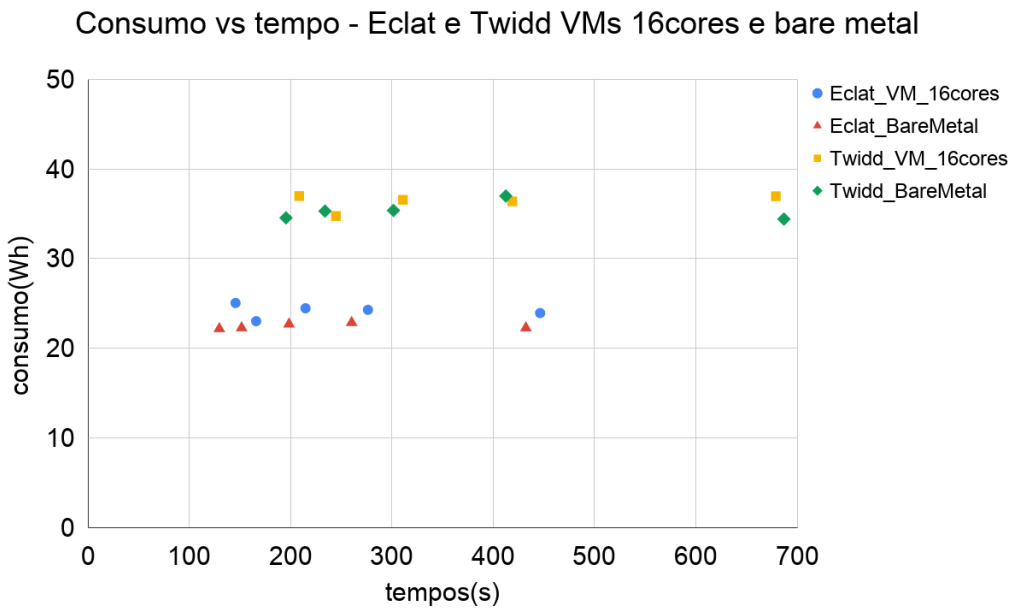



**Figura 3. Gráfico de dispersão do consumo *vs* tempo de conclusão da execução das aplicações em disponibilidades de 96, 80, 64, 48 e 32 *cores*, sobre VMs de 16 *cores* e diretamente no servidor físico (*bare metal*)**

são utilizados para execução das aplicações, já que o mestre atribui as tarefas mas não as processa efetivamente. Isso não acontece para as VMs com 4 *cores* na série de 32 *cores*, em que há um melhor aproveitamento do grau de paralelismo, consumindo mais, porém encerrando a tarefa mais depressa que com o mesmo conjunto de recursos ocupados para 8 *cores* (mas não tão depressa quanto o necessário para reduzir o consumo).

Por isso que ao disponibilizarmos mais recursos, o efeito do mestre, que tipicamente subutiliza os recursos de sua VM, se torna menos significativo e faz com que o problema se comporte de forma mais "linear", sendo cada vez mais veloz e com consumo proporcionalmente menor.

Na Figura 3 apresentamos a execução de ambos algoritmos para VMs com 16 *cores* e a execução não virtualizada das aplicações. Destacamos os pontos mais à direita no gráfico, correspondentes à ocupação de 32 *cores*, que indicam a mesma subutilização apontada nas análises anteriores: apenas um mestre e somente um executor processando a aplicação.

Não se observam grandes variações de consumo para um dado algoritmo na ocupação por essas VMs ou executado em *bare metal*. Apesar da distribuição de recursos nesse caso para Round Robin ser igual à Gulosa em nossa nuvem, predomina o comportamento guloso uma vez que se nota significativa semelhança de comportamento com as séries apresentadas no gráfico da Figura 1.

É interessante observar o impacto da virtualização, visto que na maioria dos casos há um leve atraso e um pequeno acréscimo no consumo para as execuções virtualizadas se comparadas par-a-par com sua respectiva execução *bare metal*. A exceção desse comportamento, pela sutil diferença de consumo em favor da virtualização, ocorre apenas nas execuções do Twidd para VMs com 16 *cores* nos pontos de 80 e 48 *cores*.

Novamente aqui é possível perceber que em todas as séries há melhora nos resultados ao se ofertar mais cores, mas com ganhos decrescentes de desempenho à medida que nos aproximamos da ocupação total da infraestrutura, o que eleva os custos de execução.

Ao compararmos os gráficos das Figuras 1, 2 e 3, destacamos que o melhor desempenho virtualizado para aplicação Eclat é obtido com 96 *cores* em VMs de 4 *cores*, onde a execução se completa em 135 segundos, valor que se aproxima da execução em *bare metal* que foi de 129 segundos. O pior desempenho para o Eclat virtualizado foi obtido em 32 *cores* para VMs com 16 *cores*, alcançando 446 segundos. Já o melhor consumo foi obtido com escalonamento Guloso ocupando 32 *cores* em VMs de 8 *cores*, onde apenas 18Wh foram necessários para concluir a tarefa. O pior consumo foi para o escalonamento em Round Robin ocupando 32 *cores* em VMs de 4 *cores* chegando a 35Wh. Isso se explica pelo fato dos servidores físicos consumirem energia mas não disponibilizarem recursos suficientes para concluir rapidamente a tarefa.

Para o Twidd o melhor desempenho virtualizado é obtido com 96 *cores* em VMs de 4 cores, concluindo a execução em 183 segundos, superando inclusive o *bare metal* que é de 196 segundos. Já os piores tempos ficam para 32 *cores* oferecidos em VMs de 16 *cores* levando 680 segundos, e ainda assim superando a execução em *bare metal* que leva 7 segundos a mais. Do ponto de vista do consumo, 32 *cores* distribuídos em VMs de 8 *cores* pelo escalonamento Guloso conseguem concluir a tarefa consumindo 29Wh, enquanto os mesmos 32 *cores* ocupados em Round Robin chegam a consumir 48Wh.

Ao avaliar todas as curvas percebemos que trabalhar com poucos recursos (*cores*) faz com que o desempenho das tarefas seja significativamente fraco. Configurações com poucos *cores* e consequentemente menos memória, geram os piores tempos, pois é mais custoso em termos de espaço conter o problema todo em memória. À medida que aumentamos os recursos, os resultados melhoram significativamente. Para as aplicações testadas se o objetivo é economizar energia, parece mais adequado uma abordagem Gulosa com VMs de 4 *cores* para ocupar a nuvem. Em contraste, a ocupação sob a política de Round Robin também em VMs de 4 *cores*, parece mais interessante do ponto de vista de desempenho, desde que o custo com energia não seja relevante.

## 5. Conclusão e Trabalhos Futuros

Visando avaliar o desempenho sem perder de vista o consumo de energia ao processar dados massivos em ambientes virtualizados, estabelecemos uma estrutura de nuvem monitorada para verificar o impacto do escalonamento de recursos em função da quantidade, do tamanho das VMs e políticas de ocupação física da estrutura. Nessa nuvem foram realizados testes com dois algoritmos paralelos de execução distribuída que nos permitiram perceber como a política de ocupação de recursos pode influenciar resultados de tempo de conclusão de tarefas e consumo de energia.

Em nossos testes observamos que acrescentar recursos para processar dados massivos em ambientes virtualizados nos permite acelerar significativamente a conclusão das tarefas. Pelo que inferimos, a política de ocupação Gulosa do ambiente favorece a economia de energia, enquanto a política de Round Robin se mostrou mais interessante em termos de desempenho, principalmente quando há limitações de recursos disponíveis.

Verificamos ainda que o consumo de energia para uma mesma tarefa pode apresentar consideráveis variações, chegando a dobrar de valor, conforme lhe sejam ofertados

recursos.

Para trabalhos futuros, pretendemos estender os testes para outros algoritmos PDM com características e exigências diferentes dos que utilizamos. Dessa forma, será possível avaliar o impacto de diversas cargas *big-data* que sejam executadas virtualizadas em ambiente de nuvem. Consideramos avaliar outros virtualizadores para os mesmos testes e apurar se nossos resultados permanecem válidos. Além disso, pretendemos analisar a influência dos outros componentes, como interfaces de rede e discos, e suas diferentes configurações de forma para verificar se há ganhos no desempenho, consumo ou mesmo ambos.

## Agradecimentos

Este trabalho foi parcialmente financiado por Fapemig, CAPES, CNPq, e pelos projetos MCT/CNPq-InWeb (573871/2008-6), FAPEMIG-PRONEX-MASWeb (APQ-01400-14), H2020-EUBR-2015 EUBra-BIGSEA e H2020-EUBR-2017 Atmosphere.

## Referências

Abts, D., Marty, M. R., Wells, P. M., Klausler, P., and Liu, H. (2010). Energy proportional datacenter networks. *SIGARCH Comput. Archit. News*, 38(3):338–347.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2010). A view of cloud computing. *Commun. ACM*, 53(4):50–58.

Assunção, M. D., Calheiros, R. N., Bianchi, S., Netto, M. A., and Buyya, R. (2015). Big data computing and clouds: Trends and future directions. *Journal of Parallel and Distributed Computing*, 79-80:3 – 15. Special Issue on Scalable Systems for Big Data Management and Analytics.

Berral, J. L., Goiri, Í., Nguyen, T. D., Gavalda, R., Torres, J., and Bianchini, R. (2014). Building green cloud services at low cost. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 449–460. IEEE.

Conceição, V. S., Volpini, N. D. O., and Guedes, D. (2018). *Seshat*: uma arquitetura de monitoração escalável para ambientes em nuvem. In *Anais do XVII Workshop em Desempenho de Sistemas Computacionais e de Comunicaç ao, Natal-RN. Sociedade Brasileira de Computaçao (SBC)*.

Dias, V., Meira, W., and Guedes, D. (2016). Dynamic reconfiguration of data parallel programs. In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 190–197.

Dupont, C., Giuliani, G., Hermenier, F., Schulze, T., and Somov, A. (2012). An energy aware framework for virtual machine placement in cloud federated data centres. In *Future Energy Systems: Where Energy, Computing and Communication Meet (e-Energy), 2012 Third International Conference on*, pages 1–10. IEEE.

Ferro, M., Yokoyama, A., Klôh, V., Silva, G., Gandra, R., Bragança, R., Bulcao, A., Schulze, B., and SA-PETROBRAS, P. B. (2017). Analysis of gpu power consumption using internal sensors. In *Anais do XVI Workshop em Desempenho de Sistemas Computacionais e de Comunicaç ao, Sao Paulo-SP. Sociedade Brasileira de Computaçao (SBC)*.

Gu, X., Hou, R., Zhang, K., Zhang, L., and Wang, W. (2011). Application-driven energy-efficient architecture explorations for big data. In *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, pages 34–40. ACM.

Hashem, I. A. T., Yaqoob, I., Anuar, N. B., Mokhtar, S., Gani, A., and Khan, S. U. (2015). The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, 47:98–115.

Kansal, A., Zhao, F., Liu, J., Kothari, N., and Bhattacharya, A. A. (2010). Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 39–50. ACM.

Kim, K. H., Beloglazov, A., and Buyya, R. (2009). Power-aware provisioning of cloud resources for real-time services. In *Proceedings of the 7th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '09, pages 1:1–1:6, New York, NY, USA. ACM.

Leverich, J. and Kozyrakis, C. (2010). On the energy (in) efficiency of hadoop clusters. *ACM SIGOPS Operating Systems Review*, 44(1):61–65.

Mashayekhy, L., Nejad, M. M., Grosu, D., Lu, D., and Shi, W. (2014). Energy-aware scheduling of mapreduce jobs. In *Big Data (BigData Congress), 2014 IEEE International Congress on*, pages 32–39. IEEE.

Ramanathan, R. M. (2006). Intel® multi-core processors: Making the move to quad-core and beyond. http://www.pogolinux.com/learn/files/quad-core-06.pdf. Acesso em: 07 de set. 2018, 17:20.

Shehabi, A., Smith, S. J., Sartor, D. A., Brown, R. E., Herrlin, M., Koomey, J. G., Masanet, E. R., Horner, N., Azevedo, I. L., and Lintner, W. (2016). United states data center energy usage report.

Volpini, N. D. O., Conceição, V. S., Pontes, R. L., and Guedes, D. (2018). Uma análise do consumo de energia de ambientes de processamento de dados massivos em nuvem. In *Anais do XVII Workshop em Desempenho de Sistemas Computacionais e de Comunicaç ao, Natal-RN. Sociedade Brasileira de Computaçao (SBC)*.

Whitney, J. and Delforge, P. (2014). Scaling up energy efficiency across the data center industry: Evaluating key drivers and barriers. https://www.nrdc.org/energy/files/data-center-efficiency-assessment-IP.pdf. Acessado em maio de 2015.

Ye, K., Huang, D., Jiang, X., Chen, H., and Wu, S. (2010). Virtual machine based energy-efficient data center architecture for cloud computing: a performance perspective. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 171–178. IEEE Computer Society.

# SMCis: um sistema para o monitoramento de aplicações científicas em ambientes HPC

**Gabrieli Silva, Vinícius Klôh, André Yokoyama, Mariza Ferro, Bruno Schulze**

[1]Laboratório Nacional de Computação Científica – LNCC
Av. Getúlio Vargas, 333 – 25651-075 – Quitandinha, Petrópolis – RJ –Brasil

{gabrieli,viniciusk,andremy,mariza,schulze}@lncc.br

***Abstract.*** *The performance evaluation in HPC, understanding the computational requirements of scientific applications, its relation with power consumption is a fundamental task to overcome the current barriers and to achieve the computational exascale. However, this imposes some challenging tasks, such as to monitor a wide range of parameters in heterogeneous environments, to enable fine grained profiling and power consumed across different components, to be language independent and to avoid code instrumentation. Considering these challenges, this work proposes the SMCis, an application monitoring system developed with the goal of to collect all these aspects in an effective and accurate way, as well as to correlate these data graphically, with the environment of analysis and visualization.*

***Resumo.*** *Monitorar aplicações, ambientes, degradação de desempenho e consumo energético são tarefas indispensáveis em HPC para alcançar a geração de computadores exaescala. Porém, esta não é uma tarefa trivial, pois é necessário coletar um conjunto de parâmetros distintos, que podem mudar a cada avaliação, em ambientes heterogêneos, com precisão, tanto no sistema como por componentes, ser independente da linguagem e evitar a instrumentação do código. Além disso, não é comum encontrar uma única ferramenta capaz de capturar todos esses aspectos e relacioná-los para análise, o que torna o processo de experimentação altamente custoso. Esses desafios motivaram o desenvolvimento de um sistema de monitoramento, com o qual é possível coletar, com alta acurácia e baixa sobrecarga de recursos, todos esses aspectos, além de relacioná-los graficamente através de um ambiente de análise e visualização.*

## 1. Introdução

O uso da Computação de Alto Desempenho (HPC) tem sido fundamental para permitir a descoberta de novos conhecimentos e por isso o seu emprego tem se tornado crucial para as pesquisas científicas em muitos domínios de investigação. Com o poder computacional oferecido pela atual geração de supercomputadores petaflópicos já é possível realizar simulações complexas, com realismo e precisão nunca alcançadas. Porém, apesar desses avanços, existe um número crescente de problemas para os quais as simulações ainda não são precisas o suficiente ou mesmo inviáveis de serem realizadas. Para diversas áreas de aplicação, tais como a indústria de energia, ciência dos materiais, clima e medicina de precisão, muitos desafios científicos ainda são complexos demais para os recursos computacionais disponíveis e poderão se beneficiar do crescimento da computação de alto desempenho, como o esperado para a futura geração de supercomputadores exaescala.

Muitos trabalhos, tais como [Simon 2013], [Bergman et al. 2008], [Alvin et al. 2010], [Rajovic et al. 2013] e [Messina 2017] consideram que uma das maiores barreiras para viabilizar a exaescala de processamento é o consumo energético dos ambientes HPC. Se atualmente os gastos com energia elétrica dos supercomputadores petaflópicos chegam a valores elevados, alcançar a nova geração dependerá de muito esforço para aumentar consideravelmente a capacidade de processamento e ao mesmo tempo reduzir o consumo energético.

Para alcançar essa nova geração, monitorar aplicações, ambientes, degradação de desempenho e consumo energético são tarefas indispensáveis. É preciso aprofundar o conhecimento sobre os fatores que limitam o desempenho das aplicações e interferem no consumo de energia, e mapeá-los para as arquiteturas que representam o atual estado da arte no processamento de alto desempenho.

O paradigma mais utilizado para se determinar o desempenho de hardware é por meio de programas de avaliação de desempenho, os *benchmarks*. No entanto, com o seu uso normalmente são obtidos picos teóricos de desempenho, uma medida pouco informativa quando é necessário determinar o real desempenho do hardware frente aos reais requisitos das aplicações. Os *benchmarks*, na maioria das vezes, não representam o modelo da aplicação científica, tanto em termos de requisitos computacionais que caracterizam sua aplicação, bem como na realidade da linguagem de programação ou tamanho de problemas que são executados no cenário real da pesquisa. Assim, a análise de desempenho que seja realmente informativa sobre as aplicações reais, já é em si outro desafio.

Pensando neste tipo de avaliação, foi desenvolvida uma metodologia sistemática, voltada para os requisitos das aplicações científicas [Ferro 2015]. A metodologia busca identificar os objetivos da avaliação e o conjunto de parâmetros mais relevantes a serem avaliados de acordo com o problema. As aplicações são categorizadas em termos dos seus requisitos computacionais com base em uma classificação denominada Motifs [Patterson 2013], usada como referencial para o estudo das aplicações científicas. As avaliações são feitas com base nos requisitos das aplicações reais e como estes se relacionam com outros aspectos que influenciam no seu desempenho [Licht 2014] [de Oliveira 2016] [Ferro et al. 2016] e no consumo de energia [Ferro et al. 2017], auxiliando na descoberta dos aspectos relevantes para se alcançar a melhoria geral do desempenho e do consumo energético. Entretanto, monitorar e avaliar um conjunto de parâmetros, os quais podem mudar a cada avaliação e que podem envolver aspectos da aplicação, desempenho, escalabilidade e consumo de energia, e ainda como estes se relacionam, não é uma tarefa trivial.

Assim, outro desafio é definir uma maneira precisa de coletar todos esses parâmetros e relacioná-los para uma análise eficaz. Foram investigados um conjunto de ferramentas e técnicas de monitoramento, descritas brevemente na Seção 3. No entanto, cada uma delas coleta um conjunto específico de parâmetros, os quais nem sempre atendem às necessidades da avaliação, o que tornava o processo de experimentação altamente custoso e demorado, pois era necessário combinar um grupo de ferramentas. Esses desafios motivaram o desenvolvimento de uma ferramenta de monitoramento que permitisse esse tipo de avaliação, com alta acurácia e com baixa sobrecarga de recursos. Portanto, neste trabalho é proposto o SMCis, um sistema de monitoramento de aplicações com o qual é possível coletar todos esses aspectos de maneira eficiente e relacioná-los grafica-

mente com um ambiente desenvolvido para análise e visualização de dados experimentais.

Este trabalho está organizado da seguinte forma: Na Seção 2 é apresentado uma breve discussão sobre análise de desempenho e energia. Na Seção 3 são apresentadas algumas ferramentas de monitoramento de desempenho e consumo de energia. Na Seção 4 é apresentado o SMCis, um sistema desenvolvido para monitorar desempenho e energia enquanto uma aplicação é executada. Finalmente, na Seção 5 é apresentada a conclusão.

## 2. Análise de desempenho e energia

Diversas iniciativas colaborativas entre governo, academia e indústria apontam os principais desafios e possíveis estratégias para se alcançar a exaescala de processamento em HPC [Kogge et al. 2008], [Ashby et al. 2010], [Alvin et al. 2010], [Rajovic et al. 2013], [Messina 2017], [HPC 2017]. Entre eles a preparação das próprias simulações e algoritmos para que de fato possam alcançar os níveis de processamento que serão oferecidos por essas futuras arquiteturas e a eficiência energética. A próxima geração de supercomputadores precisará ser desenvolvida usando abordagens onde os requisitos do problema científico orientem a arquitetura do computador e o projeto do software do sistema. Além disso, esses requisitos do problema científico deverão orientar a orquestração de diferentes técnicas e mecanismos de economia de energia, com a finalidade de melhorar o balanceamento entre economia de energia e desempenho das aplicações. Para isso, monitorar aplicações, ambientes, degradação de desempenho e consumo energético são tarefas indispensáveis. É preciso aprofundar o conhecimento sobre os fatores que limitam o desempenho das aplicações e interferem no consumo de energia, e mapeá-los para as arquiteturas que representam o atual estado da arte no processamento de alto desempenho.

Avaliar todos esses aspectos, impõe o desafio de definir uma maneira precisa e eficaz de coletar e relacionar todos esses parâmetros. Na metodologia utilizada neste trabalho os parâmetros a serem avaliados podem mudar a cada avaliação, mas com o foco nas aplicações, seus requisitos computacionais e como estes se relacionam com diferentes arquiteturas, tamanhos de problema e consumo de energia. Após ampla pesquisa não foi possível encontrar uma ferramenta única capaz de coletar todos os aspectos relacionados ao desempenho e eficiência energética das aplicações e do sistema. Além disso, a maioria das ferramentas são voltadas para o monitoramento do sistema como um todo ou para a criação do perfil das aplicações. Essas ferramentas avaliam o impacto que a aplicação está causando no ambiente, mas não fornecem a porcentagem do consumo de recursos para toda a execução da aplicação. Além disso, ferramentas focadas em otimização, apresentam análises de como cada função do código está consumindo recursos, mas não sobre o consumo de energia. Assim, muitas vezes era necessário combinar diversas ferramentas e ainda assim havia a dificuldade de obter uma análise sobre como a execução da aplicação consumia recursos por componente e potência.

Além disso, para alcançar os objetivos propostos, é necessário que a metodologia de testes obtenha medições precisas e refinadas. Uma granularidade fina nas medições é necessária para permitir a identificação de características da aplicação e como diferentes recursos estão sendo utilizados, permitindo identificar aspectos para redução de potência ou otimização de desempenho [Zomaya and Lee 2012]. É importante que a ferramenta utilizada seja independente da linguagem de implementação e que evite a instrumentação do código da aplicação. Normalmente aplicações científicas possuem um núcleo numérico escrito em uma linguagem e *frameworks*, e bibliotecas de comunicação

em outra. Sendo assim, a capacidade de ser independente da linguagem de implementação é essencial [Adhianto et al. 2010]. Já a instrumentação pode modificar o comportamento da aplicação, geralmente devido a sobrecarga e, para algumas aplicações, o tempo necessário para preparar os experimentos será longo.

As ferramentas de monitoramento de potência e energia também devem permitir o perfil de consumo com granularidade fina. Uma medição com taxa de amostragem de 1 ou 2 Hz (amostras por segundo) pode não ser rápida o suficiente para capturar pequenas mudanças no consumo de energia, as quais são importantes para entender o comportamento do sistema e da aplicação. Além do monitoramento, outra característica desejável é permitir controlar o uso da energia enquanto a aplicação é executada. Alguns pesquisadores apontam que para alcançar a exaescala de processamento, onde o consumo de potência será um recurso altamente restrito, serão necessárias técnicas baseadas em auto ajuste de energia. Mas para isso, além de medir o consumo de energia com granularidade fina, é necessário medir a energia consumida por diferentes componentes, como o processador e a memória, e poder controlá-la.

Na busca por essas características para monitorar o desempenho e o consumo de energia em ambientes distribuídos heterogêneos, como HPC, é que foram investigadas várias abordagens e ferramentas de monitoramento, as quais são apresentadas na Seção 3. Para cada uma delas são apresentadas as vantagens e as limitações para a computação científica no cenário atual e futuro.

## 3. Abordagens e Ferramentas de monitoramento de desempenho e energia

Existem muitas ferramentas para o monitoramento de desempenho e energia, sendo que muitas delas foram projetadas para atender objetivos específicos de grupos de pesquisa e por isso monitoram diferentes grupos de parâmetros. Nesta seção são apresentadas algumas das ferramentas analisadas, apontando as vantagens e limitações observadas. Dada a grande abrangência da área e os diversos trabalhos que propõem suas próprias ferramentas, os trabalhos relacionados se restringem a essas ferramentas.

O Nagios [Guthrie 2013] é uma ferramenta de código aberto que usa uma abordagem *online* para monitorar servidores, serviços, aplicações e *hosts* locais ou remotos. Além disso, fornece vários recursos para garantir que os sistemas, aplicações e serviços estejam funcionando adequadamente. Seu foco principal não está em analisar o desempenho de um processo específico (aplicação), mas no impacto que a aplicação está causando no ambiente, monitorando o desempenho do sistema. De acordo com [Nagios Team 2016] existe um plugin nativo (*check_proc*) para monitorar um processo específico no Linux. No entanto, após inúmeras tentativas de uso sem sucesso, desenvolvemos um plugin para este fim, com o qual é possível obter a porcentagem que o processo está utilizando da CPU, Memória e E/S, enquanto a aplicação está sendo executada [Klôh et al. 2016]. O Intel VTune [Reinders 2005], que também utiliza a abordagem *online*, fornece uma análise de diferentes parâmetros sobre a execução da aplicação. No entanto, muitos desses parâmetros possuem definições próprias que a Intel nomeou por si só, o que dificulta a interpretação dos resultados.

Nagios e Intel VTune são ferramentas que usam a técnica de amostragem para a coleta de dados. Com o Nagios é difícil obter um equilíbrio entre a frequência da amostra e a precisão dos dados, pois o intervalo mínimo de amostragem oferecido pela

ferramenta é de 10 amostras por segundo, o que oferece baixa precisão. O VTune oferece um conjunto de módulos para diferentes tipos de análises, tais como *Basic Hotspots* (que identifica o tempo passado um uma região do código) e *Memory Access* (que obtém informações de como a aplicação consome recursos relacionados à memória). No entanto, para alguns módulos, a ferramenta utiliza a coleta por amostragem e rastreamento de eventos simultaneamente. Porém, o rastreamento fornece uma alta sobrecarga para a análise. Logo, utilizar as duas técnicas simultaneamente pode não ser a melhor alternativa. Com o VTune, o intervalo de amostragem é muito pequeno (no mínimo 1 amostra/milissegundo), o que ocasiona uma alta sobrecarga.

TAU [Shende and Malony 2006] e Pablo [Reed et al. 1992] também são ferramentas que fazem uso das abordagens de amostragem e rastreamento simultaneamente. No entanto, para grandes aplicações paralelas, a geração de dados históricos sobre a execução da aplicação por meio do rastreamento será impraticável. Quanto maior o número de processadores, maior a geração de dados e, consequentemente, maior a complexidade dos dados gerados e sua apresentação nas ferramentas de visualização.

A medida que potência e energia se tornaram restrições dominantes, os fornecedores desenvolveram uma variedade de pacotes para medir e controlar o consumo de potência e energia. Estes são focados em métodos diretos para acessar os sensores internos e obter o consumo. Exemplos são as ferramentas VTune e NVSMI [NVIDIA 2017]. Porém, para monitorar energia com o VTune, é necessário um *driver* específico (*powerdk*), o qual requer muitas dependências e configurações e após inúmeras tentativas de instalação e uso não foi obtido sucesso [Silva et al. 2017].

O NVML [NVIDIA 2012] é uma API desenvolvida para monitorar e gerenciar estados de dispositivos GPU NVIDIA, como taxa de utilização de GPU, execução de processo, estado e desempenho de clock, temperatura e velocidade, consumo e gerenciamento de energia. O NVSMI permite aos administradores do sistema consultar, com os privilégios apropriados, os estados dos dispositivos GPU. Suas funções são fornecidas pela biblioteca NVML. A taxa de amostragem do NVSMI sobre a coleta de energia é um pouco baixa, em torno de 1000 milissegundos (1Hz), a qual pode não ser suficiente para perceber mudanças na potência consumida por aplicações com pequeno tempo de execução [Silva et al. 2017].

Existem soluções que integram software e hardware na coleta do consumo energético por componentes [Labasan 2016], como PowerPack [Ge et al. 2009] e PowerMon [Bedard et al. 2009]. Os componentes de hardware incluem, por exemplo, sensores, medidores e circuitos e os componentes de software incluem drivers para vários medidores e sensores. Apesar dessas ferramentas terem medições refinadas, elas utilizam hardware externo, que em ambientes HPC é uma tarefa custosa devido a grande quantidade de nós computacionais, além do custo de aquisição desses dispositivos de medição.

Medidores externos, como Watt's Up Pro [Ext 2017] e Cray XC30 [Labasan 2016], apesar de fornecerem medições de potência de forma direta, apresentam algumas limitações. Possuem baixa granularidade e coletam o consumo de energia de todo o sistema (ou do nó), o que não revela como a energia está sendo consumida por diferentes componentes, como o processador e a memória.

A medição direta por meio de sensores internos ou externos é consi-

derada a técnica mais adequada, pois os dados são coletados com mais precisão [Bridges et al. 2016]. Com os sensores internos não é necessária a aquisição de qualquer hardware externo para realizar a coleta e é possível ainda obter o consumo por componente. No entanto, nem todo hardware possui sensores internos, e quando disponíveis, as formas de acessá-los e a precisão de coleta podem mudar a cada arquitetura.

Com o objetivo de monitorar a execução de uma aplicação, analisando como seu desempenho e consumo de energia são afetados quando executada em diferentes arquiteturas, modelos de implementação e tamanhos de problema, é que as ferramentas e abordagens mencionadas foram analisadas. As vantagens e limitações apresentadas motivaram o desenvolvimento do Sistema de Monitoramento do grupo ComCiDis[1] (SMCis). Este sistema é uma das principais contribuições deste trabalho e é apresentado a seguir.

## 4. A Ferramenta SMCis

Como mencionado, para contornar as limitações encontradas nas ferramentas de monitoramento é que foi desenvolvido o SMCis. Esse sistema possibilita a coleta dos parâmetros de desempenho e energia sobre a execução da aplicação em diferentes arquiteturas, com altas taxas de amostragem e baixo impacto no consumo de recursos computacionais. Além disso, possui fácil utilização e provê a visualização gráfica dos diferentes parâmetros simultaneamente. O SMCis está disponível em [ComCiDis 2018].

### 4.1. Metodologia do sistema

Utilizando a metodologia para avaliação de desempenho apresentada na Figura 1 são definidos os objetivos da avaliação, os parâmetros a serem monitorados, para quais aplicações e em que arquiteturas. Para esse monitoramento, que era bastante custoso combinando diferentes ferramentas, foi desenvolvido o SMCis.

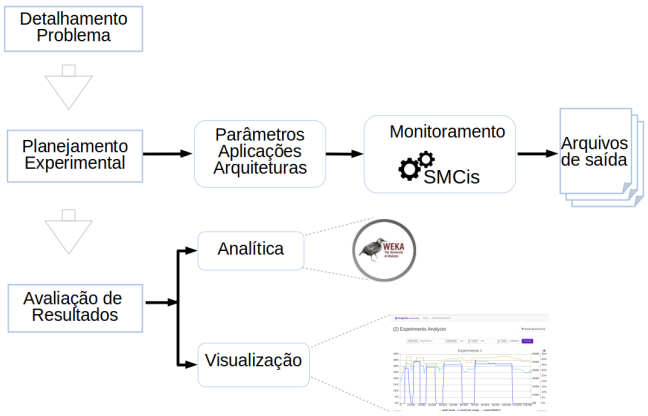



**Figura 1. Metodologia para avaliação de desempenho e energia onde a ferramenta SMCis se insere.**

Após o monitoramento, a ferramenta apresenta os resultados em um arquivo de saída que é utilizado na fase de avaliação dos resultados. Os resultados podem ser explorados tanto para uma análise analítica, onde esse arquivo é convertido para o formato .arff e utilizado na ferramenta de aprendizado de máquina Weka, ou diretamente no módulo de visualização do SMCis.

[1]ComCiDis - Computação Científica Distribuída

O SMCis utiliza a técnica de coleta por amostragem e a abordagem *offline*, pois foram consideradas as mais apropriadas ao tipo de análise realizada. Com a amostragem não há necessidade de inserir diretivas de instrumentação no código. Com a análise *offline* dos dados, a concorrência por recursos do sistema diminui, pois os dados resultantes do monitoramento são analisados em uma etapa posterior à execução da aplicação.

Levando em consideração que em ambientes HPC, medidores de energia externos não oferecem a precisão desejada, além de serem difíceis de ser utilizados, seja pela necessidade de aquisição dos aparelhos de medição ou pela necessidade de acesso físico, o SMCis foi desenvolvido para monitorar energia através dos sensores internos, presentes na maioria das arquiteturas dos ambientes de HPC. Porém, para coletar as informações desses sensores muitas vezes é necessário privilégio de super usuário, sendo esta a única limitação para coletar o consumo energético com o SMCis. Como os sensores internos fornecem informações referentes ao sistema, na metodologia experimental (Figura 2) um lançador de aplicações é utilizado para permitir o monitoramento do estado dos sensores antes e depois da execução da aplicação. Dessa forma, é possível calcular os consumos estático e dinâmico de potência e temperatura, onde o primeiro refere-se ao consumo do sistema e o segundo a execução da aplicação. Na Figura 2, é apresentado o fluxograma dos módulos de monitoramento que compõe o SMCis (detalhados na Seção 4.2).

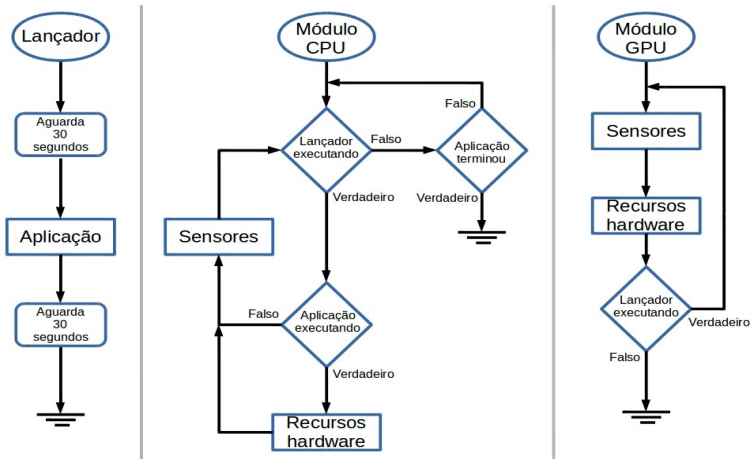



**Figura 2. Fluxograma dos módulos de monitoramento da ferramenta SMCis.**

## 4.2. Arquitetura do sistema

O SMCis é composto por dois módulos de monitoramento, um para CPU e outro para GPU, e um módulo de visualização e análise de dados experimentais, chamado *GraphCis*.

**Módulo de monitoramento em CPU**: este módulo foi desenvolvido em *Python* e utiliza a biblioteca psutil [Rodola 2018] para a coleta dos parâmetros de hardware (taxa de utilização de CPU, memória alocada e taxa de leitura e escrita para disco e rede). O psutil permite coletar informações da utilização de recursos do sistema referentes à aplicação em execução. O módulo pode ser configurado com taxas de amostragem ajustáveis de acordo com a necessidade do experimento, sendo a maior precisão na coleta a taxa de 10 Hz. Essa precisão dá-se pelo fato de o módulo calcular as taxas de leitura e escrita como sendo a diferença entre duas amostras consecutivas do monitoramento. Assim, o intervalo entre as amostras deve ser grande o suficiente para detectar o fluxo de leitura e escrita e

pequeno o suficiente para se obter acurácia nos dados coletados. Quando o método de coleta de energia é ativado, utiliza-se o IPMItool para acesso aos sensores internos, e neste caso, a taxa de amostragem limita-se à taxa do IPMItool (1 a 3 Hz).

**Módulo de monitoramento em GPU**: este módulo é similar ao NVSMI (Seção 3). Porém, suas principais características são: i) ter sido programado para coletar um conjunto menor de parâmetros, sobre a utilização dos núcleos e da memória da GPU (total, livre, usada e tempo gasto com operações de leitura e escrita em memória principal), reduzindo assim a sobrecarga sobre o consumo de recursos do sistema; ii) ser capaz de monitorar aplicações com alta precisão, pois possui uma taxa de amostragem, em média, de 500 amostras por segundo. Como este módulo também coleta energia através do IPMItool, existe a mesma limitação do módulo em CPU na taxa de amostragem, quando o método de energia é ativado.

**Módulo de análise e visualização**: este módulo foi desenvolvido como um servidor *web*, oferecendo aos usuários acesso aos dados experimentais a partir da geração de gráficos [Silva et al. 2017]. As tecnologias utilizadas em sua implementação formam um conjunto de diferentes *frameworks*, os quais são utilizados para o funcionamento do servidor (*Angularjs*, *jQuery* e *Node.js*), para a geração dos templates dos gráficos (*Canvas.js*) e para que o servidor seja responsivo (*Bootstrap*). Como os templates do *Canvas.js* interpretam dados de entrada apenas no formato JSON, os dados de saída gerados pelos módulos de monitoramento, devem estar serializados neste formato.

### 4.3. Experimentos

Nesta seção é apresentado o resultado do experimento utilizando os módulos de monitoramento em CPU e GPU e o módulo de análise e visualização de dados. Também é apresentado a medida do *overhead* dos módulos de coleta, demonstrando que o SMCis é capaz de realizar o monitoramento com altas taxas de amostragem e baixa sobrecarga.

Para os experimentos foi executada a aplicação LUD da suite Rodinia[Che et al. 2009], que é um algoritmo que decompõe uma matriz como o produto de uma matriz triangular inferior e uma superior, possibilitando resolver um sistema de equações lineares [Silva et al. 2016]. Os testes foram realizados em uma máquina com CPU x5650 Intel(R) Xeon com 2 processadores (12 núcleos) em 2,66 GHz, 23 GB de memória principal, 32 GB/s de banda de memória e cache de 12288 KB. Além de 2 GPUs Tesla M2050-T20, com 1,15 GHz, 3 GB de memória e 448 núcleos.

#### 4.3.1. Execução em CPU

O monitoramento foi iniciado 30 segundos antes da aplicação iniciar a execução e terminou 30 segundos após o fim da execução da aplicação. Como mencionado, esses intervalos foram adotados para permitir a medição do consumo estático de potência e temperatura, já que essas informações fornecidas pelo IPMItool referem-se ao consumo total do sistema. Através do consumo estático é possível realizar o cálculo do consumo dinâmico, o qual é influenciado pela execução da aplicação monitorada.

Realizando uma breve análise sobre o resultado do monitoramento (Figura 3, gerada utilizando o ambiente *GraphCis*), é possível perceber as três etapas de execução da aplicação: i) a matriz é carregada para a memória principal, entre aproximadamente, 30

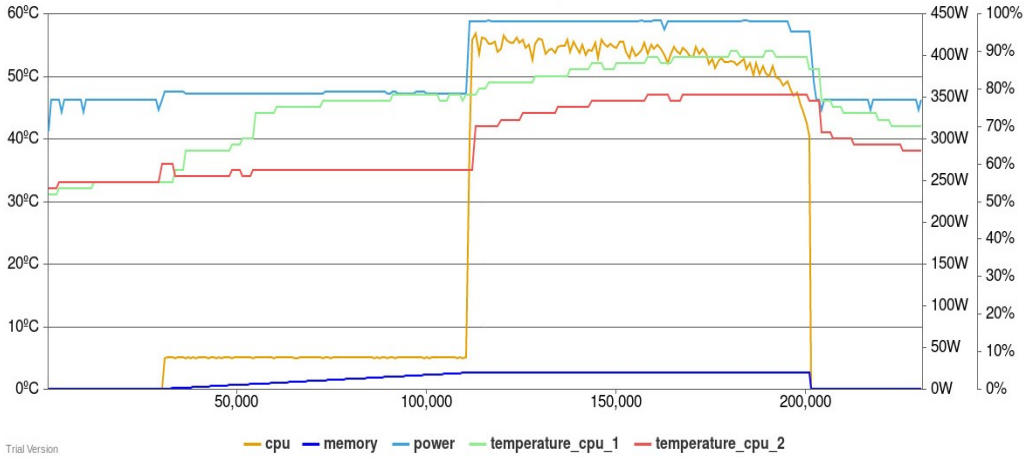



**Figura 3. Gráfico da execução em CPU.**

segundos e 110 segundos; ii) a aplicação executa a decomposição da matriz usando paralelismo, entre aproximadamente 110 segundos e 201 segundos; iii) os dados utilizados pela aplicação são limpos na memória principal a partir de, aproximadamente, 201 segundos. Foi possível analisar essas etapas relacionando o consumo de CPU e memória pela aplicação (labels CPU e memory), além das informações dos sensores (power, temperature_cpu1 e temperature_cpu2). Durante a primeira etapa, a aplicação fez pouco uso dos núcleos de processamento, onde percebe-se pouco aumento no consumo de potência e aumento na temperatura de somente uma das CPUs (temperature_cpu1). Por outro lado, quando executou a decomposição da matriz, nota-se o aumento significativo de utilização dos núcleos de processamento e consumo de potência, uma vez que todos os núcleos foram utilizados nessa etapa.

### 4.3.2. Execução em GPU

Pelo mesmo motivo do experimento em CPU (Seção 4.3.1), o monitoramento em GPU foi iniciado 30 segundos antes da aplicação iniciar a execução e terminou 30 segundos após o fim da execução da aplicação.

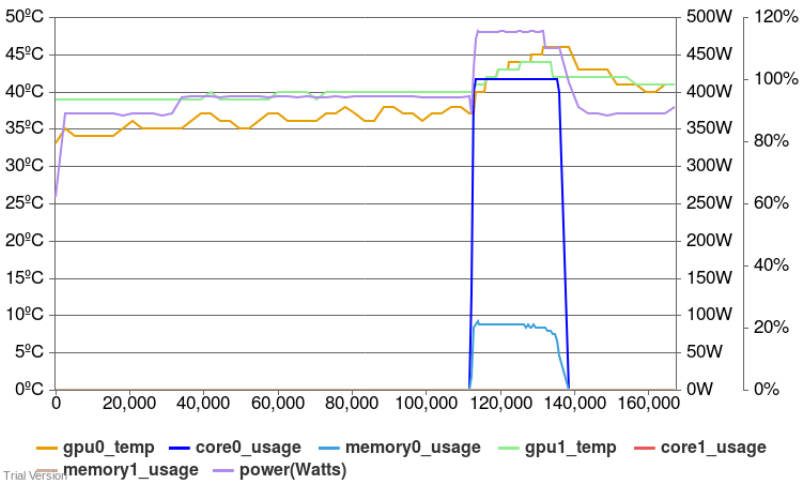



**Figura 4. Gráfico da execução em GPU.**

Realizando uma breve análise sobre o resultado (Figura 4) é possível perceber duas etapas distintas: i) a matriz é carregada para a memória principal da CPU, entre aproximadamente, 30 segundos e 110 segundos; ii) o kernel que realiza a decomposição da matriz na GPU é executado, entre aproximadamente, 110 segundos e 138 segundos. Na primeira etapa a execução ocorre de forma análoga a execução em CPU, com as GPUs em estado *idle*. Após os dados serem carregados na memória principal da CPU eles são copiados para a memória principal da GPU e, neste momento, as GPUs passam para o estado *ready*. Essas mudanças de estado foram observadas através da variação dos labels gpu0_temp e gpu1_temp. Uma vez que os dados tenham sido copiados, o kernel é executado somente na GPU 0, pois observa-se que a utilização dos núcleos dessa GPU (core0_usage) chegou a 100% e houve um aumento significativo no consumo de potência.

### 4.3.3. Overhead

Para medir o *overhead* do SMCis sobre o consumo de recursos, foram realizados experimentos monitorando os processos dos módulos de coleta. O módulo de monitoramento em CPU apresentou consumo de CPU em média de 0,12% e uso de memória principal de 0,05%. Para o módulo de monitoramento em GPU, os consumos de CPU e memória foram, respectivamente, 0,08% e 0,01%. Esse baixo consumo de memória se deve ao fato dos módulos terem sido desenvolvidos de forma a economizarem espaço em memória, equilibrando operações de escrita em disco. Também foi monitorado o consumo médio de CPU dos subprocessos que fazem chamada ao IPMItool, que apresentou taxa de 0,55%.

## 5. Considerações Finais

Neste artigo foi apresentado o SMCis, um sistema que possibilita a coleta dos parâmetros de desempenho e energia sobre a execução de uma aplicação, com portabilidade para diferentes arquiteturas, altas taxas de amostragem e baixo impacto no consumo dos recursos computacionais, além de possuir fácil utilização e permitir relacionar graficamente todos os parâmetros relevantes simultaneamente. Este ambiente, desenvolvido com foco em se obter eficiência e eficácia, vem agilizando todo o processo de monitoramento dos diferentes conjuntos experimentais gerados pela pesquisa.

## Agradecimentos

Este trabalho recebeu apoio financeiro do CNPQ, do Programa EU H2020 e do MCTI/RNP-Brasil no âmbito do projeto HPC4e, contrato de subvenção No.689772.

## Referências

[HPC 2017] (2017). High performance computing for energy (hpc4e). `https://hpc4e.eu`.

[Ext 2017] (2017). Wattsup? pro. `http://www.wattsupmeters.com`.

[Adhianto et al. 2010] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. (2010). Hpctoolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701.

[Alvin et al. 2010] Alvin, K., Barrett, B., Brightwell, R., Dosanjh, S., Geist, A., Hemmert, S., Heroux, M., Kothe, D., Murphy, R., Nichols, J., Oldfield, R., Rodrigues, A., and Vetter, J. S.

(2010). On the path to exascale. *International Journal of Distributed Systems and Technologies*, 1(2):1–22.

[Ashby et al. 2010] Ashby, S., Beckman, P., Chen, J., Colella, P., Collins, B., Crawford, D., Dongarra, J., Kothe, D., Lusk, R., Messina, P., and Others (2010). The opportunities and challenges of exascale computing. *summary report of the advanced scientific computing advisory committee (ASCAC) subcommittee at the US Department of Energy Office of Science*.

[Bedard et al. 2009] Bedard, D., Fowler, R., Lim, M. Y., and Porterfield, A. (2009). Powermon 2: Fine-grained, integrated power measurement. Technical Report TR-09-04, RENCI Technical Report.

[Bergman et al. 2008] Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., et al. (2008). Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15.

[Bridges et al. 2016] Bridges, R. A., Imam, N., and Mintz, T. M. (2016). Understanding gpu power: A survey of profiling, modeling and simulation methods. *ACM Comput. Surv.*, 49(3):41:1–41:27.

[Che et al. 2009] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee.

[ComCiDis 2018] ComCiDis (2018). Smcis: um sistema para o monitoramento de aplicações científicas em ambientes hpc. Technical report, Laboratório Nacional de Computação Científica, Petropolis - RJ. `https://github.com/ViniciusPrataKloh/SMCis`.

[de Oliveira 2016] de Oliveira, V. D. (2016). Alocação de ambientes virtuais baseados em perfils através do monitoramento e aprendizado de padrões de consumo de recursos em cmpd. Dissertação de Mestrado, IME-RJ.

[Ferro 2015] Ferro, M. (2015). *Avaliação de Sistemas de Computação Massivamente Paralela e Distribuída: Uma metodologia voltada aos requisitos das aplicaçõeses científicas*. Tese de doutorado, Laboratório Nacional de Computação Científica, Petrópolis - RJ.

[Ferro et al. 2016] Ferro, M., Nicolás, M. F., del Rosario Q. Saji, G., Mury, A. R., and Schulze, B. (2016). Leveraging high performance computing for bioinformatics: A methodology that enables a reliable decision-making. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colômbia, May 16-19, 2016*, pages 684–692. IEEE Computer Society.

[Ferro et al. 2017] Ferro, M., Silva, G. D., Klóh, V. P., and Schulze, B. (2017). *Challenges in HPC Evaluation: Towards a methodology for scientific applications' requirements*. IOS Press, Amsterdam. accepted to publish.

[Ge et al. 2009] Ge, R., Li, D., Chang, H.-C., Cameron, K. W., Feng, X., and Song, S. (2009). Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel & Distributed Systems*, 21:658–671.

[Guthrie 2013] Guthrie, M. (2013). *Instant Nagios Starter*. Packt Publishing.

[Klôh et al. 2016] Klôh, V. P., Ferro, M., Silva, G. D., and Schulze, B. (2016). Performance monitoring using nagios core. Relatórios de Pesquisa e Desenvolvimento do LNCC 03/2016, Laboratório Nacional de Computação Científica, Petropolis - RJ. `www.lncc.br`.

[Kogge et al. 2008] Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., Karp, S., Keckler, S., Klein, D., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, R. S., and Yelick, K. (2008). Exascale computing study: Technology challenges in achieving exascale systems. Technical report, DARPA IPTO, Air Force Research Labs.

[Labasan 2016] Labasan, S. (2016). Energy-efficient and power-constrained techniques for exascale computing. Available at `http://www.cs.uoregon.edu/Reports/ORAL-201610-Labasan.pdf` (2017/05/18). Oral Comprehensive Exam.

[Licht 2014] Licht, F. L. (2014). *Afinidade de Tipos de Aplicações em Nuvens Computacionais*. Tese de doutorado, Universidade Federal do Paraná, Curitiba.

[Messina 2017] Messina, P. (2017). The exascale computing project. *Computing in Science Engineering*, 19(3):63–67.

[Nagios Team 2016] Nagios Team (2016). Nagios Core Documentation.

[NVIDIA 2012] NVIDIA (2012). *NVML API Reference Manual*. `http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/NVML/nvml.pdf`.

[NVIDIA 2017] NVIDIA (2017). Nvidia system management interfaces. `https://developer.nvidia.com/nvidia-system-management-interface`.

[Patterson 2013] Patterson, D. (2013). *Orgins and Vision of the UC Berkeley Parallel Computing Laboratory*, chapter 1, pages 11–42. Microsoft Corporation, 1 edition.

[Rajovic et al. 2013] Rajovic, N., Carpenter, P. M., Gelado, I., Puzovic, N., Ramirez, A., and Valero, M. (2013). Supercomputing with commodity cpus: Are mobile socs ready for hpc? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC'13, pages 40:1–40:12, New York, NY, USA. ACM.

[Reed et al. 1992] Reed, D. A., Aydt, R. A., Madhyastha, T. M., Noe, R. J., Shields, K. A., and Schwartz, B. W. (1992). An overview of the pablo performance analysis environment. *Department of Computer Science, University of Illinois*, 1304.

[Reinders 2005] Reinders, J. (2005). Vtune performance analyzer essentials. `http://nacad.ufrj.br/online/intel/vtune/Essentials_Excerpts.pdf`.

[Rodola 2018] Rodola, G. (2018). psutil documentation. `https://media.readthedocs.org/pdf/psutil/latest/psutil.pdf`.

[Shende and Malony 2006] Shende, S. S. and Malony, A. D. (2006). The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311.

[Silva et al. 2017] Silva, G. D., Ferro, M., Oliveira, V. D., Klôh, V. P., Yokoyama, A., and Schulze, B. (2017). Estudo de abordagens de monitoramento de desempenho e energia para computação científica. Zenodo. `https://doi.org/10.5281/zenodo.1035164`.

[Silva et al. 2016] Silva, G. D., Klôh, V. P., Ferro, M., and Schulze, B. (2016). Abordagens de monitoramento de desempenho em apoio a pesquisa científica. In *Anais do XVII Simposio em Sistemas Computacionais de Alto Desempenho*, pages 74–79, Aracaju-SE. SBC.

[Simon 2013] Simon, H. D. (2013). *Barriers to Exascale Computing*. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Zomaya and Lee 2012] Zomaya, A. Y. and Lee, Y. C. (2012). *Energy Efficient Distributed Computing Systems*. Wiley-IEEE Computer Society Pr, 1st edition.

# Phase Detection and Analysis Among Multiple Program Inputs

**Rafael Mendonça Soares**[1], **Luis Fernando Antonioli**[1], **Emilio Francesquini**[1,2], **Rodolfo Azevedo**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Av. Albert Einstein, 1251 – Cidade Universitária – Campinas – SP – Brazil

[2]Centro de Matemática, Computação e Cognição – Universidade Federal do ABC (UFABC)
Av. dos Estados, 5001 – Santo André – SP – Brazil

{rafael.soares,luis.antonioli}@students.ic.unicamp.br
e.francesquini@ufabc.edu.br, rodolfo@ic.unicamp.br

***Abstract.*** *Phase analysis has been shown to be an efficient technique to decrease the time needed to execute detailed micro-architectural simulations. The SimPoint method selects small but representative portions of code from each execution phase to extrapolate performance and behavior results with high accuracy and fast execution time when compared to a complete execution of the code. However, the current SimPoint technique is limited to a single program and a single input. In this work we detail an analysis technique which is able to determine phase equivalence among multiple inputs for the same program and, consequently, avoid the redundant execution of the phases. We evaluate our proposal using SPECint 2006 with multiple inputs and show that our technique, while maintaining the precision of the original approach, reduces in 32% the number of SimPoints, on average, thus also improving performance proportionally.*

## 1. Introduction

Performance evaluation plays a fundamental role in the development of new computational systems, specially during design space exploration. A commonly used technique consists of realistic benchmark executions using detailed software models. However, not only do newer computational systems grow more complex each year, but also benchmarks grow in complexity and size. These factors have an unavoidable effect on designers productivity when we consider the higher time needed to obtain simulation results. Several solutions have been proposed to reduce this time [Eeckhout 2010].

A well-known approach is based on the simulation of small fractions of a given program and input. One of the main challenges involved in the use of this technique is to find which portions of code best represent the full execution of a program that can be used to extrapolate its full execution performance with good precision. SimPoint [Sherwood et al. 2002] is a technique that finds these portions of code based on program execution profile, which makes it independent of the micro-architecture. To do so, dynamic instructions of the program are divided into intervals – instruction sequences of the same length. These intervals are grouped into phases with similar behavior (*e.g.*, cache behavior, branches, IPC, energy consumption). For each phase, one representative interval is chosen (SimPoint), and used for the execution of a detailed simulation.

Interval clustering is made using a structure known as *Basic Block Vector* (BBV). BBV is a vector whose elements represent each basic block (BB) of the program. Each interval is characterized by a BBV, filled with the number of times each basic block was executed weighted by the basic block size. Indeed, SimPoint's authors argue that there is a strong correlation between the paths taken by the program and the expected architectural behavior [Lau et al. 2005]. Thus, interval clustering of similar BBVs can be used to cluster intervals with similar behaviors.

Indeed, the methodology presented by Sherwood *et al.* is capable of finding *Simulation Points* for a single program-input pair, and enables designers to obtain execution results quickly and within a good margin of accuracy when compared to a complete execution. However, designers often employ multiple inputs to reach a conclusion, which makes sense when we consider the number of variables and parameters involved in the micro-processor development project space exploration. In this case, there is still the need to reduce the number of simulated instructions required in each simulation round.

In this work we improve on the original SimPoint technique to analyze a program and multiple inputs at the same time, instead of a single program-input pair. Our goal is to find a SimPoint set that is representative for more than one input for the same program. The rationale is that, if the same simulation points can be used to characterize more than one input, we can decrease the total number of SimPoints needed to estimate the behavior of a given program.

We evaluated our proposal experimentally and found out that we could reduce the number of SimPoints in 32% on average while at the same time keeping the precision of the simulation on average at 0.5% of the original. We also discovered that the new clustering results in more representative phases, allowing regions considered as outliers in the single input technique, to find a better fitting SimPoint in our proposal.

The remainder of this paper is organized as follows. Section 2 presents related work and is followed by Section 3 which formalizes the phase classification problem for multiple program inputs. Then, in Section 4 we present our proposal for the problem formalization outlined in the previous section. Experimental evaluation results are presented in Section 5 and we finally conclude in Section 6.

## 2. Related Work

Some researches identify program phases by inspecting microarchitecture-dependent program behavior, such as CPI and cache miss rates [Balasubramonian et al. 2000]. Although these metrics may find cross-program phase similarity, they do not allow the phase classification to be used across different hardware design. An alternative approach is to model intervals using microarchitecture-independent metrics, such as working set signatures [Dhodapkar and Smith 2002], basic block vectors (BBVs) [Sherwood et al. 2003], and memory reuse distance [Shen et al. 2004].

The authors of PinPoints [Patil et al. 2004] were the first to propose the use of BBVs for finding cross-input similarity. They presented a superficial evaluation of the effectiveness of this technique for the SPEC2000 programs with multiple inputs. They compared the delta in CPI when using a maximum of 10 SimPoints for single program-input pair against 20 SimPoints for a set of program-input pairs of the same program.

[Eeckhout et al. 2005] presents the closest phase sharing analysis when compared to our own. The authors find cross-input and cross-program similarity at the level of intervals by characterizing intervals using a set of architecture independent characteristics such as instruction mix and register dependency distance. They presented a detailed comparison of how phases are shared among inputs and programs.

The main contributions of this work are:

- We show a detailed framework for using BBVs for capturing phase sharing among different inputs of a single program
- When considering the cross-input phase behavior similarity, we proposed to increase the number of SimPoints allowed per program proportionally to the number of inputs
- We show how phases are shared among inputs using BBVs as a phase metric for comparison of intervals

## 3. Problem Statement

Let $P$ be any program and $P_D = (i_1, i_2, ..., i_n)$ the instructions that were executed by $P$ for a given fixed input $D$, or simply its *instruction trace*. Also consider the collection of all the static basic blocks of $P$, let $(BB_1, BB_2, ..., BB_r)$ be that collection. Given $m$, the SimPoint methodology divides $P_D$ in disjoint subsets of size $m$. Let $I = (I_1, I_2, .., I_l)$ where $l = \lceil \frac{n}{m} \rceil$, the sets resulting from this division. After the nomenclature established by Sherwood *et al.* , we call these sets *instruction intervals*. For each interval $I_k \in I$, a basic block vector $BBV_k$ is generated, such that each position $BBV_k(j)$ represents the number of times the basic block $BB_j$ was executed in the interval $k$ multiplied by $|BB_j|$. Let $intervals(P, e, m)$ be a function that takes a program $P$, an input $e$, and an integer $m$ and returns the set $I$ we described above.

The main objective of the SimPoint technique is to find a partition of $I$ such that each set of this partition contains intervals with similar BBVs and, as consequence, present a similar architectural behavior. Let $(A_1, A_2, ..., A_k)$ be a partition of $I$. Each set $A_i$ contains intervals of a program phase. The first step consists in selecting a representative interval (SimPoint) from each phase for simulation. A partition of size $k$ implies $k$ SimPoints. Let $(sp_1, sp_2, .., sp_k)$ be the representatives of each one of the phases. Each result of the execution metric of a SimPoint of a phase $A_i$ is weighted by $w_i$ according to the phase coverage size, *i.e.*:

$$w_i = \frac{|A_i|}{|I|} \tag{1}$$

If we take, for instance, $CPI$ as a metric to be evaluated using the SimPoint methodology, the $CPI$ of the complete execution can be estimated by the following equation:

$$CPI(P, e) \approx w_1 \times CPI(sp_1) + w_2 \times CPI(sp_2) + ... + w_k \times CPI(sp_k) \tag{2}$$

Now, consider a set of inputs $E_P = (e_1, e_2, ..., e_n)$ of program $P$. Let $I'$ be the set resulting of the union of all the input intervals in $E_P$, *i.e.*, $I' = \bigcup_{e \in E_P} intervals(P, e, m)$.

This work finds a partition of the set $I'$, clustering all the intervals from all inputs. The objective is to find all distinct phases of $P$ considering the inputs of $E_p$. We are looking for all equivalent phases of execution common to every input.

Additionally, this work also finds a weight matrix $W$ in which the lines represent the inputs of $E_P$. and the columns represent the chosen SimPoints. Thus, $W(i)(j)$ represent the weight of the SimPoint $j$ for the input $i$.

$$W = \begin{array}{c} \\ e_1 \\ e_2 \\ \vdots \\ e_n \end{array} \begin{array}{c} \begin{array}{cccc} sp_1 & sp_2 & \dots & sp_k \end{array} \\ \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,k} \\ w_{2,1} & w_{2,2} & \dots & w_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,1} & w_{n,2} & \dots & w_{n,k} \end{pmatrix} \end{array}$$

Once again, if we take $CPI$ as the metric, the complete execution of $P$ for the inputs in $E_P$ can be estimated using $k$ SimPoints as:

$$\begin{pmatrix} CPI(e_1) \\ CPI(e_2) \\ \vdots \\ CPI(e_n) \end{pmatrix} \approx \begin{array}{c} \\ e_1 \\ e_2 \\ \vdots \\ e_n \end{array} \begin{array}{c} \begin{array}{cccc} sp_1 & sp_2 & \cdots & sp_k \end{array} \\ \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,k} \\ w_{2,1} & w_{2,2} & \dots & w_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,1} & w_{n,2} & \dots & w_{n,k} \end{pmatrix} \end{array} \times \begin{pmatrix} CPI(sp_1) \\ CPI(sp_2) \\ \vdots \\ CPI(sp_k) \end{pmatrix} \tag{3}$$

A trivial partition of $I'$ can be obtained using the SimPoint methodology individually for each input. In this case, values of the line $W(e)$, $e \in E_P$ are zero for the SimPoints of the inputs in $E_P \setminus e$. To illustrate this case, take for instance a program $P$ with an input set $E_P = \{e_i, e_j, e_k\}$. Considering that $SP(e_i) = (sp_A, sp_B, sp_C)$, $SP(e_j) = (sp_D, sp_E)$ and $SP(e_k) = (sp_F, sp_G)$, the matrix $W$ would be:

$$W = \begin{array}{c} \\ e_i \\ e_j \\ e_k \end{array} \begin{array}{c} \begin{array}{ccccccc} sp_A & sp_B & sp_C & sp_D & sp_E & sp_F & sp_G \end{array} \\ \begin{pmatrix} 21\% & 43\% & 36\% & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 38\% & 62\% & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 77\% & 23\% \end{pmatrix} \end{array}$$

The trivial matrix produces the same results of the technique presented by [Sherwood et al. 2002]. However, this work clusters all intervals of every input, enabling us to find different phases for different inputs. In other words, a subset of a partition of $I'$ may contain distinct input intervals. The objective is to find the phases presented by an input set and choose a representative interval for each one of these phases. This means that a SimPoint can be used to estimate the execution of two or more entries.

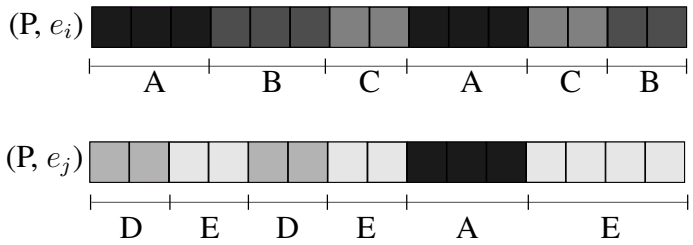



Figure 1. Phases of program $P$ with inputs $e_i$ e $ej$

Figure 1 illustrates our approach. Vertical lines show the division in intervals. The input $i$ presents 15 intervals such that: 6 intervals were clustered in phase $A$, 5 intervals in phase $B$ and 4 intervals phase $C$. The input $j$ also resulted in 15 intervals clustered in phases $D$, $E$, and $A$. Therefore, 3 input intervals of the input $j$ present similar behavior of the intervals of phase $A$ of the input $i$, which results in the creation of a single phase $A$. For this example, the matrix $W$ would have the form:

$$W = \begin{array}{c} \\ e_i \\ e_j \end{array} \begin{array}{c} \begin{array}{ccccc} sp_A & sp_B & sp_C & sp_D & sp_E \end{array} \\ \begin{pmatrix} 40\% & 33\% & 27\% & 0 & 0 \\ 21\% & 0 & 0 & 26\% & \%53 \end{pmatrix} \end{array}$$

For this particular example, the result of a SimPoint execution metric of phase $A$ has distinct values for inputs $i$ and $j$.

## 4. Multiple-Input Phase Classification

Our approach to find every distinct phase from a program $P$ for an input set $E_P$ employs the same comparison metric and clustering methodology described by [Sherwood et al. 2002]. For each interval $I_k \in I'$ a basic block vector $BBV_k$ is generated in a way such that each position $BBV_k(j)$ represents the number of times the basic block $BB_j$ was executed in the interval $k$. Then, the `k-means` clustering algorithm divides the intervals into phases based on the Euclidian distance among BBVs. Later, a single interval, the one closest to the cluster center (centroid), is chosen to represent each phase. Finally, a detailed simulation is executed using each SimPoint and properly weighted.

In the original methodology, the interval cluster size determines the weight of each phase. On the other hand, in our version the weight of each SimPoint for a given input $e$ is defined by the number of input intervals of $e$ contained by cluster $c$. Figure 2 shows a clustering example for three inputs which resulted in phases $A$, $B$, and $C$ along with the chosen SimPoints (centroids) for each phase. In this example the weight of *SimPoint A* for *Input 1* is proportional to the number of intervals of *Input 1* present in cluster $A$, the weight of *SimPoint A* for *Input 2* is proportional to the number of intervals of *Input 2*, and so on.

Let $(A'_1, A'_2, ..., A'_k)$ be a partition of the set $I'$. Each $A'_i$ represents a distinct phase presented by the program by the execution of one or more inputs of $E_P$. Formally, $w(i)(j)$ which represents the weight of the SimPoint $j$ for the input $i$ can be defined as:

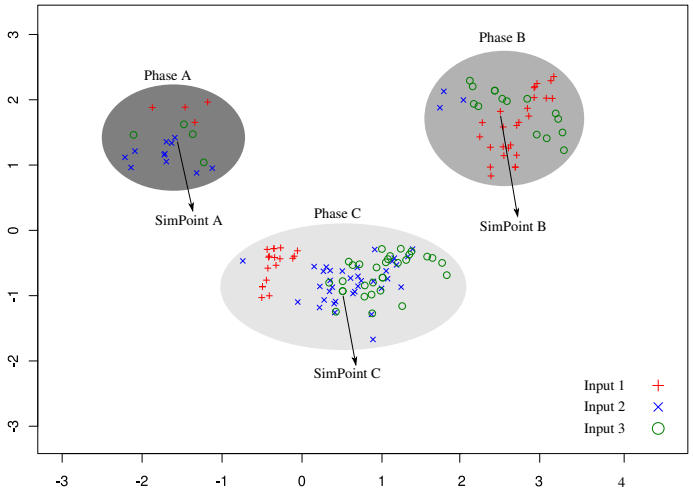



Figure 2. Phases $A$ and $B$ generated by the clustering of distinct input intervals of a same program.

$$w(i,j) = \frac{|intervals(P,i,m) \cap A_j|}{|intervals(P,i,m)|} \tag{4}$$

The SimPoint methodology requires information about the basic blocks executed in each program. Furthermore, workloads usually require a complex execution environment that cannot easily be reproduced on any simulator. PinPoint [Patil et al. 2004] is a tool that addresses such problems, it is built upon SimPoint methodology and Pin (a dynamic binary instrumentation framework) [Luk et al. ]. It automatically (i) logs a program execution, (ii) profiles the application, (iii) detects representative regions, (iv) generates traces for the regions and (v) validates the results using hardware performance counters. In addition, these five tools can be executed independently.

For the purposes of this study, the PinPoint framework has been modified to find SimPoints among multiple program inputs. Steps (i), (ii) and (iv) were used without modification. First, steps (i) and (ii) are run for all inputs of a program. Step (ii) outputs a `.bb` file such that each line represents an execution interval and stores the number of times each basic block was executed during that interval. We then merge all the `.bb` file generated for the multiple inputs of a program. We also keep a record of the number of intervals of each input. We then apply (iii) on the merged `.bb` file.

Step (iii) creates the `.simpoints` and `.weights` files. Each simulation point in the `.simpoints` file contains the number of intervals from the first interval to reach the start of the simulation point. Based on this information and the number of intervals of each input, we are able to infer the input each interval belongs to. The `.weights` are the percentage of intervals of execution being represented by each simulation point. Since the SimPoint methodology was run on the merged BBVs those weights are useless. PinPoint also provides the information about the phase each interval has been assigned to. Based on this information we weight each SimPoint for every input according to Equation 4.

Next, we apply (iv) on every input to the generated SimPoints. Finally, in (v) we evaluate the accuracy of the SimPoints by running the whole program and the SimPoint

regions using Sniper (a x86 simulator to which PinPoint was integrated) [Carlson et al. 2011].

## 5. Experimental Evaluation

We analysed our proposal using SPECint 2006 reference programs that have multiple inputs `perlbench`, `bzip2`, `gcc`, `gobmk`, `hmmer`, `h264ref`, and `astar`. We used the PinPoint framework to collect all the basic block vector profiles and generate the representative regions. We evaluate our methodology using the micro-architectural simulator Sniper. After identifying program phases, we used Sniper to compute CPI for SimPoint regions and the whole program. The baseline micro-architectural model used was `nehalem-lite` which is included along with the simulator software.

We compared our technique to the original SimPoint technique, *i.e.*, we applied it to each input separately. When running PinPoint, the user has to specify three parameters: an interval size, a warm-up size and an upper limit on the number of cluster $maxK$, which is essentially the maximum number of distinct phases to be selected by SimPoint. Thus, if a program has $N$ inputs, it will produce a maximum of $N \times maxK$ SimPoints.

To fairly compare the two SimPoint methods (single input and multiple input) we used the same SimPoint configurations of interval and warmup for both techniques. We set the interval size to 35 million instructions and warm up to 1 million instructions. We limited SimPoint's maximum number of clusters to 30 for the single input methodology. As outlined in Section 4, we found a clustering of the intervals from multiple inputs, based on the merged BBVs from all inputs, using the same clustering algorithm from the original methodology. Therefore, we also have to pick a `maxK` for this clustering. In our technique, for a program with $N$ inputs, we defined the maximum number of clusters to $30 \times N$ and $20 \times N$.

### 5.1. Experimental Results

To evaluate our approach, we consider three main aspects: (i) the total number of SimPoints, (ii) the difference in precision between our approach and the original technique, and (iii) phase equivalence between the multiple inputs. The first aspect is important because it directly relates to the simulation time, while the second determines the feasibility of our approach. The third aspect, on the other hand, gives insights into the behavior of the programs when we vary the inputs and how those changes influence the generation of SimPoints.

### 5.2. Comparison of the Number of Simulation Points

The number of simulation points is a useful information to estimate the amount of simulation time required by both techniques. SimPoint's clustering algorithm generally picks fewer simulation points than $maxK$ (upper limit) because it usually finds a good phase characterization with fewer clusters. Figure 3 shows the number of simulation points generated for each of program in SPECint 2006 with more than one input.

Figure 3 shows that our technique is able to find a phase characterization with fewer clusters than SimPoint's original methodology. On average, our technique, compared to the original technique, used 36% fewer SimPoints for $maxK = 30$ and 47%

fewer SimPoints for $maxK = 20$ (32% fewer on average). This indicates that it can find a good phase characterization clustering of the intervals from multiple inputs of a program with fewer clusters. As result, this suggests that inputs do share phases, otherwise the number of simulation points would be close to the sum of all SimPoints applied separately.

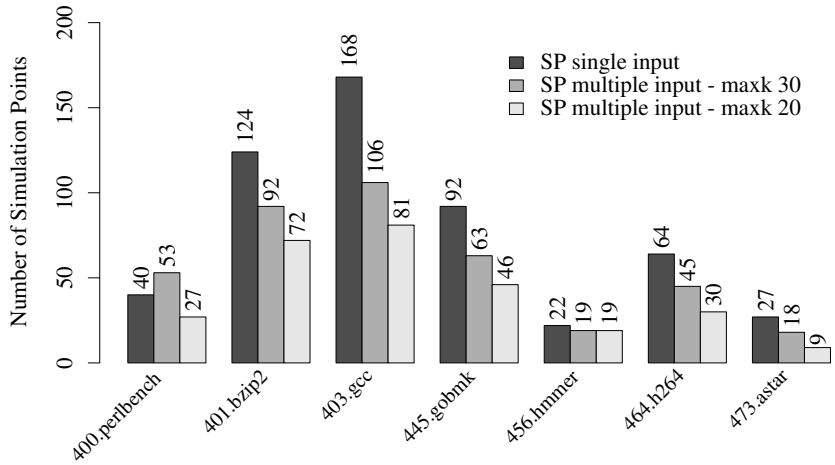



Figure 3. Number of Simulation Points

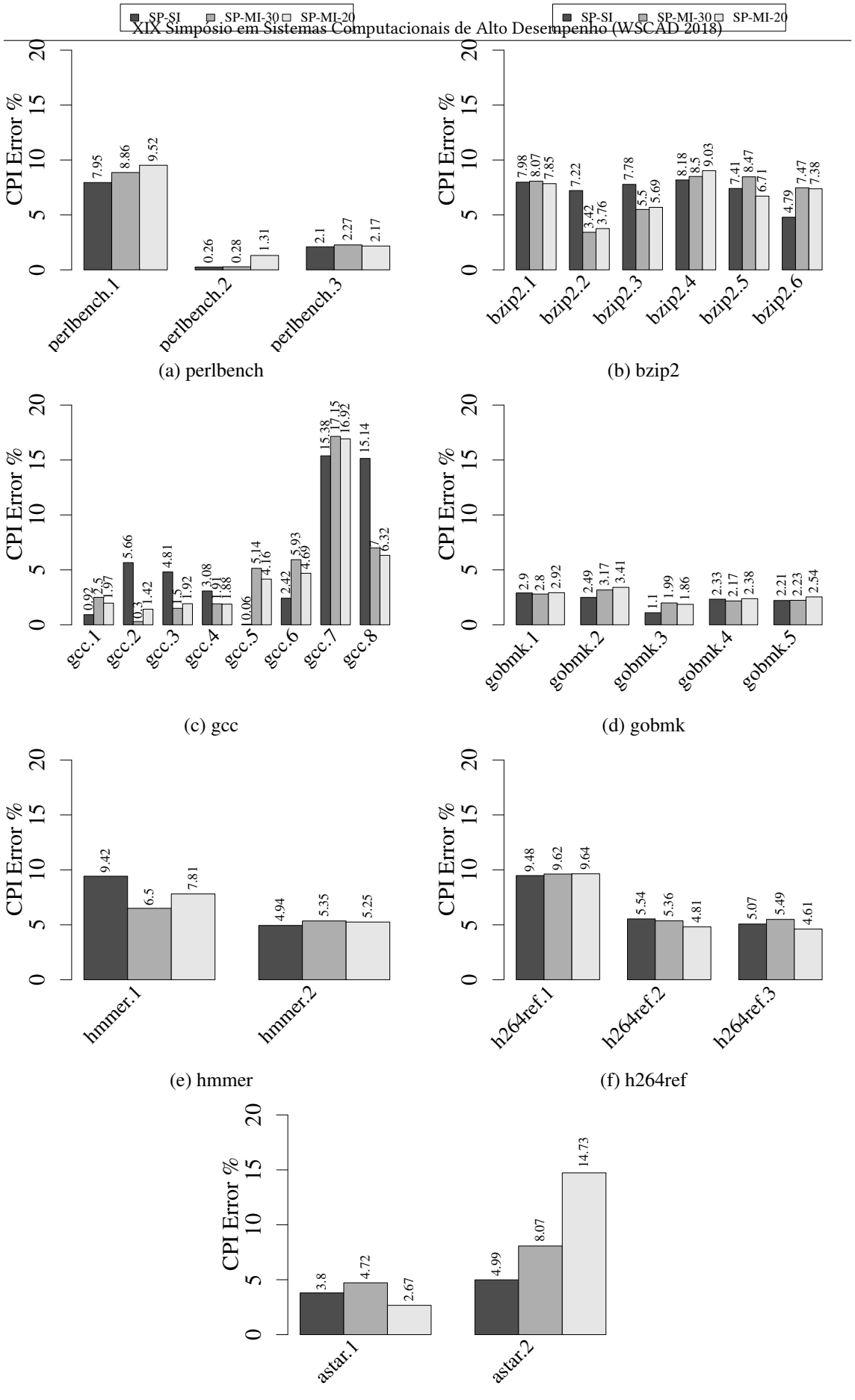
### 5.3. Comparison of Errors in CPI

Figure 4 shows the CPI error obtained by comparing a complete run to the results obtained from the SimPoint for single input and SimPoint for multiple inputs. The average error for a single input (original approach) is 5.30%; The average error in the analysis using multiple program inputs (our approach) is is 5.36% and 5.74% for $maxK = 30$ and $maxK = 20$ respectively. This implies that both techniques have similar errors. Therefore, this analysis suggests that we can reduce the number of simulation points used by a set of inputs of a program and get similar accuracy.

Indeed, even if the total number of SimPoints is lower, the number of SimPoints taken into consideration to estimate the results of each input is higher. This difference in the number of SimPoints does not change the total simulation time and offsets a potential increase in the error rate that could be caused by the use of more general SimPoints instead of more specific ones. Our experimental results show that the average CPI estimate error, when compared to the original approach, is 0.5% for $maxK = 20$ and negligible (0.06%) for $maxK = 30$.

### 5.4. Phase Sharing Analysis

In the previous sections we have shown the impact that shared phases have on the number of SimPoints and on the precision of the simulation results. Those results suggest that inputs share phases and that we can find a good phase characterization of the intervals from multiple inputs of a program. In this section we characterize how those phases are shared among inputs, *i.e.*, which phases are present in each input and their contribution to each input. To do so, we first exemplify how sharing takes place using the `astar` benchmark.




(a) perlbench (b) bzip2

(c) gcc (d) gobmk

(e) hmmer (f) h264ref

(g) astar



Figure 4. CPI evaluation for single input and multi-input ($maxK = 30$ and $maxK = 20$)

Figure 5 shows how the weights of the SimPoints differ for distinct inputs, in percentages of the execution time. The top bar represents the phases of the first input, and their weights. Only six phases play relevant role for input 1 execution. The bottom bar shows the second input. Although the light green phase (rightmost) covers more than half of the program execution, this input takes the program through more distinct phases than input 1. Recalling Figure 3, by sharing several SimPoints, we could reduce their total number (from 27 to 18, with $maxK = 30$) but still keep a similar precision.

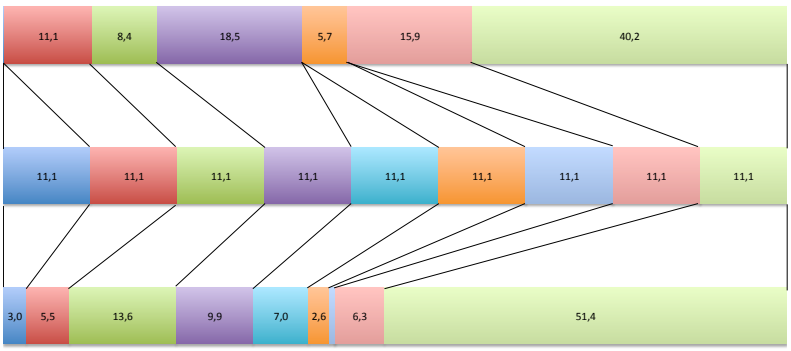



Figure 5. Distribution of SimPoints weigths for two inputs of the `astar` benchmark: input 1 on top and input 2 on bottom. Smaller weights omitted due to space restrictions.

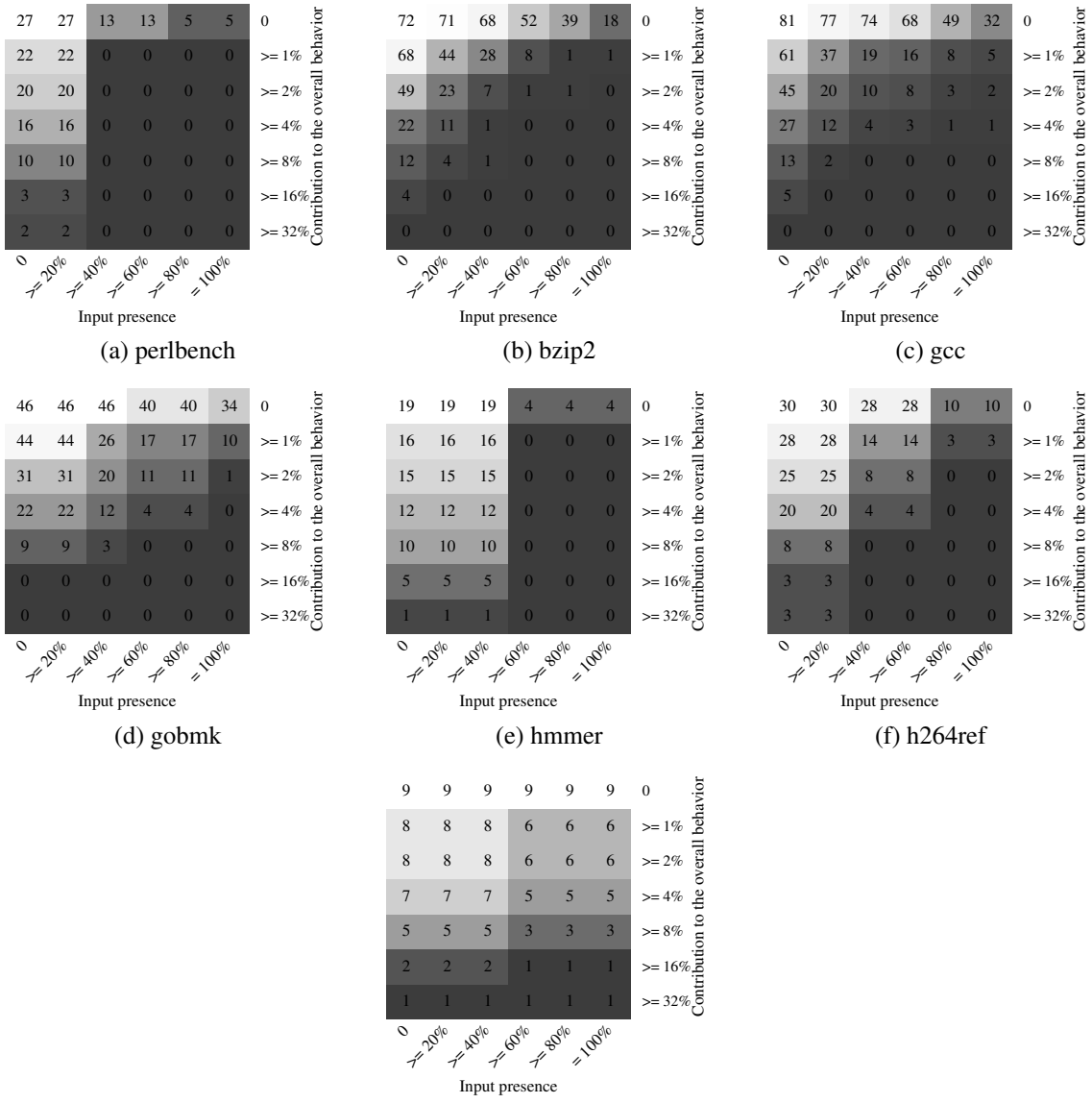
To quantify phase sharing, we characterized the number of phases and the percentage (in 20% increments) of the inputs that actually use them. Figure 6 shows our experimental results. For instance, `gcc` has 32 phases that are present in all inputs, 49 phases that are present in at least 80%, 68 phases present in at least 60%, and so on. The remainder of the results can be seen on the first line of each subfigure. Figure 6 also shows the contribution (weight) of those shared phases to the overall behavior of the inputs. These results are shown in lines (note the log axis). For instance, even though `gcc`'s inputs share 32 phases, only one of those phases covers every input with a contribution of at least 4% ($4^{th}$ line), and none has a contribution of at least 32%.

The heatmap shown in Figure 6 also allows us to quickly observe the difference in phase sharing behavior between the benchmarks. For instance, `astar` has, proportionally, a larger number of phases shared among the multiple program inputs which can be seen by the smaller number of dark cells on its heatmap. In particular, `astar` has one phase that is present in all inputs with a contribution of more than 32%, which is not the case for any other evaluated benchmark.

## 6. Conclusion

SimPoints have been extensively used to decrease simulation time and thus improve the productivity of processor designers. However, the original technique proposed by Sherwood *et al.* considers each program-input pair separately, forcing their users to employ a distinct set of SimPoints for each input of the same program. For this reason, the original technique ends up effectively disregarding similarities that are bound to exist among multiple executions with distinct inputs of the same program.

In this work we show that, when these similarities are taken into consideration, we can decrease the time needed to obtain simulation results even further. In particular, using SPECint 2006 we show that the number of SimPoints (which is directly proportional to simulation time) can be reduced on average 32% (in the specific case of `astar` we achieved a 66% reduction with $maxK = 20$). We also show that, even if the total number of SimPoints is lower, the number of SimPoints taken into consideration to estimate the results of each input can be higher, thus compensating for a possible increase in the error rate caused by the use of generic SimPoints instead of specific ones. In effect, we show that the average CPI estimate error when compared to the original approach is 0.5% for $maxK = 20$ and negligible (0.06%) for $maxK = 30$.




(a) perlbench (b) bzip2 (c) gcc

(d) gobmk (e) hmmer (f) h264ref

(g) astar

Figure 6. Heatmap showing how many phases are present and their coverage

## 7. Acknowledgments

We would like to thank CNPq, Capes (PROCAD 2966/2014) and FAPESP (2014/17925) for their support.

## References

Balasubramonian, R., Albones, D., Buyuktosunoglu, A., and Dwarkadas, S. (2000). Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO-33 2000*, pages 245–257.

Carlson, T. E., Heirman, W., and Eeckhout, L. (2011). Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11.

Dhodapkar, A. S. and Smith, J. E. (2002). Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02.

Eeckhout, L. (2010). *Computer architecture performance evaluation methods*. Morgan & Claypool Publishers, San Rafael, CA, USA.

Eeckhout, L., Sampson, J., and Calder, B. (2005). Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 2–12.

Lau, J., Sampson, J., Perelman, E., Hamerly, G., and Calder, B. (2005). The strong correlation between code signatures and performance. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, ISPASS '05, pages 236–247. IEEE Computer Society.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*.

Patil, H., Cohn, R., Charney, M., Kapoor, R., Sun, A., and Karunanidhi, A. (2004). Pinpointing representative portions of large intel®itanium®programs with dynamic instrumentation. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37.

Shen, X., Zhong, Y., and Ding, C. (2004). Locality phase prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI.

Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. (2002). Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57.

Sherwood, T., Perelman, E., Hamerly, G., Sair, S., and Calder, B. (2003). Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93.

# Performance and Energy Efficiency Evaluation for HPC Applications in Heterogeneous Architectures

**Vinícius Klôh, Daniel Yokoyama, André Yokoyama**
**Gabrieli Silva, Mariza Ferro, Bruno Schulze**

[1]Laboratório Nacional de Computação Científica (LNCC)
Getúlio Vargas, 333, Quitandinha – Petrópolis – Rio de Janeiro

{viniciusk,dmassami,andremy,gabrieli,mariza,schulze}@lncc.br

***Abstract.*** *This work aims to analyze the aspects related to the performance, without losing focus on the energy efficiency of the applications. To this end, we evaluated a representative set of experiments with three renowned benchmarks and two real world applications used by the energy industry. These experiments used a range of environments that included a medium scale HPC system with Xeon and CUDA cores and a mobile based Jetson TX2 development board composed of ARMv8 and CUDA cores. Our results enable analysing the performance and power consumption of the selected applications, and help to energy efficiecy in HPC systems.*

## 1. Introduction

Scientific computing usually requires huge processing power resources to perform large scale experiments and run simulations in a reasonable time. These demands have been addressed by High Performance Computing (HPC), which makes them invaluable tool for modern scientific research. Despite this, several areas require more computational power to perform their simulations, as expected for the computational exascale. However, such systems will be greatly constrained by power and Energy Consumption (EC) so, an approach for balancing energy efficiency and performance is required. This is a complex challenge, since power and performance are two conflicting criteria in which a reduction of power often results in a degradation of performance [Filiposka et al. 2016]. To reach this balance, it is important to understand the relationships among performance, power consumption, and the characteristics of each scientific application. The proposed approach is to study the requirements of the scientific applications, based on the applications classes named Motifs [Asanovic et al. 2009]. The Motifs' classes represent applications with similar computational and communication characteristics and its use should enable a better understanding of the scientific applications [Mury et al. 2015]. Researches suggest that representative applications of the future exascale systems can be categorized in several of these classes [Messina 2017]. Among the classes of applications are those of the energy industry, focus of this work, with its simulations requiring the computational power that will be offered by the future HPC systems. Another aspect of this scenario deals with alternative architectures considered as possible substitutes of the current general purpose high power processors for future HPC system design. In this context are the architectures with reduced EC for improved energy efficiency, such as ARM and GPGPU heterogeneous computing. This work makes an evaluation of the trade-off between performance and energy consumption for these architectures to execute some models of high performance scientific applications. Two real applications used by the energy industry,

focused particularly on applications of the seismic area for the oil and gas industry, were mapped into three renowned benchmarks (SP, BT and LU - NAS Parallel Benchmark). This was done based on the study of the Motifs' class of the real applications. These experiments are performed in multiple environments, which included a medium scale HPC system with Xeon and CUDA cores as well as a mobile based Jetson TX2 development board composed of ARMv8 and CUDA cores. Our results try to highlight the best infrastructure to perform a class of application. To this, we used three main metrics to analyze the execution of an application: the execution time, the energy consumption and the Energy Delay Product (EDP) to evaluate the trade-off between the first ones. The results show that there is a great disparity on the applications executed according to the workload size of the application, the applications' Motif, the environment used and also the goal of the experiment (execution time, energy efficiency or both using metrics such as EDP). This study point out that a deep understanding of the application, its Motif, the size, the underlying architecture and the programming model are crucial to both the performance and energy efficiency.

The remainder of this paper is organized as follows. Section 2 presents some related works. In Section 3 are the experimental setup, including the computational architectures, the applications and the methodology of the experiments. Section 4 presents the results of the experiments and finally, Section 5 presents conclusions and future works.

## 2. Related Work

Many works are focused on alternative and heterogeneous architectures using the ARM [Bez et al. 2016, Weloli et al. 2016, Puzović et al. 2016, Loghin et al. 2017] or the GPUs to improve the energy efficiency [Coplin and Burtscher 2016, Adhinarayanan et al. 2016, O'Brien et al. 2017, Ashraf et al. 2018]. Thus, due to the large number of works dealing with this issue, this section focuses only on related works discussing the performance and the energy efficiency of applications from the energy industry perspective, particularly those representing the behavior of physical phenomenona such as seismic activity.

The work [Okina et al. 2016] discusses the relationship between the power consumption and the performance of the 3D stencil computation on a FPGA accelerator focusing on the parallelization of arithmetic pipelines. They evaluated how the selection of the pipeline parameters influenced the power efficiency and the relationship between the clock frequency and the power efficiency.

In [Castro et al. 2016] is presented an approach to the Ondes3D, a seismic wave propagation simulation, using MPPA-256 and Xeon Phi manycore processors and comparing performance and energy efficiency of their solutions to the optimized solutions for multicores and GPUs. Experimental results showed that the approach to the MPPA-256 is the most energy efficient, whereas the solution for the Xeon Phi achieves a performance comparable to the state-of-the-art solution for GPUs. In terms of performance, their solution for the Xeon Phi achieved improvements with respect to MPPA-256 and general-purpose multicore processor, respectively. Similarly, [Francesquini et al. 2015] analyzes performance and the EC of the Ondes3D on multicore, NUMA, and manycore platforms. Results suggest that applications able to fully use the resources of the manycore processors can achieve better performance and may consume less energy when compared to low-power and general-purpose multicore processors.

In the work [Göddeke et al. 2013] they conducted a comparison between the ARM and the x86 architectures using three applications to numerically solve PDE problems, including a high-order spectral element method for acoustic or seismic wave propagation modelling. They evaluated weak and strong scalability on a cluster of the ARM Cortex-A9 dual-core processors and demonstrated that the ARM-based cluster can be more efficient in terms of energy-to-solution compared to a cluster of Intel Xeon X5550 processors. However, for one application that is compute-bounded on the x86 already, the difference between the peak floating point performance of the two architectures is too large to achieve a gain in terms of energy efficiency.

As the previous works, we have focused on the models of applications focused on the seismic wave propagation simulations and on the energy and performance efficiency. Despite those works mentioning the concern with the EC to make the exascale feasible, in exception to [Okina et al. 2016], they don't include real applications in their experiments and different implementation models. Thus, this work improves the related works by: i) categorizing the Motifs of such real applications and comparing the results of the benchmarks with the real applications; ii) by using a broader range of environments, studying how the analyzed applications behave on a variety of architectures; and iii) proposing, according to the metric used by the evaluation, the ideal systems to achieve optimal results.

## 3. Experimental Setup

This section describes the computational architectures, the applications and the methodology used in the experiments both to define the set of the real and the theoretical applications (benchmarks) as well as to measure energy consumption in these experiments.

### 3.1. Computational Architectures

The hardware platforms used are the ComCiDis SGI Altix Cluster (CPU Intel(R) Xeon x5650 / GPU Tesla M2050), henceforth called SGI for brevity. And the development environment NVIDIA Jetson TX2 (CPU ARM A57 and NVIDIA Denver2 / GPU NVIDIA pascal), henceforth called Jetson. Details of the architectures are presented in Table 1. For CUDA implementations, the compilers used were the NVCC for the cuda kernels and the gcc/g++ for the c/cpp parts of the code and for the experiments on CPU cores.

**Table 1. Computational architectures for the experiments.**

| | **Xeon** | **Tesla M2050** | **Jetson TX2** |
|---|---|---|---|
| **Processor** | x5650 | GF100 | Cortex-A57/Denver2 |
| **CPU Clock (GHz)** | 2.66 | - | 1.88/2 |
| **CPU Cores/Threads** | 6/12 | - | 4/2 |
| **GPU** | - | - | Pascal |
| **GPU Cores** | - | 448 | 256 |
| **GPU clock (GHz)** | - | 1.15 | 1.3 |
| **Memory (ECC off) (GB)** | 23 | 3 | 8 |
| **Shared memory/Cache L1 per SM (KB)** | - | 64 (16/48) 7 | - |
| **Cahe L2 (KB)** | - | 768 | - |
| **Memory bandwidth (ECC off) (GB/s)** | 32 | 148 | 59.7 |
| **Memory clock (GHz)** | 1.33 | 1.546 | 1.866 |
| **TDP (Watts)** | 95 | 225 | 7.5 (Max-Q profile) |

### 3.2. Applications

Based on the two real applications from the energy industry, which are our main focus, and its respective Motifs' class we defined the benchmarks to complement the experimental set [1]. So, the intent is to analyze a greater set of applications that represents the energy industry in order to study more in depth these kind of applications. The approach used in this work is to understand the computational requirements of the applications from their classification as a Motif's class. These classification characterizes applications by common requirements in terms of computation and data movement. From this, the real applications were mapped into a Motif's class and based on this class the benchmarks representing these applications were also defined.

The following real applications, 3D Wave Propagation and Full Waveform Inversion (FWI), are focused particularly on applications of the seismic area for the oil and gas industry. The 3D Wave Propagation simulates the propagation of a single wavelet over time by solving the acoustic wave propagation equation. This application is being used by a Brazilian oil and gas industry company. The equation is solved by using finite differences that have a high degree of parallelism, given the interdependence between the data [Menezes et al. 2012]. The Full Waveform Inversion is a method that potentially allows to extract more information from the seismic data. Numerical solvers for the wave equation are a key component of FWI. The main computational cost of a wave-equation solver stems from the computation of the Laplacian at each time step. These two applications can be characterized as a Structured Grid (SG) computation (SG Motif's class).

The NAS SP is a representative benchmark for the SG class, while the BT and the LU are for the DLA class. Although the real applications used in this work are not represented by the DLA class, this is an algorithm that dominates important applications for the energy industry, such as in solvers for combustion. In addition, we would like to compare the results of experiments with applications of another class than SG.

The NAS benchmark suite allows the execution of the experiments with a vast range of problem sizes. For this study the size chosen was the largest problem that all environments could execute. For the SP and LU the size 'C' and for the BT the size 'B' [2]. For the 3D Wave the problem is a grid of $512 \times 512 \times 512$ points. The FWI application has a problem size of $1200 \times 2000 \times 1200$ points.

### 3.3. Performance and Energy Evaluation

To measure the energy consumption, two metrics were collected: the idle power ($P_{idle}$) and the processing power ($P_{processing}$). The idle power is the consumption while the system is in idle, including all processes in the background. The processing power is the consumption to execute the application. The total power ($P_{total}$) was defined as the sum of the idle and the processing power, and it is used to describe the power consumed by the environment while the application is running. The power is given in Watts (W) and energy in Joules (J).

There are different ways to collect the idle power, for example, collecting the idle power when the system is idle at an arbitrary time, or collecting before and after the execution of the application, when the system is also idle. This last one is the approach used

[1] The methodology for this mapping process is presented in another work [Ferro et al. 2017]

[2] The description of the sizes can be found at https://www.nas.nasa.gov/publications/npb.htmlurl

in this work. To do that, the monitor tool described below, starts the monitoring thirty seconds before the application starts and finishes thirty seconds after the application ends. These times allows to measure the idle power and estimate the power consumed by the monitor tool and the processing power. The first sample corresponds to the power consumption before the monitor starts, that is, the idle power. The next samples correspond to the sum of the idle power and the power consumed by the monitor tool. In our methodology, this sum was defined as the idle power also, considering the monitor process as a system process in the background, because it is the consumption without processing the application. The environments were dedicated to the execution of the applications.

Collecting data from internal sensors and hardware parameters is done in a different way in each architecture. For this reason, it was developed a performance and power profiling tool based on different modules to collect these parameters. A module was developed in Python, using the psutil, to collect the hardware parameters for the SGI and for the Jetson. The energy on the SGI platform utilize the IPMITool and this module also collects information from the Jetson board that has multiple embedded monitors that allow a compartmentalized monitoring of the EC. Also, it was developed a module based on the NVML API to read the internal sensors for the SGI GPUs.

First, the power and performance profiling tools are used for each experiment, with each application, implementation model, thread level parallelism (TLP) and problem sizes. Following, the processing power is calculated through the Equation $P_{processing} = P_{total} - P_{idle}$. The power and the execution time are used to calculate the energy consumed by the application ($E_{processing}$) and the system ($E_{idle}$), using the Equations 1 and 2:

$$E_{idle} = \int_{T_{start}}^{T_{end}} P_{idle}(t)dt \tag{1}$$

$$E_{processing} = \int_{T_{start}}^{T_{end}} P_{processing}(t)dt \tag{2}$$

where $T_{start}$ and $T_{end}$ represent when the application starts and finishes, respectively.

The total energy consumed is calculated as $E_{total} = E_{idle} + E_{processing}$. The energy-performance efficiency was calculated and evaluated using the Energy-Delay Product ($EDP = Energy \times Delay$), where delay represents performance. Using this power-performance efficiency metric prevent us from choosing configurations that deliver faster execution but with much higher energy consumption.

## 4. Results

In this section are presented and compared the results for all experiments. First, the experiments for the NAS suite and following, for the 3D Wave and FWI applications.

### 4.1. Results of NAS Parallel Benchmarks (SP, BT and LU)

Figure 1 shows the execution time results for the NAS suite on the SGI and Figure 2 shows the same results on the Jetson (for both, with and without CUDA cores). The best result for the SP experiment, when executed only on the CPU, was using 6 Xeon cores. However, when the SP benchmark is executed on the GPU (CUDA cores), the result shows a greater reduction in the execution time, about $2.7\times$ lower. The SP benchmark also does not have good scalability at level of threads when using CPU (as evidence by the best

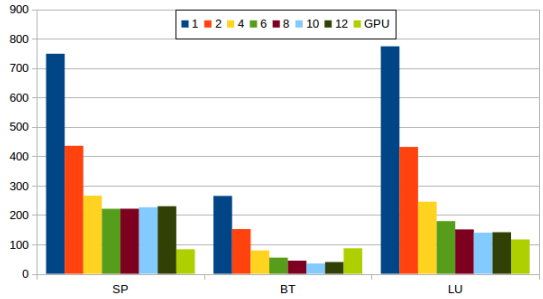



**Figure 1. Execution time for the NAS on the SGI.**

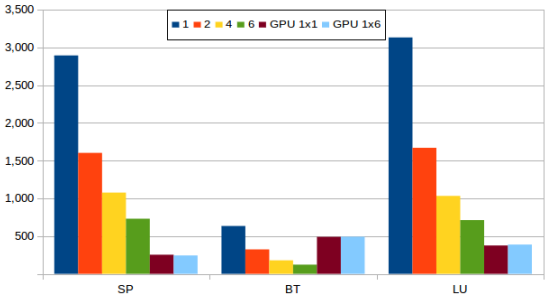



**Figure 2. Execution time for the NAS on the Jetson.**

result with 6 threads). In this case, this experiment shows that the SP benchmark could be executed on CUDA cores to obtain better results overall. The execution of the SP benchmark, exclusively on the ARM processors on the Jetson, has the best results with 6 cores (maximum available) and 6 threads. Comparing the best result of the SP, when executing on the CPU, between the SGI and the Jetson, it is observed a $3.2\times$ increase in the execution time for the Jetson board. There are two modes of execution with CUDA cores on the jetson: with a single core and with all cores turned on. These modes are chosen because the application processing is offloaded to the CUDA cores, thus the ability of hot-plugging cores could improve the performance and energy consumption. Observing the results, the execution time decreases $4\%$ when executing with all cores available. However, as observed in the following results, the EC increased.

The best result for the BT experiment is obtained with 10 Xeon cores. Unlike the SP benchmark, BT does not obtain better results when using the CUDA cores on the SGI. The best result using the CPU was about $2.5\times$ better than the result using the GPU. The execution of BT benchmark on Jetson on the ARM cores, exclusively, have the best results with 6 cores and 6 threads, even when compared to the execution on CUDA cores. The comparison of the BT benchmark execution in the SGI and the Jetson has results very close to the ones observed in the SP benchmark. The execution time is about $3.5\times$ higher. The Motif of the BT benchmark, as with the execution on the SGI with GPU, could possibly be responsible for the increase in the results with CUDA. Comparing the result on Jetson CUDA cores with the best result on the ARM cores alone, it presents a considerable increase in execution time of $4\times$. Thus, when executing this Motif, these results indicate that the execution on the ARM cores alone is preferable.

The execution of LU benchmark on the SGI, using CPU, has the best result for the execution time with 10 Xeon cores. Comparing to the execution using GPU, the gain for the execution time, is about $1.2\times$ faster. The execution of LU on the Jetson has the lowest execution time again, with 6 cores and 6 threads. The result shows that the execution time was about $5.1\times$ higher on the Jetson than on the SGI. The increase in execution time, when compared to the x86 architecture, is relatively higher than the previous benchmarks, which resulted in lower gains in the EC and the EDP. The Jetson CUDA execution time is $1.9\times$ lower when compared to the best execution with the ARM cores alone (6 cores). This result is the first that shows an improvement for executing the experiment on CUDA core on the Jetson. For the experiments with the LU, when considering the execution time, the best overall architecture observed is the SGI with GPU, presenting $3.2\times$ lower

result in comparison to the fastest execution on the Jetson board (also with GPU).

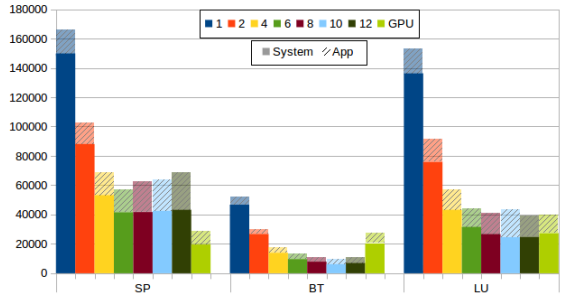



**Figure 3. Energy consumption for the NAS on the SGI.**

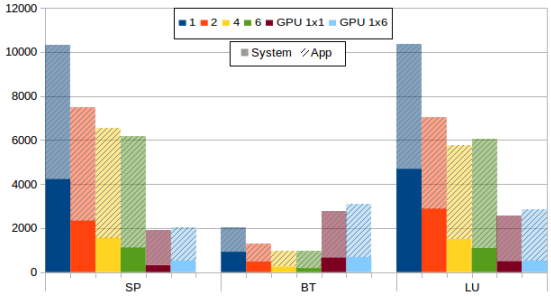



**Figure 4. Energy consumption for the NAS on the Jetson.**

Figures 3 and 4 present the EC results for the NAS suite.The hatched and solid areas displays the EC of the application and the system, respectively. The sum of both areas represents the total energy consumption of the experiment.

The EC for the NAS suite on the SGI system follows closely the results observed in the execution time. The application EC represents only a small fraction of the total power consumption. This is the opposite of when we observe the results of the Jetson, where the application is responsible for the majority of the energy consumption.

For the SP experiment on the SGI the lowest EC belongs to the execution using CUDA cores, as for the Jetson. However, due to the energy efficiency, as well as low system consumption, the Jetson with a single CPU core and CUDA execution presented the best overall consumption ($15\times$ lower). Also, worth noting is that the ability of the Jetson cores to be turned off displays good results in this experiment ($7\%$ lower).

The execution of the BT experiment on the SGI had the best results for the EC using 10 Xeon cores, while on Jetson the best execution was with all of the six cores. However, for both architectures, the CUDA execution did not present favorable results, increasing power consumption. The overall best performance is present on the Jetson with EC of about $10\times$ lower when compared with the best result using the CUDA cores. Again, the ability of the Jetson cores to be turned off displays good results ($10.35\%$ lower).

For the LU experiment on the SGI the EC was reduced when executed with 12 Xeon cores. However, the execution with CUDA cores had similar results and, as previously observed, the execution time was improved. The Jetson results display an interesting behavior. While the EC was lower with 4 cores, when dealing exclusively with the processors, the execution time was noticeably higher. Also, as with the execution time, the execution on the Jetson's CUDA cores drastically improved EC about $2.24\times$. Once again, the use of hot-plug CPUs improved energy consumption.

Figures 5 and 6 show the EDP logarithmic results for the NAS suite experiments. It is clear, when dealing with the SGI in the SP experiment, that due to the lower EC and execution time, the execution on CUDA cores was dramatically improved. The higher execution time on the Jetson CUDA cores is compensated by a much lower EC, resulting in an overall improved execution for the SP experiment with this configuration. Comparing the best results of the SGI and the Jetson architectures, when dealing with EDP there is a great advantage for the Jetson of about $5\times$ lower.

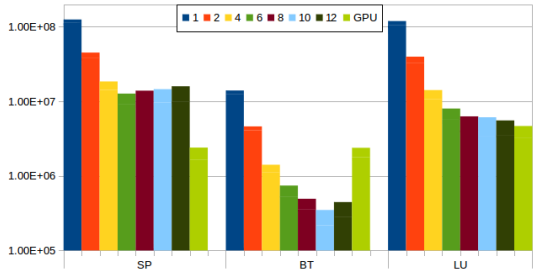



**Figure 5. EDP for the NAS on the SGI.**

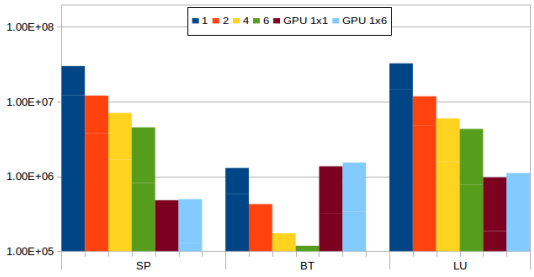



**Figure 6. EDP for the NAS on the Jetson.**

The BT experiment on the SGI has the best EDP performance for execution on 10 Xeon cores. Due to the problem size, the scalability of the experiment degrades around six cores. For the Jetson the experiment has the best performance with six ARM cores. Neither architectures were able to efficiently employ the CUDA cores for this experiment, due to high memory access. Although the execution on the SGI has the best time, due to a much lower power consumption of the Jetson architecture, the EDP for the BT experiment was much improved on the Jetson's ARM cores. Comparing the execution with 10 Xeon cores with 6 ARM cores, the EDP was reduced when executed on ARM cores in $2.95\times$.

The best LU EDP result on the SGI was on the CUDA cores, due to lower execution time, as the EC is actually lower running on 12 Xeon cores. For the Jetson, the best performance on CUDA cores and only a single ARM core turned on, as both EC and execution time were lower in this configuration. Overall, the Jetson displayed the best result, again due to much lower EC, as the execution time is $4.8\times$ lower on the SGI.

### 4.2. Synthesis of Results and Comparison with real applications

Table 2 presents all above results, as well as the comparison with real applications on the architectures studied. The table presents the results for SGI architecture based on the best execution time, and Jetson architecture presents the results based on the lowest EC. The SGI results include the number of threads (t) while the Jetson displays the number of cores (c) that achieved the best performance. This is due to the fact that the SGI architecture cannot hot-plug the cores. So, when scaling the number of cores used is always 12, even if its best performance is with a lower number of threads. The Jetson number of cores is equal to the number of optimal threads. The underlined values display the best result for an application depending on the decision factor (Time, Energy and EDP).

The use of real applications allows us to compare the results obtained with the execution of the benchmark. The two real world applications studied are of the SG Motif. While the results obtained with the 3D Wave had good co-relation with the benchmarks (both display the same best architecture based on the chosen decision factor), for the FWI, this cannot be confirmed, due to lack of CUDA implementation.

Comparing the results of the 3D Wave experiment with G size (the largest size available on all environments), the SGI achieved the best execution time for the constant (CDE) and variable (VDE) densities, about $3.3\times$ and $2.4\times$ lower respectively (NAS SP was about $3\times$ lower). However, the EC on the SGI is much higher than the Jetson for constant and variable densities, $12.6\times$ and $17.1\times$ higher, respectively (NAS SP was about $15.1\times$ higher). Due to a much lower EC compared to the increase in execution time, the

**Table 2. Summary results categorized by execution time, energy and EDP.**

| Application | Architecture | Time (s) | Energy (J) | EDP (s x J) |
|---|---|---|---|---|
| 3D Wave[3] | SGI + CUDA | 182.97 | 62189.31 | $11x10^6$ |
| | | 370.93 | 124540.35 | $46x10^6$ |
| | Jetson (6 c) + CUDA | 605.00 | 4946.74 | $30x10^5$ |
| | | 895.00 | 7285.63 | $65x10^5$ |
| FWI | SGI (12 t) | 12.94 | 3604.26 | $47x10^3$ |
| | Jetson (4 c) | 226.85 | 1089.33 | $25x10^4$ |
| NAS SP | SGI (6 t) | 221.48 | 57141.84 | $13x10^6$ |
| | Jetson (6 c) | 729,54 | 6185.77 | $45x10^5$ |
| | SGI + CUDA | 83.49 | 28590.64 | $24x10^5$ |
| | Jetson (1 c) + CUDA | 253.13 | 1896.62 | $48x10^4$ |
| NAS BT | SGI (10 t) | 35.02 | 9945.68 | $35x10^4$ |
| | Jetson (6 c) | 121.90 | 968.50 | $12x10^4$ |
| | SGI + CUDA | 86.88 | 27334.92 | $24x10^5$ |
| | Jetson (1 c) + CUDA | 489.51 | 2783.02 | $14x10^5$ |
| NAS LU | SGI (10 t) | 139.75 | 43602.00 | $61x10^5$ |
| | Jetson (4 c) | 1031.80 | 5784.27 | $60x10^5$ |
| | SGI + CUDA | 116.90 | 39828.60 | $47x10^5$ |
| | Jetson (1 c) + CUDA | 375.22 | 2575.71 | $97x10^4$ |

EDP on the Jetson is one order of magnitude lower. These results are all in agreement with the results of the NAS SP benchmark.

For the FWI application, in comparing the SGI results with the Jetson, the execution time is about $18.8\times$ higher for the Jetson. This is due to higher clock speeds on the SGI along with double number of cores. Furthermore, these results could also be attributed to a better tool-set available for the x86 architecture, such as optimized compiler and code. The EC shows an inverse result, with the Jetson consuming $3.3\times$ less energy, due to lower TDP of such boards. The EDP result shows a difference of an order of magnitude lower for the SGI. This is due to the large increase in execution time observed for the Jetson board. This could lead to the conclusion that for systems where EC is crucial, the Jetson solution appears to be the best option. However, if the execution time should be accounted in the analysis, the SGI still presents a better solution. It should be noted that advancements in the performance of the ARM architecture are being developed and the Jetson board has a GPU unit that was not used in the experiments. This unit was responsible for a large amount of the system energy used, but did not contributed to the execution of the application.

The Table 3 presents all above results with the recommended environment to be executed, according to the decision factor (time, energy and EDP).

## 5. Final Considerations and Future Works

This work presents an evaluation of the execution of scientific applications on three environments focusing on the energy efficiency as much as on the performance. To this goal, real applications and benchmarks were analyzed on a medium scale SGI HPC system and the mobile based Jetson TX2 development board. These experiments, executed on multiple heterogeneous architectures, that included CPU execution and CUDA core

[3]3D Wave application presents the first result for the execution with constant density while the second value is the variable density result.

**Table 3. Summary results for recommended environments categorized by execution time, energy consumption and EDP.**

| Application/ Benchmark | Decision factor | Recommended environment |
|---|---|---|
| 3D Wave Prop. | Time | SGI + CUDA |
| | Energy | Jetson (6 c) + CUDA |
| | EDP | Jetson (6 c) + CUDA) |
| FWI | Time | SGI (12 t) |
| | Energy | Jetson (4 c) |
| | EDP | SGI (12 t) |
| NAS SP | Time | SGI + CUDA |
| | Energy | Jetson (1 c) + CUDA |
| | EDP | Jetson (1 c) + CUDA) |
| NAS BT | Time | SGI (10 t) |
| | Energy | Jetson (6 c) |
| | EDP | Jetson (6 c) |
| NAS LU | Time | SGI + CUDA |
| | Energy | Jetson (1 c) + CUDA |
| | EDP | Jetson (1 c) + CUDA |

execution, allowed the study of how such applications performed. The research focused on ascertaining raw performance as much as the energy consumption.

The results show that there is a great disparity on the executed applications. This can be attributed to the scale of the application, the applications' Motif, the used environment and also the goal of the experiment (execution time, energy efficiency or both using metrics such as EDP). Such conclusion can clearly be observed on Table 2. As such, to develop or adapt an application for an powerful level of execution, like the future exaescale systems, all of these topics need to be considered and addressed. The range of environments used on the study, coupled with the disparity in the results observed in table 2, suggest that there is not a single solution for all problems and the Motif, programming model and architecture balance must be taken into account to achieve an optimal result.

This study uses three main metrics to analyze application execution: execution time, EC and EDP. Depending on the goal of the execution, a specific programming model or environment achieves optimal results. The execution time, was shorter on the Xeon or the Xeon coupled with GPU, while energy consumption was lower on the Jetson or the Jetson coupled with GPU. However, the EDP metrics results vary. The FWI result has the best solution on the Xeon cores, the NAS SP and the BT and the 3D Wave have optimal results on the Jetson with ARM cores and GPU and the NAS BT has the best result on the 6 ARM cores. Thus, the trade-off between execution time and EC must be balanced.

The comparison of the real application 3D Wave along with the benchmark of the same Motif showed a good co-relation among the results. This indicates that, based on the Motif of the application, the suggestion of the best architecture for placing it based on the benchmark result was correct, at least for this real world application. Further studies with a larger range of application and motifs need to be executed for confirming this result.

This study shows that a deep understanding of the application, its Motif, the size, the underlying architecture and the programming model are crucial to both the performance and energy efficiency. However, this knowledge must be balanced and unified

when striving to achieve optimal results.

In future work we intend to experiment with the scalability (OpenMP + MPI) of the applications on different architectures, analyzing an increased number of benchmarks and real applications. Also, given an application that does not use the environment up to its full potential, future experiments intend to explore the possibility of the application co-allocation to maximize usage. The goal is to analyze the impact of concurrent resources on the performance of these applications and the energy consumption. Due to the relative good energy efficiency of the Jetson TX2 observed, specially when coupled with CUDA execution, a future study proposes to explore its use as an HPC alternative.

## Acknowledgments

This work has received partial funding from the European Union's Horizon 2020 Program and from Brazilian MCTIC through the RNP under the HPC4E Project, grant agreement n 689772 and from the CAPES and the CNPq.

## References

Adhinarayanan, V., Subramaniam, B., and chun Feng, W. (2016). Online power estimation of graphics processing units. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colômbia, May 16-19, 2016*, pages 245–254. IEEE Computer Society.

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. (2009). A View of the Parallel Computing Landscape. *Commun. ACM*, 52(10):56–67.

Ashraf, M. U., Eassa, F. A., Albeshri, A. A., and Algarni, A. (2018). Toward exascale computing systems: An energy efficient massive parallel computational model. *International Journal of Advanced Computer Science and Applications*, 9(2).

Bez, J. L., Bernart, E. E., Santos, F. F., Schnorr, L. M., and Navaux, P. O. A. (2016). Performance and energy efficiency analysis of hpc physics simulation applications in a cluster of arm processors. *Concurrency and Computation: Practice and Experience*.

Castro, M., Francesquini, E., Dupros, F., Aochi, H., Navaux, P. O., and Méhaut, J.-F. (2016). Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Computing*, 54(Supplement C):108 – 120. 26th International Symposium on Computer Architecture and High Performance Computing.

Coplin, J. and Burtscher, M. (2016). Energy, power, and performance characterization of GPGPU benchmark programs. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*, pages 1190–1199. IEEE Computer Society.

Ferro, M., Mc Evoy, G., and Schulze, B. (2017). *Analysis of High Performance Applications Using Workload Requirements*, pages 7–10. Springer International Publishing, Cham.

Filiposka, S., Mishev, A., and Juiz, C. (2016). Current prospects towards energy-efficient top hpc systems. *Computer Science and Information Systems*, 13(1):151–171.

Francesquini, E., Castro, M., Penna, P. H., Dupros, F., Freitas, H. C., Navaux, P. O., and Méhaut, J.-F. (2015). On the energy efficiency and performance of irregular application executions on multicore, numa and manycore platforms. *Journal of Parallel and Distributed Computing*, 76(Supplement C):32 – 48. Special Issue on Architecture and Algorithms for Irregular Applications.

Göddeke, D., Komatitsch, D., Geveler, M., Ribbrock, D., Rajovic, N., Puzovic, N., and Ramirez, A. (2013). Energy efficiency vs. performance of the numerical solution of pdes: An application study on a low-power arm-based cluster. *Journal of Computational Physics*, 237(Supplement C):132 – 150.

Loghin, D., Ramapantulu, L., and Teo, Y. M. (2017). On understanding time, energy and cost performance of wimpy heterogeneous systems for edge computing. In *IEEE International Conference on Edge Computing, EDGE 2017, Honolulu, HI, USA, June 25-30, 2017*, pages 1–8. IEEE.

Menezes, G. S., Silva-Filho, A. G., Souza, V. L., Medeiros, V. W. C., Lima, M. E., Gandra, R., and Braganca, R. (2012). Energy estimation tool fpga-based approach for petroleum industry. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ICPPW '12, pages 600–601, Washington, DC, USA. IEEE Computer Society.

Messina, P. (2017). The exascale computing project. *Computing in Science Engineering*, 19(3):63–67.

Mury, A. R., Schulze, B., Licht, F., de Bona, L. C., and Ferro, M. (2015). A concurrency mitigation proposal for sharing environments: An affinity approach based on applications classes. In Al-Saidi, A., Fleischer, R., Maamar, Z., and Rana, O. F., editors, *Intelligent Cloud Computing*, volume 8993 of *Lecture Notes in Computer Science*, pages 26–45. Springer International Publishing.

O'Brien, K., Tucci, L. D., Durelli, G., and Blott, M. (2017). Towards exascale computing with heterogeneous architectures. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 398–403.

Okina, K., Soejima, R., Fukumoto, K., Shibata, Y., and Oguri, K. (2016). Power performance profiling of 3-d stencil computation on an fpga accelerator for efficient pipeline optimization. *SIGARCH Comput. Archit. News*, 43(4):9–14.

Puzović, M., Manne, S., GalOn, S., and Ono, M. (2016). Quantifying energy use in dense shared memory hpc node. In *Proceedings of the 4th International Workshop on Energy Efficient Supercomputing*, E2SC '16, pages 16–23, Piscataway, NJ, USA. IEEE Press.

Weloli, J. W., Bilavarn, S., Derradji, S., Belleudy, C., and Lesmanne, S. (2016). Efficiency modeling and analysis of 64-bit arm clusters for hpc. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 342–347.

# Experimentação e Análise de *Checkpoint* Dinâmico no Apache Hadoop sob Cenários de Falha

**Paulo V. M. Cardoso, Patrícia Pitthan Barcelos**

[1]Pós-Graduação em Ciência da Computação (PGCC)
Universidade Federal de Santa Maria (UFSM)
Santa Maria – RS – Brasil

pcardoso@inf.ufsm.br, pitthan@inf.ufsm.br

***Abstract.*** *The growth of reliability problems on high performance systems has motivated searches for fault tolerance mechanisms. The Apache Hadoop framework, created to store and process large amounts of data, implements Checkpoint and Recovery to help on recovery process of its distributed file system (Hadoop Distributed File System - HDFS) in presence of failure. However, once configuration attributes can not be changed at runtime, bad choices may cause performance and reliability problems. This work uses a dynamic configuration mechanism for checkpoint on Hadoop and evaluates its performance on scenarios with induced fault on the master element of HDFS.*

***Resumo.*** *Os crescentes problemas de confiabilidade em sistemas de alto desempenho motivam a busca por mecanismos de tolerância a falhas. O framework Apache Hadoop, projetado para processar quantidades massivas de dados, usa o mecanismo de Checkpoint and Recovery para auxiliar a recuperação de seu sistema de arquivos distribuído (Hadoop Distributed File System - HDFS) em caso de falha. Porém, como atributos de configuração não podem ser modificados em tempo de execução, escolhas inapropriadas podem gerar problemas de desempenho e confiabilidade. Este trabalho utiliza um mecanismo de configuração dinâmica de checkpoint no Hadoop e avalia seu desempenho frente a cenários de falha induzida no elemento mestre do HDFS.*

## 1. Introdução

A crescente produção de dados digitais tornou a demanda por sistemas computacionais de alto desempenho cada vez mais frequente. Sistemas de alto desempenho são requisitos para o processamento de grandes quantidades de dados com eficiência. Porém, à medida em que mais componentes são usados de forma conjunta, o tempo médio entre falhas torna-se menor [Egwutuoha et al. 2013].

A aplicação de técnicas de tolerância a falhas é essencial para evitar erros computacionais decorrentes de falhas. Uma técnica de tolerância a falhas bastante utilizada é a recuperação de erros, a qual pode ocorrer por avanço ou por retorno [Laprie 1985]. *Checkpoint and Recovery* (CR) corresponde a uma técnica de recuperação de erros por retorno, cujo objetivo é conduzir o sistema a um estado anterior ao problema. O CR consiste em duas fases: o estabelecimento de *checkpoints* e a recuperação, que acontece após a falha e consiste em recuperar o andamento normal do sistema a partir do seu estado estável mais recente.

Uma implementação da técnica de CR é encontrada no Apache Hadoop: um *framework* de alto desempenho desenvolvido para o processamento e o armazenamento de grandes quantidades de dados. No Hadoop, o *checkpoint* é usado para tolerar falhas em seu sistema de arquivos distribuído (*Hadoop Distributed File System* - HDFS). Porém, a configuração de atributos do mecanismo é limitada pelo *framework*: é necessário interromper o seu funcionamento para realizar uma modificação nesse atributo. Sendo assim, adaptações em tempo de execução não são permitidas.

Visto que diferentes aplicações possuem demandas exclusivas, o procedimento de *checkpoint* pode se comportar de maneira distinta em cada situação. Por isso, a escolha do intervalo ideal entre *checkpoints* – que ofereça uma alta confiabilidade mas não interfira no desempenho das aplicações – consiste em um grande desafio. Intervalos estáticos, que são definidos de forma prévia e não mudam no decorrer da execução, tornam-se prejudiciais pois não oferecem suporte às mudanças de comportamento do ambiente.

A partir da observação do comportamento estático de configuração do *checkpoint* no Hadoop, o mecanismo de *checkpoint* dinâmico – definido e validado em trabalhos anteriores [Cardoso and Barcelos 2018] – surge como uma alternativa. A configuração dinâmica permite que o período entre *checkpoints* seja adaptado em tempo real. Além disso, o uso de monitores auxilia na tomada de decisão por períodos adequados, evitando que os *checkpoints* tornem-se procedimentos intrusivos.

A fim de complementar a validação do mecanismo de *checkpoint* dinâmico, este trabalho define cenários de falha transiente no $NameNode$ (NN): o elemento mestre do HDFS. A recuperação do $NameNode$ adiciona uma série de operações para que o serviço volte ao seu funcionamento. Nesse sentido, o uso consciente de *checkpoints* é essencial para que a recuperação apresente um desempenho eficiente. Para realizar uma validação sob falha no $NameNode$, testes de desempenho do *benchmark* $TestDFSIO$ foram feitos sob um cenário sem falha, além de dois cenários com falha induzida: durante a execução de aplicações e outro entre duas aplicações.

O artigo está estruturado da seguinte forma: a Seção 2 apresenta o *framework* Apache Hadoop e sua arquitetura, além de seus mecanismos de tolerância a falhas. A Seção 3 descreve a arquitetura de *checkpoint* dinâmico. A Seção 4 descreve a construção dos cenários de falha. Na Seção 5, apontam-se os experimentos e a discussão acerca dos resultados. A Seção 6 aponta conclusões e próximos passos.

## 2. Apache Hadoop

O Apache Hadoop é um projeto *open source* voltado para o processamento distribuído de grandes quantidades de dados [White 2015]. O Hadoop oferece um eficiente sistema de distribuição de dados e aplicações em arquiteturas de alto desempenho, como *clusters* e *grids*. O conceito essencial do Hadoop é mover a aplicação até os dados – evitando mover os dados em si – para obter eficiência de processamento [Jain and Goyal 2017]. A arquitetura do Hadoop, em sua versão utilizada por este trabalho (v2.7.3), é composta por diversos módulos, dentre os quais destacam-se: o sistema de arquivos distribuído (HDFS) e o *framework* para manutenção do ambiente e da aplicação (YARN).

## 2.1. HDFS

Os dados no Apache Hadoop são armazenados em um sistema de arquivos distribuído denominado *Hadoop Distributed File System* (HDFS), que foi projetado para oferecer suporte a arquivos de grandes dimensões. Um arquivo no HDFS é dividido em blocos de tamanho pré-definido e distribuídos através do ambiente. A distribuição faz com que blocos de um mesmo arquivo possam estar armazenados em diferentes nós, para que o propósito de mover a computação até os nós, implementado pelo Hadoop, seja facilitado.

No HDFS, dados e metadados são armazenados de forma separada. Os metadados são mantidos por um servidor dedicado, chamado $NameNode$ (NN), enquanto um $DataNode$ (DN) é responsável por realizar o armazenamento dos dados. A arquitetura do HDFS é baseada no modelo *1-master N-workers*, em que um único NN serve $N$ $DataNodes$, como mostra a Figura 1. O $NameNode$ também atende às requisições de aplicações e gerencia os arquivos e $DataNodes$ do HDFS.

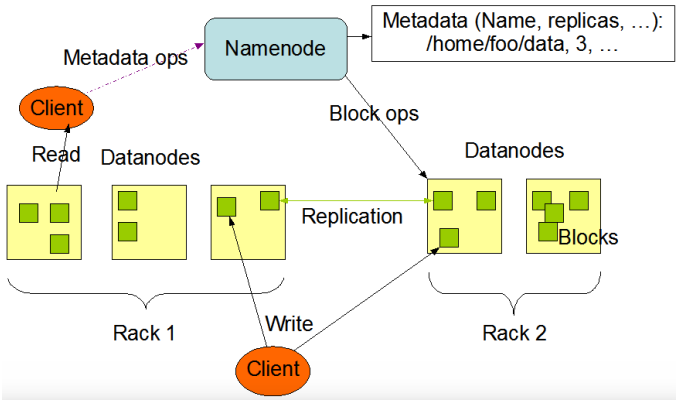



**Figura 1. Arquitetura utilizada pelo HDFS [Foundation 2017]**

A representação de arquivos e diretórios no $NameNode$ é feita por objetos do tipo `INode`, que são armazenados em sua memória local, tornando disponível um mapeamento de arquivos/blocos no HDFS. Para completar o *namespace*, há um outro mapeamento que possui informações sobre a localização dos blocos de cada DN. Em um intervalo de tempo pré-definido (por padrão 4 segundos), os $DataNodes$ realizam uma varredura em seus blocos e reportam o estado para o $NameNode$. Assim, o nó mestre do HDFS possui uma visão sobre a localização de cada bloco do HDFS.

## 2.2. Tolerância a falhas

Para garantir confiabilidade e disponibilidade, o Apache Hadoop possui implementações de técnicas de tolerância a falhas em seus processos. As técnicas devem garantir que os serviços oferecidos pelo *framework* sejam mantidos, mesmo quando um evento de falha acontece. Dentre as técnicas de tolerância a falhas implementadas pelo Hadoop, tem-se: a replicação de blocos do HDFS, o envio de mensagens *heartbeat* e o estabelecimento de *checkpoints*.

### 2.2.1. Replicação

O HDFS trabalha com a técnica de replicação para assegurar a disponibilidade de todos os seus blocos. O número de réplicas, controlado pelo fator de replicação, define a quantidade de cópias que um mesmo bloco possui no sistema. Dessa forma, dados de $DataNodes$ falhos podem ser recuperados a partir de réplicas armazenadas em outros nós, evitando que as aplicações deixem de funcionar corretamente em caso de falha.

O fator de replicação e o tamanho de um bloco são previamente definidos (respectivamente, 3 réplicas e 128MB), mas podem ser configurados para cada arquivo individualmente. A localização das réplicas é fundamental para a confiabilidade e para o desempenho da replicação. Por padrão, as duas primeiras réplicas são armazenadas em nós que estejam próximos entre si. Assim, o HDFS visa um acesso mais rápido em caso de recuperação.

### 2.2.2. Heartbeat

O envio de mensagens *heartbeat* consiste em um mecanismo de tolerância a falhas usado para a detecção de falhas em $DataNodes$ [White 2015]. Periodicamente, os DNs ativos enviam avisos ao NN a fim de reportar o estado de seus blocos e sinalizar sua plena atividade. Quando um DN deixa de funcionar, por falhas ou problemas de comunicação, o NN o define como um elemento falho e deixa de enviar instruções ao mesmo.

O intervalo padrão para o envio de mensagens *heartbeat* é de 4 segundos. A detecção de um $DataNode$ falho acontece quando um *timeout* de 10 minutos é atingido sem que o $DataNode$ envie um *heartbeat*. Um DN detectado como falho é considerado fora de serviço e seus blocos são marcados como indisponíveis. Por isso, o $NameNode$ deve realizar um procedimento de recuperação do fator de replicação a partir da criação das réplicas perdidas.

### 2.2.3. Checkpoint

*Checkpoint and Recovery* (CR) é uma técnica de tolerância a falhas reativa, classificada como técnica de recuperação por retorno (*backward error recovery*). O CR tem como ideia principal a recuperação do estado falho de um sistema através de um contexto estável previamente salvo [Egwutuoha et al. 2013]. Essa recuperação visa prevenir erros computacionais consequentes das falhas. Os contextos estáveis são salvos periodicamente e armazenados de forma segura para uma posterior recuperação do sistema.

Os casos em que a técnica de *checkpoint* se mostra vantajosa estão relacionados a aplicações de longa duração (*long-running applications*) [Cui et al. 2015], frequentemente executadas pelo Hadoop. O tempo de execução dessas aplicações pode girar em torno de horas ou até dias. Outro cenário favorável ao uso de CR é o processamento intensivo de dados, devido ao grande número de operações de *I/O*. Nesses casos, uma falha ao final da execução pode acarretar em uma perda total ou reexecução completa.

O *checkpoint* no Hadoop é implementado para tolerar falhas no $NameNode$. Apesar de não ser o principal mecanismo de tolerância a falhas do Hadoop, o salvamento de

*checkpoints* é essencial para a manutenção do $NameNode$ e de seus metadados. Assim, o *namespace* do HDFS é replicado em um arquivo chamado `FSImage`, armazenado em disco no sistema de arquivos local do NN. Nesse arquivo, são mantidas informações sobre o mapeamento de blocos e propriedades do sistema.

Para evitar que um novo `FSImage` seja criado a cada operação feita, um *log* de edições (`EditLog`), também mantido em disco local, armazena as últimas transações realizadas após a criação do `FSImage`. O merge entre o `FSImage` e o `EditLog` consiste no estabelecimento de *checkpoint*. Esse procedimento é feito de forma periódica (intervalo *default* é de 3600 segundos), mas é disparado se o HDFS atingir um número específico de transações (10 mil, por padrão).

Para que o $NameNode$ não seja interrompido durante um *checkpoint*, o Hadoop executa um elemento chamado $SecondaryNameNode$ (SNN). A função do SNN é realizar o procedimento de *checkpoint* assim que o NN requisita. A cada salvamento de *checkpoint*, o NN transfere o `EditLog` para o SNN, que mantém uma cópia do `FSImage`. Assim, o merge entre os arquivos é feito para que o novo `FSImage` seja criado. Uma cópia do novo arquivo é enviada de volta ao NN e outra é armazenada no SNN.

## 3. Checkpoint Dinâmico

O *checkpoint* apresenta-se como um importante mecanismo no contexto da tolerância a falhas. Porém, configurá-lo de forma eficiente é um grande desafio, uma vez que diversos fatores podem interferir no seu comportamento. Sob escolhas inapropriadas, o sistema e as aplicações sofrem com problemas de confiabilidade e desempenho. Nesse caso, a propriedade de tolerância a falhas do *checkpoint* perde o seu propósito.

Desta forma, o objetivo da arquitetura de *checkpoint* dinâmico é tornar mais eficiente a configuração de atributos relacionados ao mecanismo. Com configurações dinâmicas, novos atributos são calculados em tempo real, com um foco voltado a propriedades adequadas para determinados momentos. Para isso, dois elementos essenciais são usados: o monitor e o coordenador, exibidos na Figura 2. Além disso, como demonstrado em trabalhos anteriores [Cardoso and Barcelos 2018], uma implementação para o HDFS foi definida para que adaptações do período entre *checkpoints* seja possível.

### 3.1. Monitoramento

O monitoramento de recursos é uma etapa essencial da definição dinâmica de *checkpoints*. A partir de análises sobre a situação do sistema, é possível realizar uma quantificação da utilização de recursos. Assim, a arquitetura dinâmica tem a possibilidade de identificar a necessidade de adaptação do período entre *checkpoints*. Assim como a maioria das ferramentas de monitoramento de recursos [Aceto et al. 2013], a arquitetura do módulo monitor deste trabalho usa uma implementação agente-servidor.

Os agentes – executados individualmente em nós do ambiente – coletam informações sobre recursos e notificam o servidor quando há uma alteração de comportamento significativa. Os fatores de utilização passíveis de observação são: (a) CPU, (b) memória RAM e (c) operações em disco. Por sua vez, o servidor (supervisor) mantém uma comunicação passiva com os agentes. Quando uma mensagem é recebida, o supervisor armazena as informações e faz uma análise dos dados de agentes.

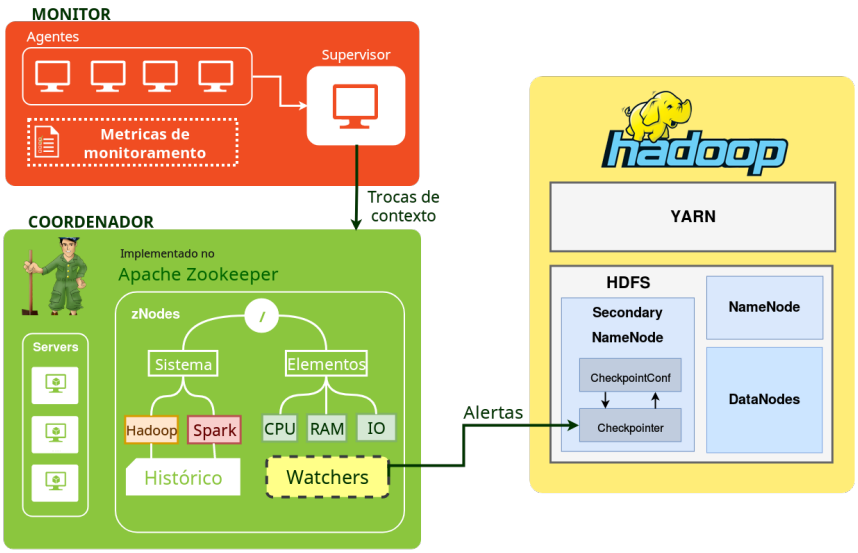



**Figura 2. Arquitetura de configuração dinâmica para o *checkpoint* no HDFS.**

Um cálculo que indica novos atributos de *checkpoint* a serem usados é feito pelo supervisor. Imediatamente, o novo valor é atualizado no coordenador do mecanismo. A relevância da alteração de comportamento nos monitores, bem como a política adotada para calcular um novo período de *checkpoint*, são definidas de acordo com métricas de monitoramento. Para isso, usam-se arquivos de configuração com os fatores de utilização observados, além da política de valores que definem mudanças de contexto no sistema.

### 3.2. Gerenciamento

Visando um armazenamento seguro de informações de configuração, o coordenador foi usado como um repositório central de atributos de configuração, a partir do *framework* Apache Zookeeper: um projeto *open source* com funcionalidades para facilitar a coordenação de sistemas distribuídos [Hunt et al. 2010]. A arquitetura do Zookeeper é formada por servidores que realizam operações requisitadas por clientes em um $namespace$ compartilhado, cuja estrutura é composta por uma árvore de $zNodes$.

Nesse *framework* foram criados diversos atributos de configuração em $zNodes$. O atributo atual é definido em um $zNode$ específico, sendo que os atributos já usados são mantidos em um histórico. Uma importante função do Coordenador é a configuração de mecanismos de alerta (*watchers*), que geram notificações para cada modificação em um $zNode$. O nó responsável por armazenar o atributo de configuração atual possui um *watcher* que direciona a uma mudança de configuração para qualquer sistema que esteja observando-o.

### 3.3. Implementação no HDFS

O processo de *checkpoint* pode se tornar um fator crítico para o desempenho do Apache Hadoop. A escolha por períodos frequentes ou mais longos implica em diferentes comportamentos de recuperação e, inclusive, pode prejudicar o andamento de aplicações. Por isso, a implementação do mecanismo dinâmico no HDFS tem como objetivo a adaptação em tempo real do período de *checkpoints*. As alterações são feitas sem a necessidade de interromper o Hadoop e seus serviços.

Alterações no comportamento do processo de *checkpoint* foram realizadas no $SecondaryNameNode$. O SNN implementa duas classes essenciais para *checkpoint*: o `CheckpointConf` e o `Checkpointer`. O `CheckpointConf` fornece um meio de acesso em memória aos dados de configuração, enquanto que o `Checkpointer` é responsável por manter e liderar o procedimento de *checkpoint*.

No mecanismo dinâmico foram realizadas alterações no tratamento do período entre *checkpoints* na classe `Checkpointer`. Essas alterações incluem um elemento de comunicação entre o HDFS e o módulo Coordenador e um tratamento de alertas, para alterações nos atributos de configuração em uma troca de contexto. Caso o período seja modificado, o `Checkpointer` deve verificar se o novo intervalo já foi atingido na espera da antiga configuração, invocando um novo salvamento de *checkpoint* caso este fato ocorra, ou esperando o tempo restante.

A modificação no `Checkpointer` é realizada a partir das funções de iniciação da classe e do procedimento de *checkpoint*. Ao iniciar, o `Checkpointer` verifica a metodologia de *checkpoint* a ser utilizada (dinâmica ou estática), conectando-se com o coordenador caso necessário. Logo, a classe de configurações é criada e o salvamento de *checkpoint* é iniciado. Caso uma mudança de atributo aconteça durante uma espera, o processo reinicia imediatamente para calcular o período restante. O *checkpoint* ocorre quando nenhuma espera é necessária.

## 4. Cenários de Falha

O $NameNode$ é o elemento central da arquitetura do HDFS. Por isso, um evento de falha neste elemento pode comprometer a disponibilidade dos dados e prejudicar o andamento das aplicações. Nas versões iniciais do Hadoop, o NN era considerado um ponto único de falha (SPOF - *single point of failure*). Já nas versões 2.x, o atributo de *High Avaliability* auxilia a recuperação do NN a partir de *backups* em tempo real, mas acrescenta operações extras ao contexto do Hadoop. Porém, falhas transientes no $NameNode$ podem ser reparadas apenas com seu reinício, evitando-se operações adicionais.

Reiniciar o $NameNode$ durante o andamento de uma aplicação pode ser fatal, dado que uma interação mal sucedida entre a aplicação e o HDFS gera erros de execução. Nas etapas de início da execução, realizam-se a alocação de recursos e a preparação da plataforma. Neste momento, ainda que a aplicação não esteja em execução, o HDFS é usado para armazenar informações sobre a aplicação. Essas informações incluem o arquivo executável (JAR), configurações e informações sobre arquivos de entrada e saída [White 2015]. A quantidade de tarefas criadas para a execução é baseada no número de divisões do arquivo de entrada (*input splits*). Para consultar essa informação, um novo acesso é feito ao HDFS. Logo após, a execução inicia e as interações com o HDFS acontecem de acordo com a demanda da aplicação.

Para criar os cenários de falha transiente com recuperação imediata, mecanismos de tolerância a exceções foram definidos em cada iteração com o $NameNode$. Para isso, métodos da classe de tradução do protocolo usado por clientes para comunicação com o $NameNode$[1] foram alterados. Os métodos: $addBlock$, $complete$, $create$, $delete$ e $getFileInfo$ foram modificados, uma vez que todos são responsáveis pela execução de comandos no HDFS.

[1]Classe `ClientNamenodeProtocolTranslatorPB`

As alterações realizadas nos métodos descritos incluem um número máximo de tentativas de comunicação a serem feitas, antes de detectar o $NameNode$ como falho, além de um tempo de espera entre as tentativas. Essa espera é fundamental para que a aplicação não seja encerrada antes de uma possível recuperação, quando há falha. Por outro lado, as alterações não comprometem a execução das aplicações quando nenhuma falha é detectada.

## 5. Experimentação

As experimentações foram realizadas na plataforma Grid'5000 [2], um ambiente distribuído de larga escala que fornece uma infraestrutura para testes relacionados a aplicações distribuídas e paralelas [Balouek and et al. 2013]. A configuração usada no *grid* era composta por 8 nós e a configuração de cada nó foi definida com dois processadores Intel Xeon E5-2630v3 (oito *cores* por CPU) e 4GB de memória RAM. O espaço total em disco disponível para o HDFS era de 1,5 TB.

Os testes feitos avaliaram o tempo de execução do *benchmark* $TestDFSIO$, que produz operações de leitura e escrita no HDFS. A escolha deste *benchmark* se deve ao intenso uso de disco, podendo-se fazer avaliações em situações de sobrecarga de *I/O*. O $TestDFSIO$ foi executado na operação de escrita, configurado para criar 8 arquivos de 32GB (256GB, no total). O tempo de execução medido foi definido como a média de 20 rodadas, em cada cenário de teste. Também foi definida uma medida de sobrecarga (*overhead*), que refere-se ao acréscimo de tempo observado por determinado caso, com base em outro. As métricas de monitoramento foram especificamente escolhidas para o *benchmark* usado. Novas métricas devem ser exploradas em trabalhos futuros.

A avaliação de desempenho dos mecanismos de *checkpoint* foi organizada em três cenários: sem falha, com falha de parada e com falha entre aplicações. O primeiro cenário consistiu em um contexto sem falhas em que apenas o tempo de execução é medido. Os outros cenários incluem a indução de uma falha transiente no $NameNode$ com recuperação imediata. Nos cenários de falha, utilizou-se o comando $kill$, do Linux, para encerrar o processo do $NameNode$.

Na falha de parada, o $NameNode$ é encerrado durante a execução do *benchmark*. Quando a falha é detectada, a aplicação espera pelo retorno do NN e volta a funcionar. A eficiência da recuperação é dada pela quantidade de operações necessárias para que o NN volte ao seu funcionamento normal. Já na falha entre aplicações, uma falha foi induzida ao final da execução do $TestDFSIO$. Para avaliar a recuperação, iniciou-se uma nova execução do *benchmark* em modo de leitura após a iniciação do elemento que falhou.

Os resultados obtidos são sumarizados nas Tabelas 1 e 2, em que o tempo de execução em segundos ($TEx$) e o respectivo desvio padrão ($DPd$) de cada cenário de teste são exibidos. A Tabela 1 mostra resultados do mecanismo estático, enquanto que a Tabela 2 exibe o resultados do mecanismo dinâmico. Em ambos os mecanismos de *checkpoint*, as configurações usadas ($CCh$) como 3600, 360, 36 e 10 segundos referem-se ao período entre *checkpoints*, no mecanismo estático, e ao intervalo de monitoramento

[2]Grid'5000 é uma plataforma para experimentos apoiado por um grupo de interesses científicos hospedado por Inria e incluindo CNRS, RENATER e diversas Universidades, bem como outras organizações (mais datalhes em https://www.grid5000.fr).

dos agentes no *checkpoint* dinâmico. A falha foi induzida depois de 460 segundos do início das execuções, representando 25% da execução do *baseline* estático sem falha.

| **Cenário** | $CCh$ | $TEx$ | $DPd$ |
|---|---|---|---|
| Sem falha | 3600s | 1846,8 | 92,7 |
| | 360s | 1998,0 | 124,4 |
| | 36s | 2056,2 | 143,1 |
| | 10s | 2126,8 | 136,0 |
| Parada | 3600s | 2260,4 | 96,5 |
| | 360s | 2089,5 | 71,5 |
| | 36s | 2108,0 | 108,5 |
| | 10s | 2176,5 | 82,3 |
| Execução | 3600s | 2107,4 | 147,2 |
| | 360s | 2018,4 | 65,2 |
| | 36s | 2064,8 | 84,3 |
| | 10s | 2032,5 | 109,2 |

**Tabela 1. Execução do mecanismo estático.**

| **Cenário** | $CCh$ | $TEx$ | $DPd$ |
|---|---|---|---|
| Sem falha | 3600s | 2093,9 | 151,7 |
| | 360s | 2042,3 | 130,8 |
| | 36s | 1987,0 | 71,1 |
| | 10s | 1974,1 | 121,4 |
| Parada | 3600s | 2167,2 | 112,5 |
| | 360s | 2092,5 | 162,3 |
| | 36s | 2016,0 | 161,9 |
| | 10s | 2003,6 | 137,6 |
| Execução | 3600s | 2085,2 | 114,0 |
| | 360s | 1914,5 | 158,5 |
| | 36s | 1950,3 | 159,0 |
| | 10s | 1899,0 | 73,4 |

**Tabela 2. Execução do mecanismo dinâmico.**

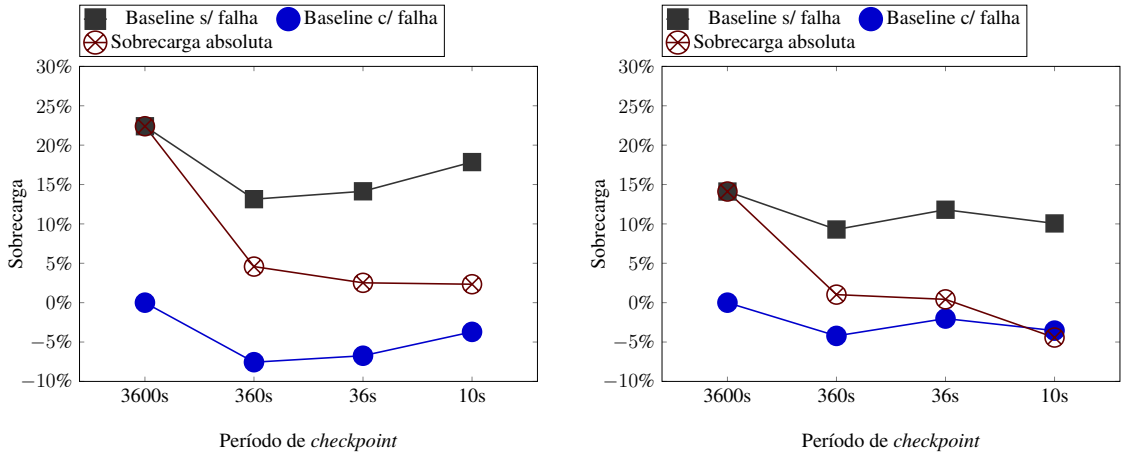
### 5.1. Checkpoint estático

É possível notar pela Tabela 1 que o aumento da frequência de *checkpoints* estáticos é impactante no desempenho da aplicação, pois aumenta o número de operações de *I/O* realizadas pelo HDFS. Dado que as operações de *checkpoint* incluem transferência de dados entre NN e SNN, além do processamento do novo `FSImage`, nota-se que *checkpoints* mais frequentes são mais custosos devido a sobrecarga gerada pela comunicação entre elementos. Já os *checkpoints* menos frequentes, mesmo que necessitem salvar mais transações por *checkpoint*, não geram sobrecarga de comunicação.

Por outro lado, nos cenários de falha observados, a atualização do *checkpoint* é fundamental para uma recuperação mais rápida. No *checkpoint* mais frequente utilizado, o tempo acrescido na execução em decorrência da falha de parada é consideravelmente menor se comparado ao *baseline*. Esse comportamento também é visto no cenário de falha entre execuções: mesmo configurado para operações de leitura, a recuperação do NN de forma concorrente ao *benchmark* é beneficiada por *checkpoints* atualizados.

Para auxiliar na compreensão dos resultados das diferentes configurações no mecanismo estático, as Figuras 3(a) e 3(b) exibem os diferentes níveis de sobrecarga identificados nos cenários de falha de parada e falha entre execuções, respectivamente. O *baseline* é estabelecido pelo caso padrão de *checkpoint* estático (3600 segundos). Além da comparação de cada configuração com o *baseline* – sobrecarga relativa –, o gráfico também mostra comparações de configurações idênticas nos diferentes cenários – sobrecarga absoluta. Sobrecargas negativas indicam ganho de desempenho, o que significa uma diminuição no tempo de execução.

A configuração de *checkpoint* estático em 360 segundos se mostrou mais eficiente em relação às sobrecargas relativas, de modo que o equilíbrio da frequência de *checkpoints* é importante. Porém, a sobrecarga absoluta é menor conforme mais *checkpoints*




(a) Cenário com falha de parada.

(b) Cenário com falha entre execuções.

**Figura 3. Sobrecargas do mecanismo estático nos cenários de falha.**

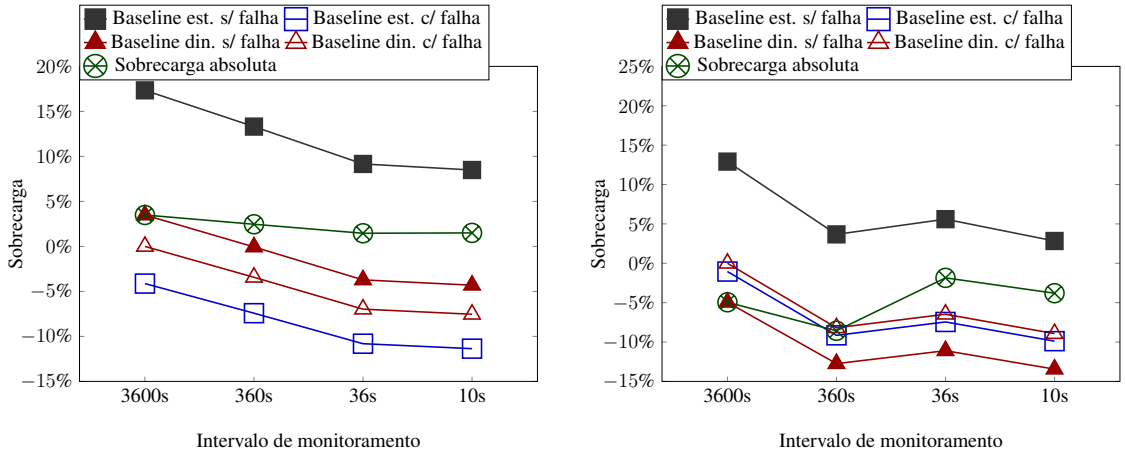
são realizados. Isto é, a configuração de *checkpoints* a cada 10 segundos em cenários de falha teve a menor sobrecarga em relação à mesma configuração quando nenhuma falha foi induzida. Nesse sentido, o fato do *checkpoint* atual (no momento da falha) estar mais atualizado torna a recuperação mais rápida. Essa característica pode ser interessante caso mais de uma falha aconteça. Já no cenário de falha entre execuções, ainda que as sobrecargas representem operações distintas ($write$ e $read$), a base de dados usada é a mesma. Além disso, é possível verificar que o comportamento da variação do período entre *checkpoints* é mantido.

## 5.2. Checkpoint dinâmico

O mecanismo dinâmico apresentou uma melhora no tempo de execução conforme utilizou-se um intervalo de monitoramento menor. Ainda assim, o comportamento é praticamente linear: o tempo acrescido na variação do monitoramento é pequeno, indicando uma baixa intrusão da arquitetura no processamento da aplicação. Além disso, a escolha por *checkpoints* em momentos de baixa carga de processamento se mostrou eficiente. A exceção é o caso 3600 segundos, em que o ganho proporcionado pelo mecanismo é limitado por escolhas que acabam tornando-se defasadas no decorrer das execuções.

A Figuras 4(a) e 4(b) exibem a sobrecarga gerada pelo mecanismo dinâmico. Foram definidas as sobrecargas relativas e absolutas relacionadas à ambos os mecanismos de *checkpoint*, para fins de comparação. Nota-se que as falhas adicionam tempo de execução, mas a recuperação é eficiente já que a sobrecarga absoluta não ultrapassa 4%. Contudo, a eficiência de recuperação não é contida pela sobrecarga de *checkpoints* ao decorrer da aplicação – como ocorreu no caso do *checkpoint* estático de 10 segundos.

Em todas as variações do período de monitoramento, os resultados mostram uma diminuição de tempo nos testes com monitoramento mais frequente, em relação aos *baselines* estático e dinâmico. Essa diminuição deve-se ao fato do período entre *checkpoints* possuir uma frequência adaptada ao contexto da aplicação. Desta forma, nos momentos em que a aplicação demanda pouco uso do sistema, o monitoramento consegue identificar uma situação favorável para o salvamento de *checkpoints*. Em contrapartida, o meca-




(a) Cenário com falha de parada.

(b) Cenário com falha entre execuções.

**Figura 4. Sobrecargas do mecanismo dinâmico nos cenários de falha.**

nismo dinâmico adia o salvamento quando o sistema está sobrecarregado. O *overhead* da realização de um *checkpoint* em momentos de alto uso do sistema é contido nesse caso.

A Figura 5 faz uma comparação da quantidade de *checkpoints* realizados pelos mecanismos dinâmico e estático nos cenários executados. O comportamento de *checkpoints* estáticos é previsível, uma vez que os atributos são fixos. No caso mais frequente, cerca de 200 salvamentos são realizados por execução. No mecanismo dinâmico, porém, o comportamento é diferente: poucos *checkpoints* em momentos estratégicos. Essa característica é mais evidente no cenário de falha entre execuções, em que o monitoramento frequente decide por fazer *checkpoints* perto da finalização da primeira execução. Esse momento pode ser favorável a um *checkpoint* pelo baixo processamento da aplicação.

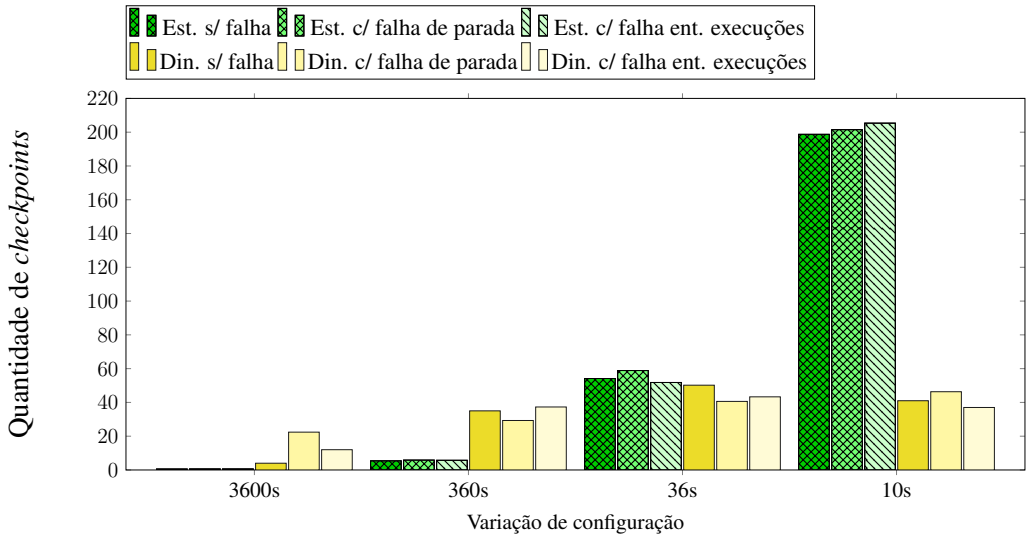



**Figura 5. Número de *checkpoints* realizados pelos mecanismos estático e dinâmico nos diferentes cenários de teste.**

## 6. Considerações finais

Este trabalho validou um mecanismo de configuração dinâmica para o *checkpoint* do HDFS. A ideia do *checkpoint* dinâmico é oferecer suporte a mudanças no intervalo desse recurso em tempo de execução. A partir de testes com indução de falhas no $NameNode$, observou-se uma perspectiva favorável de utilização do mecanismo dinâmico, já que seu comportamento se mostrou mais eficiente em relação ao Hadoop com *checkpoint* estático.

O baixo nível de sobrecarga observado pode proporcionar ganhos para o sistema, moderando os aumentos no tempo de execução verificados em situações da versão estática de *checkpoint*. Por adaptar-se ao uso monitorado do ambiente, o mecanismo dinâmico ofereceu uma solução otimizada para a periodicidade de *checkpoints*. A escolha por momentos apropriados para o salvamento dos *checkpoints* também se mostrou essencial para que a ferramenta apresentasse um desempenho satisfatório em cenários com e sem falha.

Em trabalhos futuros, novos cenários de falha serão criados para uma completa validação do mecanismo de *checkpoint* dinâmico. Entre os cenários, condições de falha de comunicação, variações de *benchmarks* e diferentes configurações para falhas de parada deverão ser testados. Para otimizar a arquitetura, serão desenvolvidas diferentes métricas de monitoramento, com o auxílio do histórico de atributos. Além disso, a arquitetura dinâmica será explorada em outras ferramentas.

## Referências

Aceto, G., Botta, A., De Donato, W., and Pescapè, A. (2013). Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115.

Balouek, D. and et al. (2013). Adding virtualization capabilities to the Grid'5000 testbed. In *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*. Springer Intl Publishing.

Cardoso, P. V. and Barcelos, P. P. (2018). Validation of a dynamic checkpoint mechanism for apache hadoop with failure scenarios. In *Test Symposium (LATS), 2018 IEEE 19th Latin-American*, pages 1–6. IEEE.

Cui, L., Hao, Z., Li, L., Fei, H., Ding, Z., Li, B., and Liu, P. (2015). Lightweight virtual machine checkpoint and rollback for long-running applications. In *Int. Conference on Algorithms and Architectures for Parallel Processing*, pages 577–596. Springer.

Egwutuoha, I. P., Levy, D., Selic, B., and Chen, S. (2013). A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326.

Foundation, A. S. (2017 (acessado em julho de 2017)). *Apache Hadoop 2.7.3*. `https://hadoop.apache.org/docs/r2.7.3/`.

Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, page 9.

Jain, H. and Goyal, A. (2017). An improved approach for analysis of hadoop data for all files. *International Journal of Computer Applications*, 157(4).

Laprie, J.-C. (1985). Dependable computing and fault tolerance: Concepts and terminology. In *25th International Symposium on Fault-Tolerant Computing, 1995*. IEEE.

White, T. (2015). *Hadoop: The Definitive Guide, 4th Edition*. "O'Reilly Media, Inc.".

# Balanceamento de Carga de Aplicações Iterativas em Arquiteturas Heterogêneas

**Luis F. V. Trivelatto**[1], **Edmar A. Bellorini**[1], **Guilherme Galante**[1]

[1]Ciência da Computação
Universidade Estadual do Oeste do Paraná (UNIOESTE)
Caixa Postal 711 – 85.819-110 – Cascavel-PR

{luis.trivelatto, edmar.bellorini, guilherme.galante}@unioeste.br

***Abstract.*** *The objective of this paper is to present a multilevel load balancing library for iterative applications, considering its execution in clusters of heterogeneous nodes, composed of multiple cores and accelerators. The main contribution of this research is the possibility of performing the load balancing between the heterogeneous nodes of the cluster and also in a second level, redistributing the load between the processor cores and accelerators. The library proved to be effective, allowing a higher architecture utilization rate and reducing the execution time of the applications.*

***Resumo.*** *O objetivo deste trabalho é apresentar uma biblioteca de balanceamento de carga multinível para aplicações iterativas, considerando a sua execução em clusters formados por nodos heterogêneos, compostos por múltiplos núcleos de processamento e aceleradores. A principal contribuição desta pesquisa é a possibilidade de realizar o balanceamento de carga entre os nodos heterogêneos do cluster e também em um segundo nível, redistribuindo a carga de modo apropriado entre os núcleos do processador e aceleradores. A biblioteca mostrou-se efetiva, permitindo a elevação da taxa de utilização das arquiteturas e reduzindo o tempo de execução das aplicações em todos os cenários de teste.*

## 1. Introdução

A computação paralela em ambientes heterogêneos tem recebido uma atenção cada vez maior devido ao número crescente destes tipos de sistemas [Cabrera et al. 2018]. Essa importância pode ser vista, por exemplo, no fato de que uma grande fração dos supercomputadores do TOP500 e Green500 agora combinam CPUs e algum tipo de aceleradores [Mittal e Vetter 2015]. Clusters compostos por nós heterogêneos estão sendo cada vez mais usados para computação de alto desempenho devido aos benefícios em termos de desempenho e eficiência energética [Da Costa et al. 2015].

Infelizmente, desenvolver uma aplicação que possa utilizar todos os recursos de processamento disponíveis de forma eficaz, e fazê-lo de forma consistente, não é trivial [Boyer et al. 2013]. Em particular, a heterogeneidade tem um impacto significativo na distribuição da carga de trabalho. Ignorar que diferentes componentes da arquitetura podem ter diferentes capacidades computacionais geralmente resulta em sistemas mal balanceados, o que dificulta a utilização efetiva de todos os recursos da máquina [Bosque et al. 2013]. Por isso, aplicações executadas em sistemas heterogêneos

geralmente têm como alvo apenas o dispositivo mais potente, deixando outros dispositivos ociosos e potencialmente desperdiçando parte do poder computacional disponível. Nesse sentido, o balanceamento de carga é uma tarefa crítica para melhorar o uso dos recursos e para a obtenção de alto desempenho em arquiteturas heterogêneas.

Uma classe de aplicações bastante sensível ao desbalanceamento de carga é a das aplicações iterativas [Menon e Kalé 2013]. Neste tipo de aplicação, é possível extrair o paralelismo particionando-se o domínio a ser processado em diversos subdomínios, os quais são processados paralelamente e após isso, os dados são sincronizados entre as linhas de execução paralelas para permitir o processamento da próxima iteração. Este modelo iterativo aparece em diversos algoritmos paralelos, tais como método de Jacobi, programação dinâmica, problemas de caminho mínimo e problemas de decomposição de domínio.

O alto desempenho de rotinas iterativas em plataformas heterogêneas pode ser alcançado quando todos os processadores concluem seu trabalho ao mesmo tempo. Isso é obtido particionando a carga de trabalho computacional de maneira equivalente em todos os processadores, levando em consideração suas particularidades. Caso o balanceamento de carga não seja realizado, o desempenho será limitado pelo processador mais lento, uma vez que este leva mais tempo para alcançar o ponto de sincronismo ao final da iteração.

Assim, o objetivo deste trabalho é apresentar uma solução para balanceamento de carga para aplicações iterativas, considerando a sua execução em clusters formados por nodos heterogêneos, compostos por múltiplos núcleos de processamento e aceleradores. A principal contribuição é permitir o balanceamento de carga para ambientes heterogêneos, não visando apenas o uso dispositivo com maior capacidade (geralmente a GPU), mas permitindo o uso de toda a capacidade de computação da arquitetura. Isso é realizado por um balanceador multinível, o qual realiza o balanceamento de carga entre os nodos do cluster e também em um segundo nível, redistribuindo a carga de modo apropriado entre os núcleos do processador (em nível de threads) e aceleradores. A solução foi implementada na forma de uma biblioteca, chamada MultiBalance, de modo a permitir a adaptação de aplicações legadas com poucas linhas de código. A solução proposta mostrou-se efetiva, permitindo a elevação da taxa de utilização das arquiteturas e reduzindo o tempo de execução das aplicações em todos os cenários de teste.

O restante do trabalho é organizado da seguinte forma. A Seção 2 apresenta os trabalhos relacionados. A Seção 3 apresenta a solução de balanceamento de carga multinível proposta neste trabalho. Na Seção 4 realiza-se a avaliação experimental. Por fim, a Seção 5 conclui este trabalho.

## 2. Trabalhos Relacionados

Balanceamento de carga em arquiteturas heterogêneas é uma tarefa crítica para melhoria do uso dos recursos e para a obtenção de alto desempenho. Considerando a importância do tópico, alguns trabalhos acadêmicos têm desenvolvido soluções para balanceamento para esse tipo de arquiteturas. Nesta seção, são descritas as soluções encontradas na literatura que mais se aproximam da presente proposta.

[Lu et al. 2012] apresentam um modelo de programação para otimizar a exploração dos recursos computacionais em clusters em que cada nodo apresenta

múltiplos núcleos e um acelerador. Neste modelo, a comunicação entre os nodos ocorre utilizando MPI, porém apenas com um processo em cada nodo. Este processo ainda controla a execução dos kernels na GPU e cria threads via OpenMP para explorar os demais núcleos do nodo. Os autores apresentam os cálculos para o balanceamento estático ótimo entre vários núcleos e uma GPU, considerando o speedup obtido pela GPU em relação a uma CPU. Esse balanceamento é válido apenas dentro de um nodo do cluster, e não há uma proposta de balanceamento considerando toda a arquitetura a ser explorada.

[Bosque et al. 2013] propõem um algoritmo de balanceamento dinâmico para clusters heterogêneos. No trabalho, a heterogeneidade se dá pelas diferentes capacidades computacionais dos nodos do cluster. O algoritmo é executado de modo distribuído e baseado em tarefas, de forma que cada nodo decide se há a necessidade de realizar um balanceamento de carga a partir de um índice de carga (*load index*), calculado considerando fatores estáticos e dinâmicos (o número de núcleos, o número de tarefas, etc.). Se o índice de um nodo está baixo, significa que ele está sobrecarregado, e deve procurar um nodo com índice alto para transferir tarefas de sua fila local.

[Acosta et al. 2013] apresentam um algoritmo de balanceamento dinâmico baseado em processamento iterativo. O algoritmo busca balancear a carga de trabalho tentando igualar o tempo de execução de cada unidade de processamento em uma iteração. Inicialmente, a carga é dividida uniformemente entre as unidades de processamento. Ao fim de cada iteração, uma comunicação coletiva é realizada para compartilhar o tempo de execução de cada unidade de processamento naquela iteração. O algoritmo é implementado na biblioteca `ULL_MPI_calibrate`. [Cabrera et al. 2018] estende essa proposta adicionando a possibilidade de gerenciar o consumo energético.

Uma limitação dessas soluções está no fato de realizarem o balanceamento apenas no nível dos processos MPI, não havendo a possibilidade de um segundo nível de balanceamento, entre os núcleos e aceleradores. Dessa forma, o presente trabalho pretende estender a solução proposta por [Acosta et al. 2013] de modo a permitir o balanceamento de carga multinível, conforme apresentado na seção que segue.

## 3. Balanceamento Multinível

A biblioteca MultiBalance permite balancear a carga de trabalho entre as unidades de processamento em aplicações paralelas iterativas utilizando MPI + OpenMP, possibilitando a adaptação a códigos já existentes. A biblioteca realiza o balanceamento em dois níveis – no nível dos processos MPI e no nível das threads OpenMP.

Na Seção 3.1 apresenta-se o modelo de aplicação para o qual a biblioteca foi implementado. A Seção 3.2 descreve o algoritmo de balanceamento implementado. Por sua vez, a Seção 3.3 descreve as funções da biblioteca e o modelo de uso.

### 3.1. Modelo das aplicações

A biblioteca é desenvolvida levando em conta o modelo de aplicação no qual um processamento é realizado iterativamente sobre um certo domínio, onde cada iteração depende da anterior. Exemplos de rotinas computacionais científicas incluem o método de Jacobi, métodos baseados em malha, processamento de sinais e processamento de imagens.

A seguir, apresenta-se um esqueleto de uma aplicação iterativa implementada utilizando MPI e OpenMP, com a possibilidade de uso de acelerador (GPU com API CUDA).

```
1  ...
2  int displs[num_proc]; // Vetor com o offset do intervalo de trabalho de cada processo.
3  int counts[num_proc]; // Vetor com a qtde. de elementos computados por cada processo.
4  ...
5  #pragma omp parallel num_threads(num_threads)
6  for(int it = 0; it < num_iteracoes; it++)
7  {
8      int tid = omp_get_thread_num();  // Id da thread
9      int ini = displs[proc];
10     int fim = displs[proc] + counts[proc];
11
12     #pragma omp for
13     for(int i = ini; i < fim; i++)
14     {
15         if(thread_usa_GPU(tid)) CUDA_processa(i);
16         else                    processa(i);
17     }
18
19     //Sincronização dos dados
20     #pragma omp barrier
21     #pragma omp single
22     MPI_Allgatherv(&result[displs[proc]],
23                    counts[proc],
24                    MPI_DATATYPE,
25                    result,
26                    counts,
27                    displs,
28                    MPI_DATATYPE,
29                    MPI_COMM
30     );
31 }
```

O esquema generalizado dessas aplicações pode ser resumido da seguinte forma: (i) os dados são particionados sobre os processadores e subdivididos entre as threads (linha 13), (ii) a cada iteração, alguns cálculos independentes são executados em paralelo pelos processadores/núcleos (linhas 15 e 16) e (iii) ocorre alguma sincronização de dados (linha 22). É importante salientar que o arquivo executável é o mesmo para todos os nodos, independente se possui ou não GPU (ou acelerador). A escolha por utilizar ou não as rotinas de GPU é realizada pela função $thread\_usa\_GPU()$, que verifica se a máquina possui GPU e a relaciona a uma thread.

### 3.2. Algoritmo de balanceamento

O algoritmo utilizado é baseado no apresentado em [Acosta et al. 2013], no qual o princípio básico é tentar igualar o tempo de execução de cada unidade de processamento. Unidades de processamento com maiores tempos de execução recebem menos carga, enquanto que unidades de processamento com tempos de execução menores recebem mais carga.

Diferentemente do trabalho original, no balanceamento multinível proposto há a realização do balanceamento em dois níveis – no nível dos processos e no nível de threads. Assim, cada thread podem receber cargas de trabalho diferentes, podendo-se atrelar um acelerador a uma thread específica, a qual recebe a carga de trabalho apropriada para processamento usando o recurso adicional de hardware.

O balanceamento funciona como se segue. Seja $T[]$ um vetor com o tempo de trabalho de cada unidade de processamento, e counts[] um vetor com a carga de trabalho. Primeiramente, obtém-se o maior e menor $T[i]$ para observar se um determinado limiar

foi excedido. Caso não tenha sido, considera-se que o sistema está balanceado e o algoritmo é encerrado. Senão, calcula-se o poder relativo de cada unidade de processamento como a razão entre a carga de trabalho e o tempo de trabalho de tal unidade; este valor é armazenado em um vetor $RP[]$. Também é calculada a soma dos elementos do vetor $RP[]$, armazenado em $SRP$. Ou seja,

$$RP[i] = \frac{counts[i]}{T[i]} \qquad SRP = \sum_i RP[i] \tag{1}$$

O vetor $counts[]$ é então recalculado de forma que a carga de trabalho é distribuída para cada processo de acordo com a razão entre $RP[i]$ e $SRP$. Assim, se $problemSize$ é o tamanho do intervalo a ser dividido, então

$$counts[i] = round\left(problemSize * \frac{RP[i]}{SRP}\right) \tag{2}$$

Tendo calculado o vetor $counts[]$, o vetor $displs[]$ é atualizado de acordo. Se $[L, L + problemSize)$ representa o intervalo a ser dividido, então

$$\begin{aligned} &displs[0] = L \\ &displs[i] = displs[i-1] + count[i-1], \quad para \quad i \geq 1 \end{aligned} \tag{3}$$

O balanceamento é realizado alterando-se os vetores $counts$ e $displs$ a cada iteração, para alterar a carga dos processos MPI; e, internamente, mantém vetores equivalentes para as threads, a partir dos quais fornece um subintervalo contínuo de processamento para cada thread.

### 3.3. Biblioteca MultiBalance

A biblioteca MultiBalance[1] foi implementada em C e fornece 4 funções que devem ser inseridas para a realização do balanceamento de carga:

*MultiBalance_Init_lib(...):* Inicialização da biblioteca;
*MultiBalance_Finalize_lib(...):* Finalização da biblioteca;
*MultiBalance_begin_section(...):* Indica o início do trecho paralelo a ser balanceado;
*MultiBalance_end_section(...)* Indica o final do trecho a ser balanceado.

O balanceamento ocorre por meio da inserção das funções *MultiBalance_begin_section(...)* e *MultiBalance_end_section(...)*, no início e no final do trecho a ser balanceado, dentro do laço que controla as iterações da aplicação, como apresentado no exemplo a seguir. Note que o uso da biblioteca demanda poucas alterações no código original, bastando a inserção das 4 funções de modo apropriado. Neste caso também se remove o *pragma omp for*, pois a divisão será feita pela biblioteca, onde cada subintervalo é indicado pelos valores de *ini* e *fim*.

[1] `https://github.com/luistrivelatto/TCC`

```
1  ...
2      MultiBalance_Init_lib(0, N, MPI_COMM);
3      ...
4      MultiBalance_init_section(it, counts, displs, limiar, tid, &ini, &fim);
5      pragma omp for
6      for(int i = ini; i < fim; i++)
7      {
8          if(thread_usa_GPU(tid)) CUDA_processa(i);
9          else                    processa(i);
10     }
11     MultiBalance_end_section(it, tid);
12     ...
13     MultiBalance_Finalize_lib();
14
15 ...
```

## 4. Avaliação Experimental

Nesta seção apresenta-se um conjunto de experimentos nos quais emprega-se a biblioteca MultiBalance no balanceamento de carga em aplicações iterativas.

Inicialmente, descreve-se o ambiente computacional utilizado e nas seções subsequentes, apresentam-se os casos de teste e os resultados demonstrando a efetividade da solução proposta.

### 4.1. Ambiente Computacional

Os experimentos foram executados em um cluster heterogêneo composto por três nodos com as seguintes configurações:

**Máquina 1** processador Intel Core i3-2100, com 2 núcleos a 3,10 GHz, 4 GB RAM;
**Máquina 2** processadores Intel Xeon E5620, com 4 núcleos a 2,40 GHz, 16 GB RAM, GPU Tesla K20c com 2.560 núcleos CUDA a 706 MHz e 5 GB RAM;
**Máquina 3** processador Intel Pentium Dual-Core E5200, com 2 núcleos a 2,50 GHz, 1 GB RAM.

A interligação dos nodos é feito por uma rede Gigabit Ethernet. Todas as máquinas tem como sistema operacional GNU-Linux Ubuntu 16.04 LTS. Utilizou-se o compilador C gcc-5.4.0, a biblioteca de troca de mensagens OpenMPI versão 1.10.2-8, e CUDA Toolkit 9.2. A tecnologia HyperThreading foi desabilitada em todos os processadores.

### 4.2. Casos de teste

Foram implementadas duas aplicações iterativas para a validação da biblioteca MultiBalance: (1) Método de Jacobi [Kelley 1995] e (2) Problema da Alocação de Recursos (*Resource Allocation Problem* - RAP) [Ibaraki e Katoh 1988].

O **método de Jacobi** é um algoritmo para determinar as soluções de um sistema de equações lineares de forma iterativa. Seja

$$Ax = b \tag{4}$$

um sistema de $N$ equações lineares, onde:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \; x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \; e \; b = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \tag{5}$$

Por meio de operações algébricas, é possível obter a seguinte aproximação para $x$:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left( b_i - \sum_{i \neq j} a_{ij} x_j^k \right), i = 1, 2, ...n \tag{6}$$

onde $x^k$ representa a $k$-ésima aproximação de $x$, e $x^{k+1}$ representa a próxima aproximação, na iteração seguinte. Desta forma, uma aproximação inicial é usada para obter $x^0$ e, a partir destes valores, a solução é obtida iterativamente aproximando mais a solução anterior.

Para paralelizar o algoritmo, divide-se a computação das variáveis entre as unidades de processamento. Ao fim de cada iteração, utiliza-se uma comunicação coletiva para compartilhar os valores calculados do vetor $x$ entre os processos.

O **Problema da Alocação de Recursos** (RAP) consiste em alocar $M$ unidades de um recurso indivisível para $N$ tarefas de forma a maximizar a eficiência do sistema. O problema pode ser enunciado como:

$$max\ z = \sum_{j=1}^{N} f_j(x_j) \quad sujeito\ a \sum_{j=1}^{N} x_j = M \tag{7}$$

onde $f_j(x_j)$ é o ganho de se alocar $x_j$ unidades do recurso para a tarefa $j$. Uma abordagem para resolver o problema é utilizar programação dinâmica. Seja $G[i][j]$ o melhor ganho considerando as primeiras $i$ tarefas e as primeiras $j$ unidades do recurso. Então pode-se definir as seguintes equações de recorrência:

$$\begin{aligned} &G[i][j] = max\{G[i-1][j-x] + f_i(x), 0 < x \leq j\},\ para\ i \geq 2 \\ &G[1][j] = f_1(j),\ para\ 0 < j \leq M \\ &G[i][0] = 0 \end{aligned} \tag{8}$$

O valor de $G[N][M]$ então contém o benefício máximo para o problema. Para calcular esse valor, deve-se calcular todos os valores da tabela. Uma vez que cada linha depende dos valores da linha anterior, a tabela pode ser computada linha a linha.

O algoritmo pode ser paralelizado dividindo o intervalo $[0, M + 1)$ entre as unidades de processamento, de forma que, a cada uma das $N$ iterações, cada unidade é responsável por computar parte das colunas da matriz. Ao fim da $i$-ésima iteração, uma comunicação coletiva replica a linha $i$ da matriz em todos os processos, de forma a calcular a próxima iteração.

Observa-se que a carga de trabalho é irregular, uma vez que diferentes elementos de uma mesma linha da matriz requerem diferentes quantidades de processamento. Especificamente, a computação do elemento da $j$-ésima coluna custa $O(j)$. Neste caso, a aplicação da biblioteca de balanceamento torna-se ainda mais interessante, até mesmo em um arquitetura homogênea, pois a divisão ótima da carga de trabalho é não trivial.

Foram implementadas cinco versões para o método de Jacobi e para o RAP: (1) sequencial, (2) paralela com OpenMP e MPI sem balanceamento, (3) paralela com OpenMP e MPI com balanceamento, (4) paralela com OpenMP, MPI e CUDA sem balanceamento, e (5) paralela com OpenMP, MPI e CUDA com balanceamento.

Nas versões sem balanceamento, utilizou-se uma distribuição uniforme da carga de trabalho entre os processos MPI. Esta carga foi distribuída uniformemente entre as threads disponíveis. Nos testes com balanceamento, utilizou-se limiar de 0% para a biblioteca, o que significa que o balanceamento sempre será realizado.

As implementações sequenciais foram executadas na máquina 1, a qual apresentou o melhor desempenho. Nas execuções com CUDA, três threads executam trabalho útil na CPU e uma thread é utilizada para controle da GPU, a qual realiza todos os cálculos atribuídos à thread.

### 4.3. Resultados

Essa seção apresenta os resultados do uso da biblioteca MultiBalance.

Nos gráficos das seções a seguir, os processos são referidos como P1, P2 e P3, enquanto as threads são referidas como P1 T1 para indicar a thread 1 do processo 1, P1 T2 para indicar a thread 2 do processo 1, e assim por diante.

#### 4.3.1. Jacobi

Para os testes do método de Jacobi foram utilizadas 10.000 variáveis com a execução de 5.000 iterações. Embora o método convirja mais rapidamente, configurou-se o experimento com essa quantidade de iterações de modo a gerar carga de trabalho. A Tabela 1 apresenta o tempo médio (de 10 execuções) de cada implementação.

**Tabela 1. Tempo de execução e *speedup* por implementação.**

| | Sequencial | OpenMP + MPI | OpenMP + MPI Balanc. | OpenMP + MPI + CUDA | OpenMP + MPI + CUDA Balanc. |
|---|---|---|---|---|---|
| Tempo (segs.) | 1564 | 470 | 303 | 470 | 245 |
| Speedup vs sequencial | - | 3,32 | 5,17 | 3,32 | 6,39 |
| Speedup vs não balanc. | - | - | 1,55 | - | 1,92 |

Observa-se que as versões balanceadas apresentaram *speedup* de 1,55 e 1,92 com relação às implementações equivalentes sem balanceamento. O *speedup* maior na implementação com CUDA é explicado pelo mesmo motivo pelo qual as implementações com e sem CUDA, não balanceadas, apresentam tempos de execução praticamente iguais: o gargalo de cada iteração ocorre no processo 3, e os outros processos ficam ociosos até que o processo 3 finalize sua parte do processamento. Ou seja, em função do desbalanceamento de carga, não há benefício nenhum em utilizar a GPU em vez de utilizar um núcleo da CPU. Esse comportamento pode ser melhor compreendido ao se observar a Figura 1, que ilustra o tempo médio e a carga média por iteração para os processos/threads.

Destaca-se que o uso do balanceamento permite aproveitar o maior poder de execução da GPU (P2 T1), que recebe uma fração significativa da carga de trabalho, igualando seu tempo de trabalho com o tempo das demais threads. Também pode-se observar que, nas execuções balanceadas, o tempo médio de trabalho em cada iteração de todas as threads é praticamente o mesmo, diminuindo o tempo ocioso de cada thread (de 50 ms para cerca de 0.8 ms) e melhorando o uso da arquitetura.

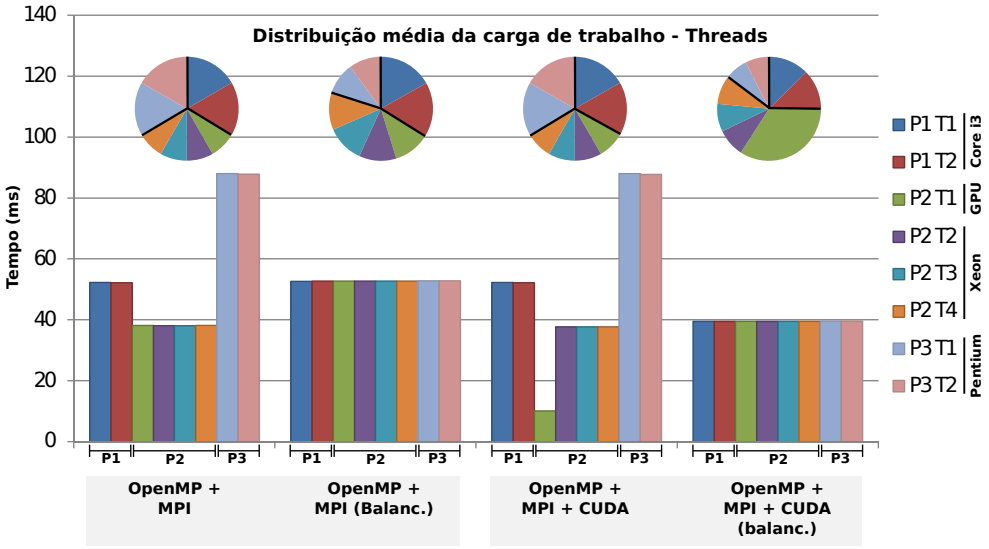



**Figura 1. Resultado do uso da biblioteca de balanceamento de carga. Gráficos de barras: tempo de trabalho médio por iteração por processo/thread. Gráficos de pizza: distribuição média da carga de trabalho entre as threads.**

A Figura 2 apresenta o tempo de trabalho e carga das threads nas primeiras iterações do teste empregando CPUs e GPU. Note que, em poucas iterações a carga já está próxima do balanceamento ideal; na quarta iteração, a diferença entre o maior e o menor tempo de trabalho entre as threads foi de 0,4 milissegundo.

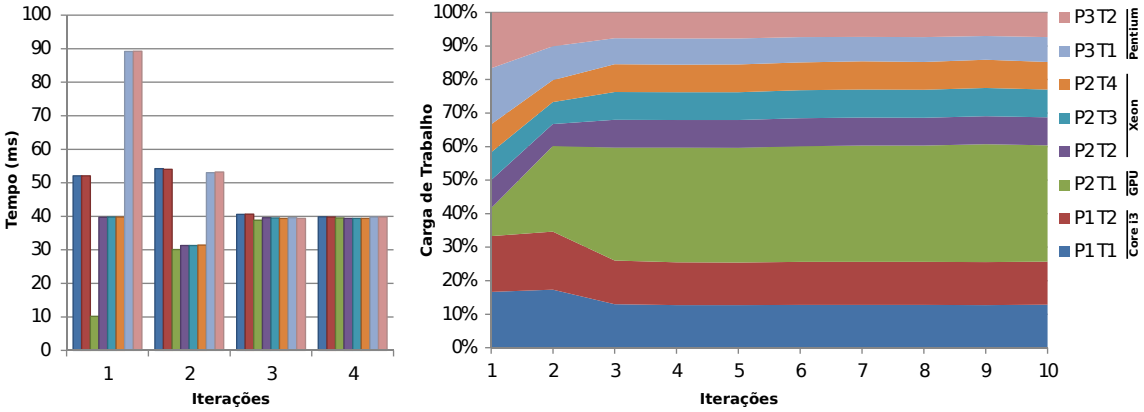



**Figura 2. Tempo de trabalho e distribuição da carga de cada thread nas primeiras iterações.**

Nos experimentos apresentados, o uso da biblioteca acrescenta cerca de 2 ms por iteração, devido ao cálculo dos tempos e comunicação das novas distribuições de carga.

### 4.3.2. RAP

As implementações do RAP foram executadas com 5.000 tarefas e 10.000 unidades do recurso (portanto, 5.000 iterações). A Tabela 2 apresenta o tempo médio (de 10 execuções) de cada implementação.

As versões com a biblioteca de balanceamento apresentaram speedup de 2,68 e 11,18 com relação às versões sem balanceamento. Novamente, a implementação balanceada com CUDA apresenta speedup maior que a implementação balanceada sem CUDA, pelo mesmo motivo observado nos testes do método de Jacobi. Inclusive, a implementação não balanceada com CUDA apresenta tempo médio de execução levemente maior do que a implementação sem CUDA. Isso se deve à sobrecarga causada pela inicialização da GPU (alocação e cópia de memória) antes de iniciar as iterações.

**Tabela 2. Tempo de execução e *speedup* por implementação.**

| | Sequencial | OpenMP + MPI | OpenMP + MPI Balanc. | OpenMP + MPI + CUDA | OpenMP + MPI + CUDA Balanc. |
|---|---|---|---|---|---|
| Tempo (segs.) | 874 | 513 | 191 | 515 | 46 |
| Speedup vs sequencial | - | 1,70 | 4,56 | 1,69 | 19,04 |
| Speedup vs não balanc. | - | - | 2,68 | - | 11,18 |

É de se destacar o speedup considerável da versão balanceada com GPU, mais que 3 vezes maior que o speedup observado na mesma implementação no método de Jacobi. Como se pode observar na Figura 3, que apresenta o tempo de trabalho e a carga média por iteração para os processos/threads, a GPU apresentou desempenho bem superior às CPUs neste problema, ficando responsável, em média, por 79% da carga da trabalho. A implementação do RAP se adapta melhor ao modelo de paralelismo da GPU que o método de Jacobi, permitindo um maior desempenho.

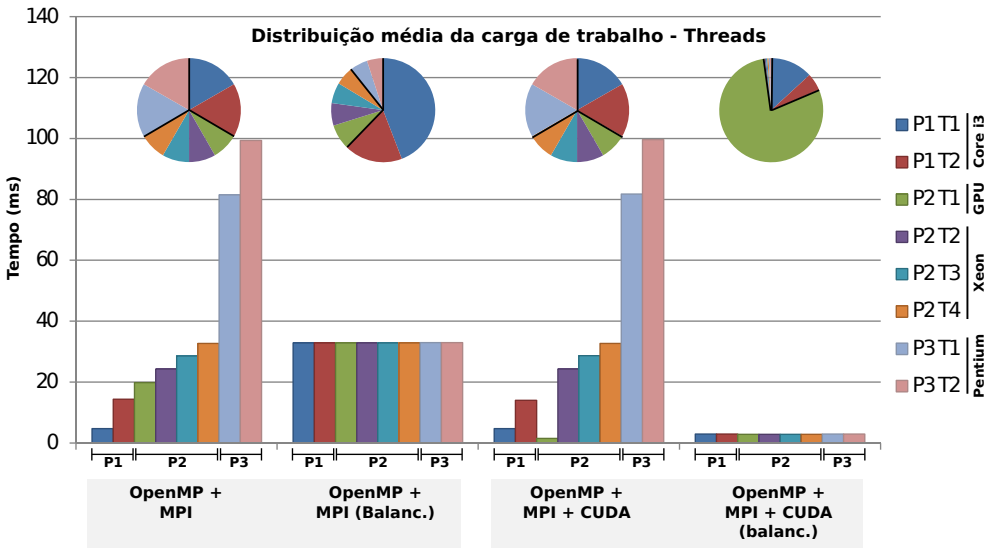



**Figura 3. Resultado do uso da biblioteca de balanceamento de carga. Gráficos de barras: tempo de trabalho médio por iteração por processo/thread. Gráficos de pizza: distribuição média da carga de trabalho entre as threads.**

Outro ponto a se ressaltar é que a implementação do RAP apresenta uma distribuição de dados irregular, o que faz com que o custo da carga de trabalho a ser dividida não seja uniforme - de forma simplificada, o $i$-ésimo elemento da carga de trabalho tem custo $i$. Como o tamanho da carga de trabalho a ser dividida era de 10.001 elementos, sendo que a GPU (P2 T1) ficou responsável, em média, pela carga no inter-

valo [1858, 9792), assim, os 79% da carga de trabalho correspondem a 92% do trabalho real por iteração.

Na Figura 3, pode-se ainda observar que, nas versões não balanceadas, núcleos de uma mesma máquina, com a mesma carga de trabalho, apresentaram tempos de trabalho diferentes em função da característica do problema. Na teoria, o algoritmo de balanceamento entende que esses núcleos possuem capacidades computacionais diferentes e altera a distribuição da carga para compensar por esse fator. Na realidade, os núcleos possuem a mesma capacidade, mas tal distribuição compensa a assimetria do custo da carga. Ou seja, os testes com o RAP demonstram a funcionalidade do algoritmo de balanceamento diante de problemas onde a distribuição ótima da carga é não trivial.

A Figura 4 apresenta o tempo de trabalho e carga dos processos/threads nas primeiras iterações do teste. Observa-se no gráfico que há oscilações na divisão de carga até por volta da iteração 15, quando a divisão começa a se estabelecer. De fato, na iteração 4, que pode ser observada nos gráficos de tempo de iteração (esquerda), a diferença entre maior e menor tempo de trabalho entre as threads era 6,1 milissegundos, correspondendo a 211% do menor tempo observado. Já na iteração 15, a diferença era de 0,2 milissegundos, correspondendo a 7% do menor tempo observado.

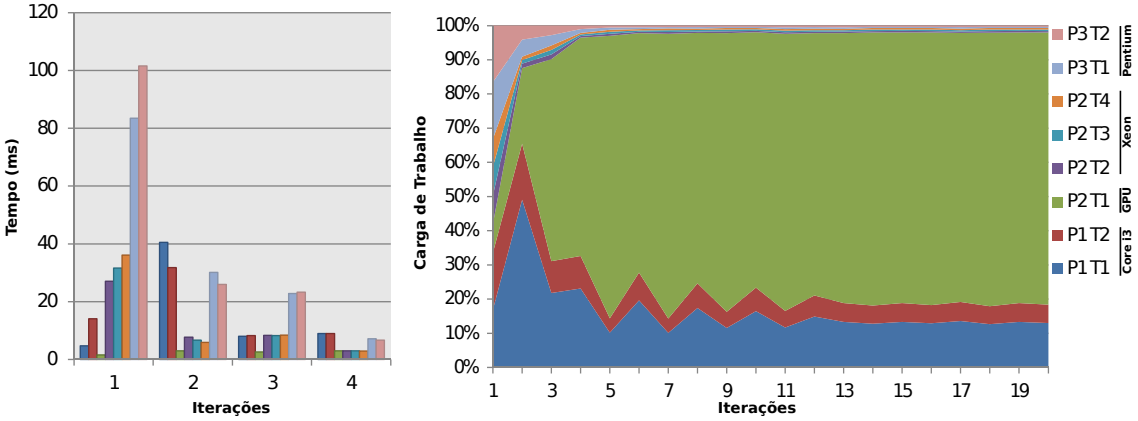



**Figura 4. Tempo de trabalho e distribuição da carga de cada thread nas primeiras iterações.**

Nos experimentos do RAP, o uso da biblioteca acrescenta cerca de 1 ms por iteração, devido ao cálculo dos tempos e comunicação das novas distribuições de carga.

## 5. Conclusão

Este trabalho teve como objetivo apresentar uma biblioteca de balanceamento de carga multinível para aplicações iterativas visando arquiteturas heterogêneas. A biblioteca é uma extensão do trabalho proposto por [Acosta et al. 2013], no qual realiza-se o balanceamento apenas no nível dos processos MPI. Assim, a principal contribuição desta pesquisa é a possibilidade de realizar o balanceamento da carga, primeiramente, entre os nodos de um cluster heterogêneo e, em um segundo nível, redistribuir a carga do nodo de modo apropriado entre os núcleos do processador e aceleradores.

O algoritmo de balanceamento apresentado foi validado por meio das implementações descritas nos estudos de caso. Os resultados apresentados mostram a

eficácia da biblioteca, onde foi possível constatar melhoria no uso efetivo da arquitetura, bem como na redução dos tempos de execução das aplicações. Os testes mostram ainda que a adição de unidades com maior capacidade de processamento, como as GPUs, pode ser completamente infrutífera se não forem tomados os cuidados para garantir que tais unidades não sejam limitados por meio de gargalos gerados por outros componentes da arquitetura.

Como trabalhos futuros estão previstos (1) a avaliação do algoritmo de balanceamento em problemas com cargas dinâmicas; (2) a avaliação de algoritmos de balanceamento baseados em outros critérios que não o tempo de execução (consumo de energia, por exemplo); e (3) adaptação da biblioteca para uso em outras classes de aplicações paralelas.

## Referências

Acosta, A., Blanco, V., e Almeida, F. (2013). Dynamic load balancing on heterogeneous multi-gpu systems. *Comput. Electr. Eng.*, 39(8):2591–2602.

Bosque, J. L., Toharia, P., Robles, O. D., e Pastor, L. (2013). A load index and load balancing algorithm for heterogeneous clusters. *The Journal of Supercomputing*, 65(3):1104–1113.

Boyer, M., Skadron, K., Che, S., e Jayasena, N. (2013). Load balancing in a changing world: Dealing with heterogeneity and performance variability. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 21:1–21:10, New York, NY, USA. ACM.

Cabrera, A., Acosta, A., Almeida, F., e Blanco, V. (2018). Energy efficient dynamic load balancing over multigpu heterogeneous systems. In Wyrzykowski, R., Dongarra, J., Deelman, E., e Karczewski, K., editors, *Parallel Processing and Applied Mathematics*, pages 123–132, Cham. Springer International Publishing.

Da Costa, G., Fahringer, T., Rico-Gallego, J.-A., Grasso, I., Hristov, A., Karatza, H., Lastovetsky, A., Marozzo, F., Petcu, D., Stavrinides, G., Talia, D., Trunfio, P., e Astsatryan, H. (2015). Exascale machines require new programming paradigms and runtimes. *Supercomput. Front. Innov.: Int. J.*, 2(2):6–27.

Ibaraki, T. e Katoh, N. (1988). *Resource Allocation Problems: Algorithmic Approaches*. MIT Press, Cambridge, MA, USA.

Kelley, C. T. (1995). *Iterative Methods for Linear and Nonlinear Equations*. SIAM.

Lu, F., Song, J., Yin, F., e Zhu, X. (2012). Performance evaluation of hybrid programming patterns for large cpu/gpu heterogeneous clusters. *Computer Physics Communications*, 183(6):1172–1181.

Menon, H. e Kalé, L. (2013). A distributed dynamic load balancer for iterative applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 15:1–15:11, New York, NY, USA. ACM.

Mittal, S. e Vetter, J. S. (2015). A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35.

# On the Efficiency of Transactional Code Generation: A GCC Case Study

**Bruno Chinelato Honorio**[1], **João P. L. Carvalho**[2], **Alexandro Baldassin**[1]

[1]Univ. Estadual Paulista – UNESP, Rio Claro, Brazil

[2]Institute of Computing – UNICAMP, Campinas, Brazil

brunochonorio@gmail.com,alex@rc.unesp.br,joao.carvalho@ic.unicamp.br

***Abstract.*** *Memory transactions are becoming more popular as chip manufacturers are building native support for their execution. Although current Intel and IBM microprocessors support transactions in their instruction set architectures, there is still room for improvement in the compiler and runtime front. The GNU Compiler Collection (GCC) has language support for transactions, although performance is still a hindrance for its wider use. In this paper we perform an up-to-date study of the GCC transactional code generation and highlight where the main performance losses are coming from. Our study indicates that one of the main source of inefficiency is the read and write barriers inserted by the compiler. Most of this instrumentation is required because the compiler cannot determine, at compile time, whether a region of memory will be accessed concurrently or not. To overcome those limitations, we propose new language constructs that allow programmers to specify which memory locations should be free from instrumentation. Initial experimental results show a good speedup when barriers are elided using our proposed language support compared to the original code generated by GCC.*

## 1. Introduction

The lack of good abstractions and tools for parallel programming in the face of multicore processors and the concurrency revolution has created an urge for new paradigms [Sutter and Larus 2005]. One such paradigm is known in the literature as *Transactional Memory* (TM), wherein the main unit of concurrency is conveyed by means of *transactions* [Harris et al. 2010]. One key property of a transaction is its all-or-nothing nature: either all instructions of a transaction are executed or none is. Therefore, TM increases the abstraction level by moving the concurrency control from programmers to the underlying implementation infrastructure, which can be realized in hardware (HTM), software (STM), or a mix of both (HyTM).

During the past few years a lot of progress has been made in the research of TM systems which, in turn, stimulated the development and subsequent availability of compiler and hardware support [Intel 2012, Nakaike et al. 2015, Ruan et al. 2014b]. In particular, the GNU Compiler Collection (GCC) supports the use of transactions since 2014 (for the C language) and a technical specification for C++ was formalized in 2015 [SC22 2015]. Moreover, chip manufacturers such as IBM and Intel have hardware support for transactions in their main line of processors since 2013 [Wang et al. 2012, Jacobi et al. 2012, Intel 2012].

Despite all the research efforts in the last decade and the recent advances in the language and hardware spheres, TM is still not commonly used commercially in the development of new concurrent software. One of the main reasons is due to compiler inefficiencies, of which compiler over-instrumentation is one of the major villains [Yoo et al. 2008, Dragojevic et al. 2009]. When generating code, a compiler instruments all accesses (reads and writes) to shared memory by inserting *barriers* into the generated code. These barriers are used by STM runtimes to detect conflicts among transactions and concurrency control. Since compilers usually have limited information at compile time, they must be conservative and insert barriers even in situations where they will not be necessary.

In order to shed some light into the current scenario of compiler inefficiencies, this paper performs a thorough evaluation of transactional applications and proposes a language support for explicit barrier elision. In particular, we perform an extensive evaluation of the GCC compiler with several STM runtimes and characterize the amount of barriers generated. Our language support is provided by means of *pragmas* to elide instrumentation of known memory regions, allowing experienced programmers to improve the performance of transactional code. Previous works have provided hints at the main source of overhead for different compilers [Yoo et al. 2008, Dragojevic et al. 2009, Ruan et al. 2014b], but no study exists targeting a state-of-the-art compiler with TM support. This work seeks to fill that void.

This paper makes the following contributions:

- It proposes a language construct to allow programmers to instruct the compiler to elide barriers for certain memory regions (see Section 3);
- It characterizes the over-instrumentation present in the STAMP benchmark [Minh et al. 2008] (see Section 4.2);
- It shows the performance gained with the elision of barriers for some of the STAMP benchmarks, where a speedup of up to 7.2x with regard to the GCC baseline was observed (see Section 4.3).

This paper is organized as follows. Section 2 presents the backgrounds required to appreciate our work and the related works. Section 3 describes the new proposed pragma for barrier elision. Section 4 first performs a study on the number of barriers presented in the STAMP benchmark and then show the speedup one can obtain by using the proposed pragma. Finally, we conclude the paper in Section 5.

## 2. Background and Related Work

Transactional Memory (TM) [Harris et al. 2010] makes use of *transactions* as the main unit of concurrency. Much like their database counterparts, memory transactions encapsulate a group of instructions such that their effects on memory (read and write operations) are performed atomically. In other words, either all instructions belonging to a transaction are executed or none is. Herlihy and Moss [Herlihy and Moss 1993] first introduced the TM concept back in 1993 as a hardware support. The research in the area flourished in the first decade of the 21st century, along with the introduction of multicore processors.

TM systems can be implemented in software (STM), hardware (HTM) or a mixed of both (Hybrid TM, or HyTM). Much of the earlier works focused on STM because of

their flexibility and the lack of real hardware support. The introduction of hardware support for TM by companies such as IBM and Intel [Wang et al. 2012, Jacobi et al. 2012, Intel 2012] renewed the interest in the area, particularly for HyTM systems.

In terms of programmability, TM systems were initially deployed as software libraries. Only recently language support started to appear, although Intel offered compiler support for TM since 2008 (today discontinued) [Intel 2009]. Language extensions for C++ were formally described in the Transactional Memory Technical Specification (TMTS) in 2015 [SC22 2015]. It is a result of years of experience with library-based TM systems and involved a collaboration among hardware vendors (Intel, IBM, and old Sun Microsystems), programming language folks (mostly notably Red Hat and Oracle), C++ experts (including HP and Google) and also academic researchers. The GNU Compiler Collection (GCC) has support for transactions in the C language since 2014 (as of version 4.7) and, for C++, since 2016 (version 6.1).

## 2.1. Compiler Instrumentation

The compiler, given a program with transactional blocks, must generate code to initialize and commit transactions, as well as instrument shared-variables accesses within such blocks. The only compiler with transactional memory support that is been actively developed and maintained, to the best of our knowledge, is part of GNU's Compiler Collection (GCC). GCC's transactional support is composed of two parts : (i) transactional annotations to enable automatic memory access instrumentation, and (ii) a runtime library with a couple of transactional memory systems available out of the box.

GCC provides mechanisms to both annotate code blocks and function attributes to selectively instrument accesses within functions. Memory accesses inside `__transaction_atomic` blocks are instrumented with calls to the runtime library that implement transactional barriers used to detect conflicts. Every `__transaction_atomic` block is transformed into a two-path code: one with the transactional barriers to be used with STMs and another without barriers that can be used with HTMs or an alternative synchronization strategy (e.g., holding a global lock). The TM support on GCC offers two types of function attributes: `__transaction_safe` and `__transaction_pure`. For every function with the transactional-safe attribute, the compiler will generate, apart from the original function, a version with all memory access done via barriers. The compiler assumes that transactional-pure functions do not have side-effects, therefore memory accesses inside them are generated without barriers.

GCC follows Intel's Transactional Memory ABI (Application Binary Interface) to generate transactional code [Intel 2009]. Intel's TM ABI defines the interface between code generated for IA32 architectures and the transactional memory runtime. It is implemented as a shared library, enabling users to switch between different TM implementations without recompiling the code.

The TM support shipped today with GCC is still in an experimental phase, meaning that it might still miss a set of optimizations and thus might not deliver a satisfying performance. One of the biggest sources of inefficiency for transactional code are due to unnecessary read and write barriers. This problem is even more pronounced with software implementations, since each barrier invocation leads to metadata lookup to check the consistency of the execution. Considering the performance cost that each barrier adds, the

compiler should only generate barriers to shared memory accesses. For example, local variables do not need instrumentation, since their visibility is limited by the scope of the transaction. Unfortunately, it is not easy to determine at compile time which variables should or should not be instrumented. This impossibility forces the compiler to take a pessimistic approach and add unnecessary barriers. This phenomenon has been described as *over-instrumentation* [Yoo et al. 2008, Dragojevic et al. 2009]. The implementation available today through GCC assumes that the runtime will be able to dynamically elide such barriers by not producing metadata for them.

To illustrate how over-instrumentation happens, consider the example shown in Code 1. In this example, `foo` is a function that starts a transaction (line 2) which first creates a linked-list (line 3) and then calls `initList` passing over the list (line 4). After some processing with `L`, the transaction publishes some elements of `L`. Note the elements of `L` will only become visible to other transactions when, and if, the publisher transaction commits. Therefore, all elements of `L` could be accessed in `initList` without barriers. However, as `initList` is annotated with the attribute `__transaction_safe`, the compiler will create an instrumented version of the function. The loop from line 13 through 16 walks through the linked-list, zeroing out each element (line 14). From the point of view of `foo`, the linked-list `L` is only local but, since the compiler can not make such deduction, all read and write access to the list will be done via barriers in `initList`.

**Code 1. Example of over-instrumentation.**

```
1  void foo() {
2    __transaction_atomic {
3      list_t *L = createList();
4      initList(L);
5      // some processing with L
6      // publish elements of L
7    }
8  }
9
10 __attribute__((transaction_safe))
11 static void initList(list_t *l) {
12   node_t * ptr = l->head;
13   while (ptr != NULL) {
14     ptr->v = 0;
15     ptr = ptr->next;
16   }
17
18 }
```

**Code 2. Assembly generated from Code 1**

```
1  foo:
2  call _ITM_beginTransaction
3  # ...
4  call createList
5  # ...
6  call initList
7  # some processing with L
8  # publish elements of L
9  call _ITM_commitTransaction
10 # ...
11 # ...
12 initList:
13 # node_t* ptr = l->head
14 call _ITM_RU8
15 # if ptr == NULL, goto RETURN
16 testq %rax, %rax
17 je  .L9
18 movq  %rax, %rbx
19 WHILE:
20 leaq  %rbx, %rdi
21 # ptr->v = 0;
22 xorl  %esi, %esi
23 call _ITM_WU4
24 # ptr = ptr->next;
25 movq  %rbx, %rdi
26 call _ITM_RU8
27 # if ptr != NULL, goto WHILE
28 testq %rax, %rax
29 movq  %rax, %rbx
30 jne WHILE
31 RETURN:
32 retq
```

Code 2 shows, in a simplified manner, the assembly code generated by GCC for the example in Code 1. Lines from 1 to 11 refer to the procedure `foo`, while lines 12 to 32

refer to the procedure `initList`. In this portion of code, the words in boldface indicate the calls to the transactional ABI to start a transaction (line 2), allocate the linked-list (line 4) and commit the transaction (line 9). The read barriers to access the head of `L` (line 14) and the `next` field (line 26), as well as the write barrier to reset the field `v` (line 23) are also in boldface. This example shows that, in fact, the compiler instruments all access to the linked-list `L`. As every access will imply a call to the transactional library, it is clear the instrumented path will exhibit a very high overhead compared with the uninstrumented path. The compiler is unable to infer that the list is private to the transaction and, therefore, it inserts transactional barriers to every load and store inside `initList`.

## 2.2. Related Work

Harris et al. were among the first to describe compiler optimizations for TM in a research compiler named Bartok back in 2006 [Harris et al. 2006]. Some of the static analyses performed by their compiler include code-motion optimizations of transactional barriers and instrumentation avoidance for newly allocated objects. They also discuss runtime optimizations for transaction-local objects to avoid extra instrumentation. In the same direction, Adl-Tabatabai et al. propose compiler and runtime optimizations for the StarJIT compiler [Adl-Tabatabai et al. 2006]. It is important to notice that both works employed managed runtime environments.

Wang et al. first discussed the challenges of optimizing transactional code in an unmanaged language (C/C++) [Wang et al. 2007]. They describe optimizations to perform redundant barrier elimination, inlining of fast paths of STM routines, and transaction ckeckpointing. The first work to recognize the over-instrumentation problem performed by compilers is due to Yoo et al.[Yoo et al. 2008]. They propose a new construct, `pragma tm_waiver`, to let programmers specify if the compiler should instrument a particular block of code or function. This feature is similar to the one we are proposing in this paper, but it did not allow programmers to specify exactly which variables should not be instrumented. Wu et al. discuss optimizations for both manage and unmanaged environments using the IBM XL compiler and the Java virtual machine [Wu et al. 2009]. The work of Ruan et al. investigates the interplay between compiler instrumentation and performance of transactions [Ruan et al. 2013]. They claim that the compiler must consider the platform it is generating code to when determining which optimizations to apply.

The work closest to ours is the one due to Dragojević et al. [Dragojevic et al. 2009]. The authors investigate optimizations for captured memory, i.e., memory allocated inside transactions which cannot escape the allocating transaction. Accesses to captured memory do not require barriers and therefore can be elided by the compiler. A *capture analysis* is developed for both the runtime and the compiler. They implemented the analysis in the Intel C++ compiler based on the existing intra-procedural pointer analysis. Carvalho and Cachopo later extended the capture analysis for a managed environment based on a Java virtual machine [Carvalho and Cachopo 2013].

Although techniques based on capture analysis (at the compiler level) may provide some performance gains, the compiler still needs to be conservative and might not be able to elide all possible barriers. Therefore, the solution proposed in this work relies on programmers to inform the compiler which parts of the shared memory should be elided through pragmas. Although at first sight this could be viewed as an extra burden for

programmers, we envision this feature being used by experts who can have one more optimization tool at their disposal.

## 3. Language Support for Barrier Elision

This section presents a new language support for GCC that enables the elision of unnecessary transactional barriers. First, we show how to use the elision support, implemented as a new GCC pragma, called `tm`, and how to specify which variables must have their barriers elided (3.1). Then we describe how the pragma and the elision mechanism are implemented (3.2).

### 3.1. Elision pragma: syntax and usage

Pragma directives offer language extensions through simple and clear code annotations. Our `tm` pragma provides an `elidebar` clause to specify which variables must have their barriers elided in the scope of the block following the pragma.

The syntax of pragma `tm` is defined as follows:

```
#pragma tm elidebar (var1 [, var2 ...])
{
   //scope affected
}
```

After the `pragma-tm` directive, a list of variables for which transactional barriers should be elided are specified through the `elidebar` clause. The elision takes effect on the scope of each `pragma-tm` block, thus different transactional regions can have barriers eliding different variables. The `elidebar` clause enables an easy-to-use way to reduce over-instrumentation, as the results of Section 4 show.

### 3.2. Implementation

The GCC compiler has been developed and supported for over 30 years. It is a very sophisticated compiler with support for multiple languages (e.g. C/C++, Fortran, Ada and Go). Each supported language has a front-end in the compiler that does the lexical, syntactic, and semantic analysis. Since these languages could be used on multiple different architectures, the compiler needs to represent code in a way to enable language and architecture-independent optimizations. GCC has three intermediate representations (IR) that are generated in the following order: GENERIC, GIMPLE and RTL. GENERIC represent the code in trees, GIMPLE represent the code in three-address form. In the final representation, RTL, the code's instructions are described, one by one, in an algebraic form that describes what each instruction does. The machine code is finally generated after the compiler goes through all its intermediate passes, each working with a given representation.

The central data structure of the IR is a tree. All operations done inside the IR will always be around the trees being used as the input and the output of these operations. One such operation is the analysis to decide whether a given variable inside the transactional atomic block should be instrumented or not. These operations are performed by the `requires_barrier` function, defined in the `trans-mem.c` source file, which is responsible for optimizing the transactional memory constructs.

**Code 3. Simplified version of the requires Barrier function.**

```
1  static bool requires_barrier (tree x, //...)
2  {
3    tree orig = x;
4    while (handled_component_p (x))
5    x = TREE_OPERAND (x, 0);
6
7    switch(TREE_CODE(x))
8    {
9     //...
10    default:
11     return false;
12   }
13 }
```

Code 3 shows, in an simplified way, the `requires_barrier` function. This function receives a tree as a parameter (line 1), and through the `TREE_OPERAND` macro (line 5), it access the operands of tree `x`. To determine if an operand needs to be instrumented or not, a conditional test occurs (lines 7-12), using the parameter `TREE_CODE`, which represents operand types, such as declared variables, functions, memory references, and others. Depending on the operand's type, a set of operations will occur to analyze if it should be instrumented or not. If so, the function returns `TRUE`, otherwise it returns `FALSE`.

Since the whole analysis of variables either returns true or false, the pragma analysis, basically, returns false for each variable that appears within the `elidebar` clause. To know if the variable was specified by the clause or not, we added a flag to the node structure of the IR tree. If the flag is set, the compiler knows it is a variable specified in the pragma and, therefore, `requires_barrier` will return false for every instance of it that is within the elision scope.

## 4. Experimental Results

In this section we show a characterization of transactional accesses for STAMP (Section 4.2). We also show the performance gains that the programmer could achieve by eliding unnecessary barriers through our proposed mechanisms via a new GCC pragma (Section 4.3). But first, we start with a description of our experimental setup.

### 4.1. Experimental Setup

The following TM implementations are evaluated:

- **`NOrec`:** This is the code developed by the Rochester Synchronization Group and released as part of the RSTM package [Marathe et al. 2006]. `NOrec` employs a global sequence lock, performs deferred versioning and eager conflict detection by means of value-based validation;
- **`libitm`:** This is basically `NOrec` with its methods to start, commit, abort and load/store barriers wrapped by our implementation of the functions from Intel®'s TM ABI [Intel 2009] and used by the GCC runtime;
- **`libitm+elidebar`:** This is `libitm` with `__transaction_atomic` blocks annotated with our new GCC pragma `tm` and the `elidebar` clause to selectively omit transactional memory barriers.

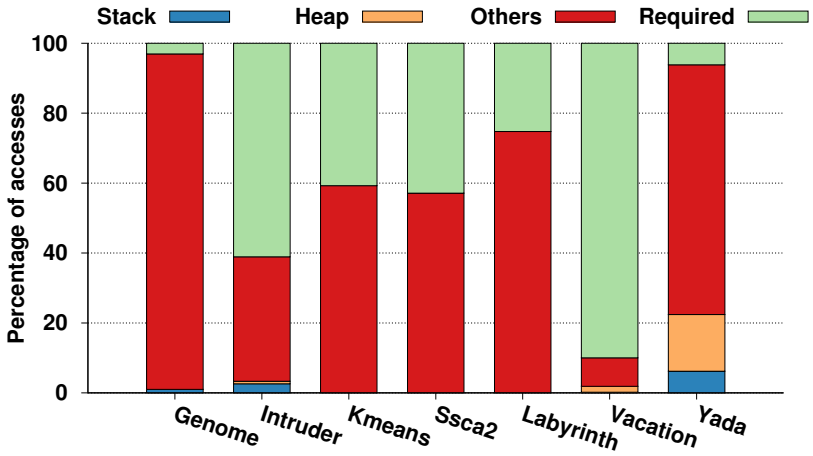



**Figure 1. Breakdown of all accesses for STAMP (reads and writes).**

The results reported in this section represent the mean of 20 executions. All applications and TM implementations were compiled with GCC (GNU C/C++ Compiler) 6.2, modified to implement our elision mechanism (Section 3.2) and optimization level three (`-O3`). In this section we present experiments conducted on a machine powered with Intel® Core™ i7-2600K 3.40GHz processor and 8GB of RAM. The i7-2600K has 4 physical cores and 2 hardware threads per core (total of 8 SMT threads). The machine runs CentOS 6.9 and Linux kernel 3.14. Our analysis makes use of the STAMP [Minh et al. 2008] benchmark suite. STAMP consists of scientific applications from a diversity of fields such as bioinformatics (`Genome`), security (`Intruder`) and computational geometry (`Yada`). It is important to notice that both binary, with manual and automatically inserted barriers, of STAMP applications were obtained by compiling the same source code.

### 4.2. Breakdown of STAMP Transactional Accesses

In order to provide a quantitative analysis about how many unnecessary barriers are inserted by GCC we followed a similar approach to Dragojević's et al. [Dragojevic et al. 2009]. For each STAMP applications we divided the number of barriers inside transactions into four types: those accessing captured memory to transactional-local stack (Stack) and heap (Heap), as well as required barriers (R) and non-required (NR) for other reasons (e.g., read-only). Similar to [Dragojevic et al. 2009] we executed each application (single-threaded) and assume that all manually annotated accesses are required. As noted by Dragojević et al., this assumption might over estimate the number of barriers, however, as the results show, it seems a reasonable assumption.

Stack accesses are detected at runtime by checking if the memory address read/written is within the range of the values returned by the builtin function `__builtin_dwarf_cfa` before starting the transaction and at the moment the read/write barrier is called. This builtin function returns the *Canonical Frame Address* (a.k.a call frame address) of the caller function. Captured heap accesses are detected using the set container from C++ STL. The set holds the addresses of memory allocated inside the transaction. The set is clear upon transactional commit or rollback.

Figure 1 shows the percentage of each type of access, both reads and writes, for each STAMP application. As the figure shows, 6 out of 7 applications have at least 40%

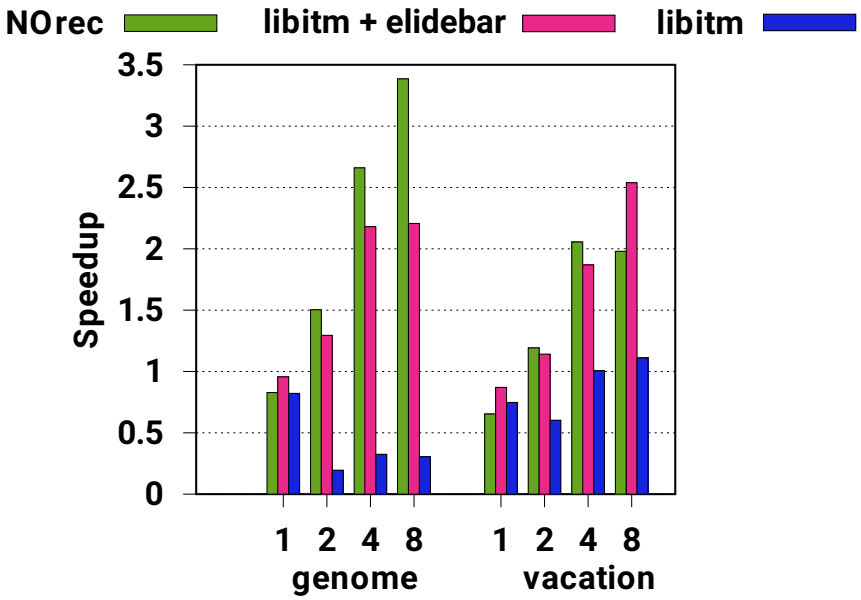



**Figure 2. Speedup relative to sequential version for `Genome` and `Vacation`.**

of accesses that could be elided. This result confirms the findings of Dragojević et al. and reveals that GCC heavily over instruments accesses both inside transactions and within transactional-safe functions. Dragojević et al. propose a mechanism to elide barriers to both stack and heap captured memory, however the majority of elidable accesses are neither of these types and, therefore, can not be elided without the help of the programmer. In order to better understand that, let's take `Genome` as an example.

`Genome` reconstructs a nucleotide sequence from, possibly, duplicated segments. The reconstruction is divided into three steps. First, duplicated segments are eliminated using a hash-table. After that, unique segments are matched, at their start/end positions, to find possible connection-points. At last, the sequence is reconstructed using the previous connection-points found. Each step has a thread-wait barrier at the end, so threads only move to the next step once all of them have finished the previous one. The first step takes over $90\%$ of the `Genome` execution time [de Carvalho et al. 2015] and in that step the only data which is modify by multiple threads is the hash-table itself. Thus, the only required barriers are those accessing the hasthtable. Yet, GCC inserts barriers even on functions which compare two nucleotide segments despite the fact that such segments are read-only on that step.

### 4.3. Pragma Elidebar to The Rescue

In order to show the performance gains that can be achieved by eliding unnecessary barriers via our new GCC pragma (Section 3), we selected two applications from STAMP (`Genome` and `Vacation`). Figure 2 shows the speedup (y-axis), over sequential execution time, for the baseline implementation with manually added barriers (`NOrec`), the automatically instrumented version with `pragma tm elidebar` (`libitm+elidebar`) and without (`libitm`). The figure shows speedup results of `Genome` and `Vacation` when executed with 1, 2, 4 and 8 threads (x-axis). Each application was executed 20 times, therefore Figure 2 shows the mean speedup. As the standard deviation was bellow $1\%$ of the mean, we omitted the error bars.

As Figure 2 shows, both Genome and Vacation scale well with NOrec. However, no speedup was observed with neither of them when executed under libitm. As discussed in 4.2, Genome spends most of its execution time in a phase in which most barriers are accessing read-only data and, therefore, can be elided. Such barriers were not placed on the manually instrumented version, explaining why Genome scales with NOrec. Once these unnecessary barriers are elided with pragma tm elidebar, a significant performance gain is observed, as the results with libitm+elidebar show. In fact, libitm+elidebar is over 7.2x faster than libitm with 8 threads for Genome. In addition, libitm+elidebar falls behind NOrec only by less than 54%. Such results were obtained by using a single pragma tm elidebar in the version of strcmp function in sequencer.c file.

Contrasting Genome, Vacation have only less than 12% of barriers that could be elided, as the breakdown of Figure 1 shows. A careful analysis of Vacation hotspots revealed that the application spends most of its time performing lookup operations on a map data structure. Such map is part of STAMP's libraries and is implemented as a red-black tree by default. Although most barriers inserted either by the compiler, or manually, are indeed required for correctness sake, we discovered an opportunity for optimization. Given that the semantics of a lookup is such that it only returns true if, and only if, the data structure contains the element been searched, therefore the only required barrier in such operations are those to ensure no violations happen. As a consequence, by adding a valid flag to the item been manipulated in the map, its possible to elide all previously inserted barriers via the elidebar clause, seeing that all operations on the map always check the valid flag through barriers. It is sufficient to abort a lookup if during the traversal an invalid element is reached. Remove operations should clear the valid flag and insert operations should set it. With this optimization we noticed significant performance improvement as Figure 2 shows. With most barriers elided with the elidebar clause, libitm+elidebar is over 2.2x faster than libitm. Although the optimization employed does not necessarily require the mechanism of pragma tm elidebar, as the barriers could be elided manually by the programmer, it is not hard to see that such activity might become time consuming and error-prone. In addition, the optimization performed on Vacation is applicable to a broader scenario. For example, with the ability to selectively choose which barriers to elide, the programmer is capable to ensure that only the required fields of a shared object are instrumented. All remaining fields can have their unnecessary barriers elided easily through the elidebar clause.

Although our results do show some significant speedups relative to the baseline (libitm) for the two STAMP applications discussed, it should be noticed that using the pragma requires considerable understanding of how the application works and how data is shared. This is usually the case when a new application is being coded for the first time, but might require a lot of time otherwise. In the case of the STAMP benchmarks, we reported the results for the two applications (Genome and Vacation) we had chosen (based on the profile stage of Section 4.2 and previous work [de Carvalho et al. 2015]). Apart from these applications, only Yada and Intruder might have space for improvement as Figure1 shows. As previous results show [de Carvalho et al. 2015], both Kmeans and SSCA2 exhibit very small speedups in SW and are more fitted to HTM. The version of Labyrinth modified by Spear at al. [Ruan et al. 2014a] has very short transactions, with very low conflict probability and, therefore exhibits only a negligible overhead with

`libitm` compared to manual instrumentation. Nevertheless, both remaining candidate applications, and particularly `Yada`, may require a significant analysis and understanding of its data structures and code to enable correct usage of elision pragmas.

## 5. Conclusion

In this paper we conducted a study showing that the GCC may insert a lot of unnecessary read/write barriers by characterizing the applications from the STAMP benchmark suite. In order to allow programmers to avoid compiler over-instrumentation, we proposed a new language support for GCC via a new pragma clause to instruct the compiler to elide barriers for the selected shared variables.

Our results applying the proposed pragma in two applications (`Genome` and `Vacation`) showed a performance gain of 7.2x for `Genome`, and a 2.2x gain for `Vacation` with 8 threads over the baseline (standard GCC). The significant speedups obtained by using our new language construct shows the great potential of performance gain that lies in solving the over-instrumentation phenomenom presented in transactional code generation. Although using the proposed pragma is not trivial, we believe that it is a powerful tool for programmers to unleash all the potential of the transacional model.

## Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments. This work was supported by FAPESP (grants 2016/15337-9 and 2016/12103-7), and Center for Computational Engineering and Sciences (CCES).

## References

Adl-Tabatabai, A.-R., Lewis, B. T., Menon, V., Murphy, B. R., Saha, B., and Shpeisman, T. (2006). Compiler and runtime support for efficient software transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37.

Carvalho, F. M. and Cachopo, J. (2013). Runtime elision of transactional barriers for captured memory. In *Proceedings of the 18th Symposium on Principles and Practice of Parallel Programming*, pages 303–304.

de Carvalho, J. P. L., Murari, R., and Baldassin, A. (2015). Reavaliando o conjunto de aplicações stamp em um novo hardware transacional. In *Anais do Simposio em Sistemas Computacionais de Alto Desempenho*, pages 216–227.

Dragojevic, A., Ni, Y., and Adl-Tabatabai, A.-R. (2009). Optimizing transactions for captured memory. In *Proceedings of the 21st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 214–222.

Harris, T., Larus, J., and Rajwar, R. (2010). *Transactional Memory*. Morgan & Claypool Publishers, 2 edition.

Harris, T., Plesko, M., Shinnar, A., and Tarditi, D. (2006). Optimizing memory transactions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–25.

Herlihy, M. and Moss, J. E. B. (1993). Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300.

Intel (2009). *Intel Transactional Memory Compiler and Runtime Application Binary Interface*. Intel Corporation, 1.1 edition.

Intel (2012). *Intel® Architecture Instruction Set Extensions Programming Reference*. Intel Corporation.

Jacobi, C., Slegel, T., and Greiner, D. (2012). Transactional memory architecture and implementation for IBM system z. In *Proceedings of the 45th ACM/IEEE International Symposium on Microarchitecture*, pages 25–36.

Marathe, V. J., Spear, M. F., Heriot, C., Acharya, A., Eisenstat, D., Scherer, W. N., and Scott, M. L. (2006). Lowering the overhead of nonblocking software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*.

Minh, C. C., Chung, J., Kozyrakis, C., and Olukotun, K. (2008). STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46.

Nakaike, T., Odaira, R., Gaudet, M., Michael, M. M., and Tomari, H. (2015). Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42nd International Symposium on Computer Architecture*, pages 144–157.

Ruan, W., Liu, Y., and Spear, M. (2014a). Stamp need not be considered harmful. In *Ninth ACM SIGPLAN Workshop on Transactional Computing*.

Ruan, W., Liu, Y., Wang, C., and Spear, M. (2013). On the platform specificity of STM instrumentation mechanisms. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 1–10.

Ruan, W., Vyas, T., Liu, Y., and Spear, M. (2014b). Transactionalizing legacy code: An experience report using GCC and memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 399–412.

SC22, I. J. (2015). *Technical Specification for C++ Extensions for Transactional Memory*. ISO copyright officer, n4514 edition.

Sutter, H. and Larus, J. (2005). Software and the concurrency revolution. *Queue*, 3(7):54–62.

Wang, A., Gaudet, M., Wu, P., Amaral, J. N., Ohmacht, M., Barton, C., Silvera, R., and Michael, M. (2012). Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 127–136.

Wang, C., Chen, W.-Y., Wu, Y., Saha, B., and Adl-Tabatabai, A.-R. (2007). Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 34–48.

Wu, P., Michael, M. M., von Praun, C., Nakaike, T., Bordawekar, R., Cain, H. W., Cascaval, C., Chatterjee, S., Chiras, S., Hou, R., Mergen, M., Shen, X., Spear, M. F., Wang, H. Y., and Wang, K. (2009). Compiler and runtime techniques for software transactional memory optimization. *Concurrency and Computation: Practice and Experience*, 21(1):7–23.

Yoo, R. M., Ni, Y., Welc, A., Saha, B., Adl-Tabatabai, A.-R., and Lee, H.-H. S. (2008). Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 265–274.

# Transactional Boosting no Glasgow Haskell Compiler

**Jonathas A. O. Conceição**[1] *, **André R. Du Bois**[1], **Rodrigo G. Ribeiro**[2]

[1] Programa de Pós Graduação em Computação/CDTec
Universidade Federal de Pelotas (UFPel)

{jadoliveira, dubois}@inf.ufpel.edu.br

[2]Programa de Pós-Graduação em Ciência da Computação
Universidade Federal de Ouro Preto (UFOP)

rodrigo@decsi.ufop.br

***Resumo.*** *Transactional Boosting é uma técnica que pode ser usada para transformar ações linearmente concorrentes em ações transacionalmente concorrentes, possibilitando assim sua utilização em blocos transacionais. Esta técnica pode ser utilizada para resolução de falsos conflitos, evitando assim a perda de desempenho de algumas aplicações. O objetivo deste trabalho é apresentar uma extensão do STM Haskell, bem como as modificações necessárias ao Run-Time System do compilador, para permitir o desenvolvimento de aplicações que utilizam Transactional Boosting, e assim apresentar a viabilidade desta técnica em Haskell.*

## 1. Introdução

*Software Transactional Memory* (STM) é uma alternativa de alto nível ao sistema de sincronização por *locks*. Nela todo acesso à memória compartilhada é agrupado como transações que podem executar de maneira concorrente. Se não houve conflito no acesso à memória compartilhada, ao fim da transação um *commit* é feito, tornando assim o conteúdo do endereço de memória público para o sistema. Caso ocorra algum conflito um *abort* é executado descartando qualquer alteração ao conteúdo da memória. Diferente da sincronização por *locks*, transações podem ser facilmente compostas e são livres de *deadlocks* [Harris et al. 2008].

Memórias Transacionais funcionam através da criação de blocos atômicos onde alterações de dados são registradas para detecção de conflitos. Um conflito ocorre quando duas ou mais transações acessão o mesmo endereço e pelo menos um dos acessos é de escrita. Entretanto, essa forma de detecção de conflitos pode, em alguns casos, gerar falsos conflitos levando a uma perda de desempenho. Um exemplo seria quando duas transações modificam partes diferentes de uma lista encadeada [Sulzmann et al. 2009, Herlihy and Koskinen 2008]. Embora essas ações não conflitem, o sistema detecta um conflito já que uma transação modifica uma área de memória lida por outra transação. Este tipo de detecção de conflito pode ter grande impacto na performance quando se utiliza certos tipos de estruturas encadeadas. Por outro lado, utilizando sincronização por *locks*, ou mesmo algoritmos *lock-free*, programadores experientes podem

*O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - CAPES/Brasil 88882.151433/2017-01 e da Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul - FAPERGS

alcançar um alto nível de concorrência ao custo de complexidade no código. *Transactional Boosting* [Herlihy and Koskinen 2008] pode ser aplicado para transformar objetos linearmente concorrentes em objetos transacionalmente concorrentes, oferecendo assim uma solução para esses falsos conflitos.

O objetivo deste trabalho é desenvolver uma biblioteca para *Transactional Boosting* em STM Haskell, permitindo assim a aplicação desta técnica de maneira nativa no *Glasgow Haskell Compiler* (GHC).

As contribuições deste artigo são:

- Desenvolvimento de uma nova primitiva que possibilita a aplicação da técnica de *Transactional Boosting* na biblioteca STM Haskell;
- Extensão do *RunTime System* do GHC para dar suporte a tal primitiva;
- Implementação de casos de uso da primitiva apresentada e análise do desempenho da mesma.

O artigo é organizado da seguinte forma: A Seção 2 descreve a biblioteca STM Haskell; na Seção 3 o conceito de *Transactional Boosting* é apresentado juntamente com a nova primitiva desenvolvida; a Seção 4 mostra três exemplos de aplicação da técnica; na Seção 5 as modificações necessárias no *RunTime System* do Haskell são descritas; a Seção 6 apresenta os resultados de alguns experimentos feitos para avaliar nossa implementação; na Seção 7 trabalhos relacionados são discutidos; por fim, na Seção 8 conclusões e trabalhos futuros são relatados.

## 2. STM Haskell

STM Haskell [Harris et al. 2008] é uma biblioteca do *Glasgow Haskell Compiler* que provê primitivas para o uso de memórias transacionais em Haskell. O programador define ações transacionais que podem ser combinadas para gerar novas transações como valores de primeira ordem. O sistema de tipos da linguagem só permite acesso à memória compartilhada dentro das transações e transações não podem ser executadas fora de uma chamada ao *atomically*, garantindo assim que a atomicidade (o efeito da transação se torna visível todo de uma vez) e isolamento (durante a execução, uma transação não é afetada por outra) são sempre mantidos.

A biblioteca define um conjunto de primitivas para utilização de Memórias Transacionais em Haskell (Figura 1). Nela o acesso a memória compartilhada é feito através de variáveis transacionais, as *TVar*s, que são variáveis acessíveis apenas dentro de transações. Ao fim da execução de um bloco transacional, um registro de acesso às *TVars* é analisado pelo *RunTime System* para determinar se a transação foi bem-sucedida ou não, para assim realizar o *commit* ou *abort* da transação.

Existem três primitivas para o uso de *TVars*: (1) **newTVar** é utilizada para criar uma *TVar* que pode conter valores de um tipo a qualquer; (2) **readTVar** retorna o conteúdo de uma *TVar*; e (3) **writeTVar** escreve um valor em uma *TVar*. As transações acontecem dentro da monada STM e essas ações podem ser compostas para gerar novas ações através dos operadores monádicos (bind (>>=), then (>>), e **return**), ou com a utilização da notação **do**.

O **retry** e o **orElse** são primitivas que controlam a execução do bloco. **retry** é utilizada para abortar uma transação e colocá-la em espera até que alguma de suas *TVars* seja

```
-- Execution control
atomically :: STM a -> IO ()
retry :: STM a
orElse :: STM a -> STM a -> STM a

-- Transactional Variables
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

**Figura 1. Interface do STM Haskell.**

alterada por outra transação. **orElse** é uma primitiva de composição alternativa, ela recebe duas ações transacionais e apenas uma será considerada; se a primeira ação chamar **retry** ela é abandonada, sem efeito, e a segunda ação transacional é executada; se a segunda ação também chamar **retry** todo o bloco é reexecutado.

## 3. Transactional Boosting em Haskell

*Transactional Boosting* [Herlihy and Koskinen 2008] é uma técnica proposta inicialmente no contexto de Programação Orientada a Objetos, como uma maneira de transformar objetos linearmente concorrentes em objetos transacionalmente concorrentes, permitindo assim sua utilização dentro de blocos atômicos. Nas transações, esses objetos são tratados como caixas-pretas e *handlers* para lidar com efetivações e cancelamentos são registrados no sistema transacional. Dessa forma, para que seja criado uma versão *boosted* de uma ação realizada em um objeto, é necessário que essa tenha um inverso, para que o sistema STM possa lidar com o cancelamento de uma transação. Além disso, apenas operações comutativas podem ser executadas concorrentemente.

Neste trabalho, foi adicionada uma nova primitiva ao STM Haskell que permite a chamada de funções não transacionais escritas em Haskell dentro de uma ação STM, além disso a primitiva permite a adição de ações que serão executadas no caso de cancelamento ou no caso de efetivação da transação. A função de *boost* tem o seguinte protótipo:

```
boost :: IO(Maybe a) -> (a -> IO ()) -> IO () -> STM a
```

Os argumentos são:

- Uma ação (do tipo **IO**(**Maybe** a)), que é a função original que vai ser executada. Quando a ação original é executada ela pode retornar um resultado do tipo a, ou se por algum motivo ela não pôde ser completada, e.g. um *lock* interno não pode ser adquirido, a função deve retornar **Nothing**.
- Uma ação de cancelamento (do tipo a **-> IO**()), usada para reverter a ação executada em caso de *abort*. Quando chamada, esta deve receber o valor retornado pela ação original para que seu efeito seja revertido.
- Uma ação de *commit* (do tipo **IO**()), que é usado para tornar público a ação feita pela versão *boosted* da função original.

Com isso *boost* retorna então uma nova ação STM que pode ser usada dentro dos blocos atômicos para executar a ação original.

| **Function Call** | **Inverse** |
|---|---|
| generateID | noop |

**Commutativity**

x <- generateID $\Leftrightarrow$ y <- generateID x $\neq$ y

x <- generateID $\not\Leftrightarrow$ y <- generateID x = y

**Figura 2. Especificação do gerador de identificadores únicos.**

## 4. Exemplos

Nesta Seção é apresentada a implementação em Haskell de três exemplos clássicos de *Transactional Boosting* [Herlihy and Koskinen 2008]. Estes exemplos são um gerador de identificadores únicos (Seção 4.1); Um Produtor e Consumidor (Seção 4.2) e Conjuntos (Seção 4.3).

### 4.1. Gerador de Identificadores Únicos

Um gerador de IDs únicos em STM pode ser problemático, sua implementação mais comum seria utilizando um contador compartilhado que é incrementado a cada chamada. Como diferentes transações estão acessando e incrementando um mesmo endereço de memória, o contador, o sistema de memória transacional da linguagem detectaria vários conflitos. Entretanto esses não são necessariamente conflitos. Desde que todos os retornos sejam diferentes não é necessário que os IDs sejam totalmente sequenciais.

Uma simples implementação *thread-safe* para fazer o gerador de IDs únicos seria utilizando uma instrução de *Compare-and-Swap* (CAS) disponível em diversas arquiteturas *multicore*. Haskell provê uma abstração chamada **IORef** para representar locais de memórias mutáveis. A biblioteca *atomic-primops* [Newton 2016] permite ao programador realizar operações CAS implementadas em Hardware com **IORef**s. Assim um gerador de IDs únicos pode ser implementado da seguinte forma:

```
type IDGer = IORef Int

newID :: IO IDGer
newID = newIORef 0

generateID :: IDGer -> IO Int
generateID idger = do v <- readIORef idger
  ok <- atomCAS idger v (v+1)
  if ok then return(v+1) else generateID idger
```

A função generateID recebe como argumento a referência ao contador compartilhado e aplica o CAS até conseguir incrementá-lo. Usando *Transactional Boosting* o gerador terá que seguir a especificação da Figura 2. Usando a primitiva `boost` uma versão transacional do gerador de IDs pode ser implementada da seguinte forma:

```
generateIDTB :: IDGer -> STM Int
generateIDTB idger = boost ac undo commit
  where
  ac = generateID idger >>= (\newID -> return(Just newID))
  undo _ = return()
```

| Function Call | Inverse |
|---|---|
| offer buff x | tryPopL buff |
| x <- take buff | pushR buff x |

**Commutativity**

offer buff x $\Leftrightarrow$ y <- take buff, buffer non-empty

offer buff x $\not\Leftrightarrow$ y <- take buff, otherwise

**Figura 3. Especificação do Produtor-Consumidor.**

```
commit = return()
```

O novo gerador de IDs é agora uma operação transacional e pode ser chamado livremente dentro de transações. Quando executado, este simplesmente utiliza a versão CAS do gerador para incrementar o contador. Em caso de *commit* ou *abort*, nada precisa ser feito (ver Figura 2), por isso as ações estão vazias.

### 4.2. Produtor e Consumidor

O Produtor-Consumidor se trata de um problema onde há uma sequência de dados a serem processados por *threads* que se comunicam através de um *buffer* compartilhado.

O *buffer* deve oferecer então duas funções: offer, utilizada para adicionar um valor ao *buffer* e take para retirar um valor do *buffer*. Para implementar o *buffer* utilizando *Transactional Boosting* foi utilizado um *double-ended queue thread-safe*, seguindo a especificação da Figura 3.

```
offer :: TBBuffer a -> a -> STM ()
offer (TBBuffer c ioref) v = boost ac undo commit
  where
  ac = do pushL c v
    return (Just ())
  undo _ = do mv <- tryPopL c
    case mv of
      Just v -> return ()
  commit = do v <- readIORef ioref
    ok <- atomCAS ioref v (v+1)
    if ok then return () else commit
```

O **TBBuffer** se trata de um *dequeue* e um **IORef** que conta o tamanho do *buffer*. A função offer usa **pushL** para adicionar um valor ao *queue*. Se a transação abortar o valor colocado no *buffer* deve ser removido utilizando o **tryPopL**. Em caso de *commit* só resta atualizar o tamanho da fila para tornar o novo elemento visível à *thread* consumidora. A função take utiliza **tryPopR** para consumir os dados do *buffer*:

```
take :: TBBuffer a -> STM a
readFifo (TBBuffer c ioref) = atomically (boost ac undo commit)
  where
  ac = do
    size<-readIORef ioref
    if size == 0 then return Nothing
      else do mv <- tryPopR c
```

```
        case mv of
          Just v -> return (Just v)
  undo v = pushR c v
  commit = do v <- readIORef ioref
    ok <- atomCAS ioref v (v-1)
    if ok then return () else commit
```

Se não há elementos o suficiente no *buffer* o **Nothing** resultante irá disparar um *abort* na transação. Caso contrário a função irá decrementar o contador do *buffer* e consumir o valor. A ação de *undo* deverá incrementar o contador e devolver o valor ao *buffer*.

### 4.3. Conjuntos

Uma implementação de conjuntos normalmente oferece três funções, *add*, *remove* e *contains*.

| **Function Call** | **Inverse** |
|---|---|
| add set x / False | noop |
| add set x / True | remove set x / True |
| remove set x / False | noop |
| remove set x / True | add set x / True |
| contains set x / _ | noop |

**Commutativity**

add set x / _ $\Leftrightarrow$ add set y / _, x $\neq$ y

remove set x / _ $\Leftrightarrow$ remove set y / _, x $\neq$ y

add set x / _ $\Leftrightarrow$ remove set y / _, x $\neq$ y

add set x / False $\Leftrightarrow$ remove set x / False $\Leftrightarrow$ contains set x / _

**Figura 4. Especificação da estrutura de conjuntos.**

Para a versão *boosted* de conjuntos apresentada aqui foi utilizada uma lista encadeada *thread safe*, descrita em [Sulzmann et al. 2009]. Na implementação é importante garantir que se uma transação está trabalhando num elemento, nenhuma outra transação vai utilizar o mesmo elemento (vide Figura 4). Isso pode ser alcançado utilizando *key-based locking*, implementado utilizando uma tabela hash para associar um *lock* para cada elemento do conjunto. Vários modelos de tabela hash *thread safe* em Haskell foram apresentados em [Duarte et al. 2016], dentre eles, o algoritimo de *lock fino* foi utilizado para esta implementação de conjuntos.

Para adicionar um elemento ao conjunto deve-se adquirir o *lock* associado ao elemento e então inserí-lo na lista encadeada. Como a lista pode conter elementos duplicados é necessário verificar se o elemento já não está contido antes de inserí-lo. Se um elemento foi inserido e a transação abortar, o elemento deve ser removido e o *lock* liberado. Caso a transação termine sem conflitos é necessário apenas liberar o *lock*:

```
add :: IntSet -> Int -> STM Bool
add (Set alock list) element = boost ac undo commit
  where
    ac = do ok <- AbstractLock.lock alock element
      case ok of
        True -> do found <- List.find list element
```

```
        if found then return (Just False)
          else do List.addToTail list element
            return (Just True)
      False -> do return Nothing
  undo v = do case v of
      True -> do List.delete list element
        AbstractLock.unlock alock element
        return()
      False -> AbstractLock.unlock alock element >> return()
  commit = AbstractLock.unlock alock element >> return ()
```

Para remover um elemento é preciso adquirir o *lock* associado e então deletar o elemento da lista. Para reverter um *remove* o elemento deve ser devolvido ao conjunto e o *lock* liberado. O *commit* assim como antes precisa apenas liberar o *lock*.

```
remove :: IntSet -> Int -> STM Bool
remove (Set alock list) element = boost ac undo commit
  where
    ac = do ok <- lock alock element
      case ok of
        True -> do v <- List.delete list element
          return (Just v)
        False -> return Nothing
    undo ok = do case ok of
        True -> do List.addToTail list element
          AbstractLock.unlock alock element
          return ()
        False-> AbstractLock.unlock alock element >> return ()
    commit = AbstractLock.unlock alock element >> return ()
```

Por fim o *contains* precisa apenas adquirir o *lock* do elemento e então conferir se ele está na lista. Tanto para o *commit* como para o *abort* o *contains* precisa apenas liberar o *lock* adquirido. Por sua simplicidade o *contains* teve o seu código omitido neste artigo.

## 5. Implementação

Para a proposta de *Transactional Boosting* em [Herlihy and Koskinen 2008] é necessário que o sistema de memória transacional da linguagem permita a definição de *handlers* para quando uma transação realizar o *abort* ou *commit*, entretanto essa não é uma funcionalidade oferecida pelo *STM Haskell*. Para este trabalho uma extensão do *STM Haskell* e do RTS do GHC foi feita para que o compilador oferecesse suporte nativo para *Transactional Boosting*.

A implementação da primitiva se divide em duas partes. Uma interface Haskell em alto nível, que provê a função `boost`, e uma camada principal no *RunTime System* (RTS) escrita em C e em Cmm (um *assembly* de alto nível utilizado na implementação do RTS [Ramsey et al. 2005]).

### 5.1. Implementação da primitiva `boost`

A implementação em Haskell da primitiva é responsável por lidar com o sistema de tipos da linguagem, executar a ação original e passar dados do alto nível para o baixo nível:

```
boost :: IO(Maybe a) -> (a -> IO ()) -> IO () -> STM a
boost iomac undo commit = STM (\s -> expression s)
  where
    expression = unIO $ iomac >>= \mac -> case mac of
      Just ac -> IO (\s -> (boost# (return ac) undo commit) s)
      Nothing -> IO (\s -> (abort# s))
```

A função começa executando a ação à ser aplicado o *boost* (iomac). Se o resultado for um **Just** ac a ação foi bem sucedida, neste caso a primitiva **boost#** é chamada para registrar no RTS ações para o *commit* e *undo* da ação executada. Caso o resultado seja um **Nothing** a ação não pode ser executada agora e a transação deve recomeçar. Ambas as funções **boost#** e **abort#** são *Primitive Operations* e serão explicadas em mais detalhe na Seção 5.2, o caractere # ao fim do nome da função indica uma chamada ao RTS no GHC. A seguir é apresentado o protótipo das *Primitive Operations* em Haskell:

```
boost# :: IO a -> (a -> IO ()) -> IO () -> STM a
abort# :: STM a
```

### 5.2. Implementação no RunTime System

O *RunTime System* (RTS) é uma biblioteca escrita predominantemente em C que é ligada a qualquer programa em Haskell compilado com o GHC. Ele provê suporte e infraestrutura para funcionalidades como *garbage collector*, transações, exceções, escalonamento, controle de concorrência, entre outras. O RTS pode ser visto como três grandes subsistemas: Armazenamento, responsável pelo layout de memória e *garbage collector*; Execução, responsável pela execução de código Haskell; O Escalonador, que gerencia threads e dá suporte *multicore*. As modificações feitas no RTS para este trabalho aconteceram principalmente nas partes de Armazenamento e Execução.

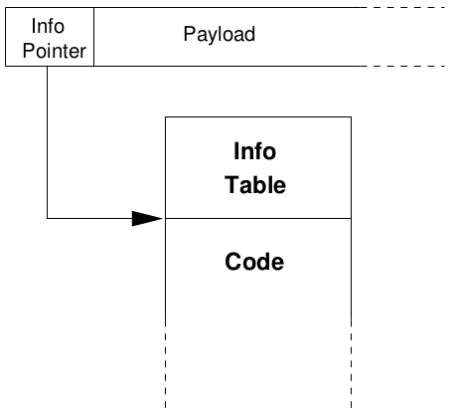



**Figura 5. *Layout* de um *Heap Object* [Marlow and Peyton Jones 1998].**

*Heap Objects*, são um aspecto central do armazenamento das funções em execução no RTS. Estas são estruturas de dados escritas em C e todo *Heap Object* segue o *layout* da Figura 5. A primeira parte desses objetos é chamado de *Info Pointer*, que aponta para o *Info Table* da estrutura, a segunda é o *Payload* onde ficam os dados carregados pelo *Heap Object* [Marlow and Peyton Jones 1998]. A *Info Table* contém informações sobre o tipo de estrutura, informação essa utilizada principalmente pelo *garbage collector*, e também o código responsável por avaliar o objeto.

Um tipo importante de *Heap Object* é o *Thread State Object* (TSO). Ele representa o estado atual de uma thread incluindo seu *stack* de execução. O *stack* consiste de uma sequência de *stack frames* onde cada *frame* corresponde a um *Heap Object*.

Para dar suporte à *Transactional Boosting* um novo *Heap Object* (StgBoostSTMFrame) foi criado para armazenar o resultado da função de **boost**, bem como a ação de *abort*. O *Heap Object* criado possui duas referências em seus campos: (1) ação de *abort*; (2) resultado da ação de boost executada, que é usado como argumento da ação de *abort*. O objeto tem o código a seguir:

```
typedef struct {
  StgHeader    header;
  StgClosure *tbAbort;
  StgClosure *tbResult;
} StgBoostSTMFrame;
```

Na parte de Execução temos as chamadas *Primitive Operations* (*PrimOps*), estas são operações que por alguma impossibilidade ou por uma questão de desempenho são implementadas diretamente no RTS, e este é o contexto onde a maioria das funções do STM Haskell se encontram. As *PrimOps* são escritas em Cmm, um assembly de alto nível que é compilado dentro do próprio GHC. Apenas códigos em Cmm pode manipular diretamente o *stack* do TSO ao qual pertencem e invocar novas execuções de transações.

A primitiva **boost#** é responsável por instanciar um novo *StgBoostSTMFrame* com suas respectivas referências; o *StgBoostSTMFrame* é então colocado na *stack* do TSO atual. As ações de commit, são armazenadas em uma lista associada ao TSO, juntamente com o estado da transação (conjuntos de leitura e escrita). Ao fim da transação, se ela for bem-sucedida, as ações de commit são executadas. O **abort#** por sua vez, quando chamado percorre o *stack* do TSO realizando as ações de *abort* necessárias para cada tipo de *stack frame* encontrado e então recomeça a transação. Três tipos de *frames* podem ser encontrados dentro dos blocos transacionais, são eles:

- StgCatchRetryFrame, o *frame* associado ao **orElse**. Neste caso somente as ações de abort do primeiro ramo serão executadas e a execução continua no segundo ramo do `orElse`
- StgBoostSTMFrame, o *frame* associado à ação de *Boost*. Quando encontrado sua ação de *abort* associada é executada.
- StgAtomicallyFrame, o *frame* associado ao **atomically**. Encontrá-lo no *stack* indica que a se retornou ao início da transação. Neste caso o registro de acesso à memória atual é descartado e a transação reexecutada do início.

A primitiva de **retry#** (*PrimOp* utilizado pelo **retry**) funciona de maneira semelhante ao **abort#**. Quando executado ele também percorre o *stack* associado a seu TSO para tratar dos casos específicos até chegar no StgAtomicallyFrame.

## 6. Resultados e Experimentos

Os experimentos foram executados numa máquina com processador Intel Core i7, frequência de 3.40GHz, 4 cores físicos e 4 lógicos, 8GiB de memória RAM. O sistema operacional usado foi o Ubuntu 14.04, a versão do GHC foi a 7.10.3.

A Figura 6 apresenta os resultados para três implementações do gerador de identificadores únicos: IDSTM que é uma implementação utilizando o STM Haskell puro; IDCAS que utiliza CAS para incrementar o contador; IDTB que que utiliza *Transactional Boosting*. Foram executados 10 milhões de chamadas de incremento ao identificador, dividindo igualmente as operações entre as threads disponíveis. Trinta execuções foram feitas, e o gráfico apresenta as médias de cada execução com o tempo em escala logarítmica. Como esperado, a versão usando *Transactional Boosting* adiciona um certo *overhead* em comparação com a versão usando apenas CAS, porém se comparado com o desempenho da ação transacional pura, a versão *boosted* se mostra uma alternativa muito mais eficiente.

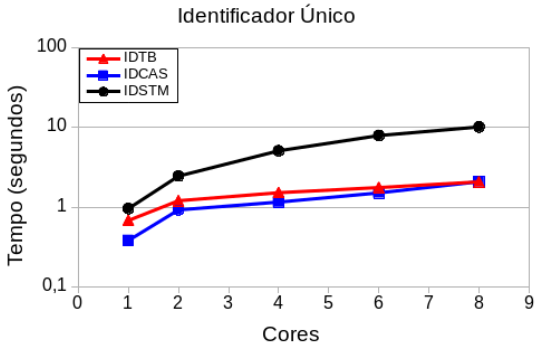



**Figura 6. Tempo de execução do Identificar Único.**

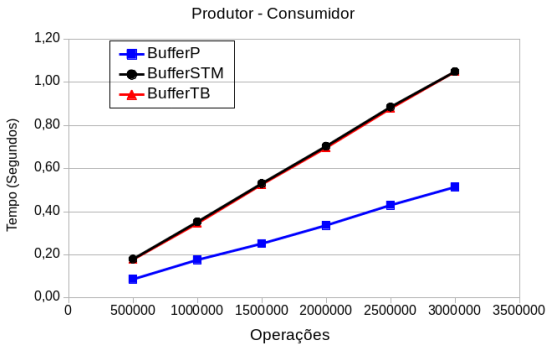



**Figura 7. Tempo de execução do Produtor-Consumidor.**

A Figura 7 por sua vez apresenta os experimentos com o Produtor e Consumidor. Nas execuções foram criadas duas threads, um produtor e um consumidor, para compartilhar o *buffer*. Três implementações são apresentadas: BufferSTM que utiliza apenas a biblioteca STM Haskell; BufferP é a implementação do *dequeue thread safe* e linearmente concorrente; BufferTB é a versão usando *Transactional Boosting*. Neste exemplo o *overhead* existente sobre a aplicação do *boost* torna o BufferTB equivalente ao BufferSTM em termos de tempo de execução. A biblioteca TChan STM, utilizada na versão puramente transacional, utiliza listas duplamente encadeadas, assim, com apenas um produtor e um consumidor, poucos falsos conflitos ocorrem; neste caso pode-se observar que o *overhead* da utilização do *boost* é equivalente ao *overhead* da própria camada STM, sendo o *Transactional Boosting* levemente mais rápido.

Para avaliação de um exemplo com grande quantidade de falsos conflitos, assim como do uso do **retry** e **orElse**, foram implementados dois exemplos que utilizam a estrutura de conjuntos descrita na Seção 4.3. Para ambos os exemplos, listas de 2000 operações foram geradas aleatoriamente para serem aplicadas sobre conjuntos inicializados com 2000 elementos.

No primeiro experimento, dois conjuntos diferentes foram inicializados, e todas as *threads* recebiam referências para ambos os conjuntos. Cada *thread* tenta realizar a operação no primeiro conjunto, caso a operação falhasse, e.g., tentar remover um elemento que não esta no conjunto, um **retry** era chamado e a *thread* tentava a mesma operação no segundo conjunto; se a operação no segundo conjunto também chamasse um **retry** todo o bloco abortava e a *thread* procurava uma nova operação. O gráfico da Figura 8 mostra a

média de 30 execuções em escala logarítmica.

No segundo experimento, dois tipos de listas eram gerados em execuções separadas: Lista de leitura, contendo 40% de operações de add e remove mais 60% de operações de contains; Lista de Escrita contendo 75% de operações de add e remove mais 25% de contains. No gráfico da Figura 9 mostra a média de 30 execuções em escala logarítmica.

Pelas Figuras 8 e 9 pode-se observar que a utilização do *Transactional Boosting* resulta numa estrutura de conjuntos de desempenho bem superior em comparação à alternativa feita puramente com o STM Haskell. Isso acontece pois a detecção de conflitos feito pelo sistema de Memórias Transacionais gera uma excessiva quantidade de falsos conflitos lidando com estruturas encadeadas.

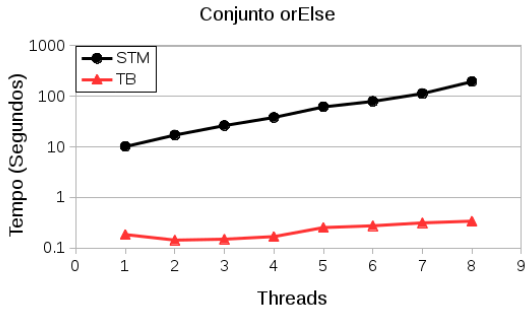



**Figura 8. Tempo médio de execução de 2000 operações.**

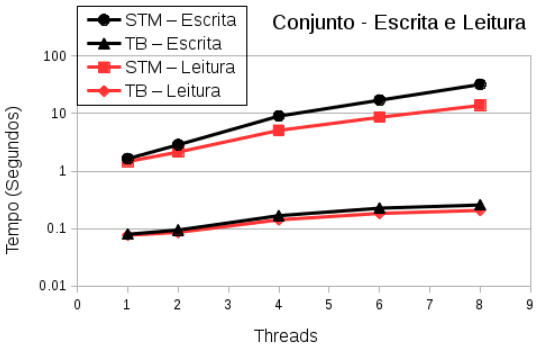



**Figura 9. Tempo médio de execução de 2000 operações.**

## 7. Trabalhos Relacionados

Outros trabalhos oferecem extensões para o STM Haskell que abordam o problema de perda de desempenho em casos de falsos conflitos. O **unreadTVar** [Sönmez et al. 2007] pode ser utilizado para melhorar o tempo de execução e uso de memória quando se atravessa uma estrutura transacional de dados encadeados. A primitiva pode ser usada para retirar do registro de leitura valores que podem gerar falsos conflitos. Similarmente em [Sulzmann et al. 2009] a primitiva **readTVarIO** é utilizada como uma maneira de ler *TVars* para percorrer uma estrutura encadeada sem que ela seja adicionada ao registro de leitura da transação.

Em [Harris 2007] os autores apresentam *Abstract Nested Transaction*, que também oferece uma solução para problemas de desempenho relacionados aos falsos conflitos em STM. Nele acessos à memória são armazenado num registro próprio para a detecção de conflitos. E em caso de conflito, inicialmente apenas a expressão que acessa dados conflitantes é reavaliada para checar se o resultado da expressão foi alterado desde a sua primeira execução, e apenas se houver uma alteração toda a transação é reexecutada.

Estes métodos acima apresentam maneiras de implementar novos tipos de dados com o objetivo de evitar falsos conflitos ou antecipar sua detecção e tratamento. O *Transactional Boosting*, por outro lado, permite a composição de estruturas linearmente concorrentes eficientes já existentes às ações transacionais. A primitiva de *Transactional Boosting* aqui apresentada foi inicialmente proposta em [Du Bois et al. 2014]. Porém a

implementação apresentada no artigo era toda feita em Haskell usando um sistema transacional também totalmente escrito em Haskell. Os resultados positivos apresentados por esse protótipo levaram os autores a buscar a implementação com suporte do RTS apresentada aqui.

## 8. Conclusões e Trabalhos Futuros

Neste artigo foi apresentada uma extensão do STM Haskell e do RTS do GHC que permite a aplicação da técnica de *Transactional Boosting* em programas escritos na linguagem Haskell. Além disso, foram apresentados três estudos de caso e seus respectivos desempenhos que validam a extensão apresentada.

O sistema aqui descrito oferece uma maneira simples de transformar objetos linearmente concorrentes em objetos transacionalmente concorrentes que podem ser executados em transações. *Transactional Boosting* é uma técnica de baixo nível para controle de concorrência e pode resultar em problemas como *deadlocks*. Entretanto se empregada por programadores experientes pode ser usada no desenvolver bibliotecas transacionais de alto desempenho. Para trabalhos futuros visamos apresentar uma semântica formal para uso da primitiva.

## Referências

Du Bois, A., Pilla, M., and Duarte, R. (2014). Transactional boosting for haskell. In Quintão Pereira, F., editor, *Programming Languages*, volume 8771 of *Lecture Notes in Computer Science*, pages 145–159. Springer International Publishing.

Duarte, R. M., Du Bois, A. R., Pilla, M. L., and Reiser, R. H. S. (2016). Comparando o desempenho de implementações de tabelas hash concorrentes em haskell. *Revista de Informática Teórica e Aplicada*, 23(2):193–209.

Harris, T. (2007). Abstract nested transactions. In *TRANSACT 2007*.

Harris, T., Marlow, S., Jones, S. P., and Herlihy, M. (2008). Composable memory transactions. *Commun. ACM*, 51(8):91–100.

Herlihy, M. and Koskinen, E. (2008). Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 207–216, New York, NY, USA. ACM.

Marlow, S. and Peyton Jones, S. (1998). The new ghc/hugs runtime system.

Newton, R. R. (2016). Atomic-primops: A safe approach to cas and other atomic ops in haskell.

Ramsey, N., Jones, S. P., and Lindig, C. (2005). *The C– Language Specification Version 2.0*.

Sönmez, N., Perfumo, C., Stipic, S., Cristal, A., Unsal, O. S., and Valero, M. (2007). unreadtvar: Extending haskell software transactional memory for performance. *Trends in Functional Programming*, 8:89–114.

Sulzmann, M., Lam, E. S., and Marlow, S. (2009). Comparing the performance of concurrent linked-list implementations in haskell. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 37–46. ACM.

# Evaluation of Timing Side-channel Leakage on a Multiple-target Dynamic Binary Translator

**Otávio O. Napoli[1], Vanderson Martins do Rosario[1], Diego F. Aranha[1] and Edson Borin[1]**

[1]Institute of Computing – University of Campinas (Unicamp)
Av. Albert Einstein, 1251 - 13.083-852, Campinas – SP – Brazil

{otavio.napoli, vanderson.rosario, edson, dfaranha}@ic.unicamp.br

***Abstract.*** *Timing side-channel attacks are an important issue for cryptographic algorithms. If the execution time of an implementation depends on secret information, an adversary may recover the latter through measuring the former. Different approaches have emerged recently to exploit information leakage on cryptographic implementations and to protect them against these attacks. However, little has been said about ISA emulation and its impact on timing attacks. In this paper, we investigate the impact of an emulator (dynamic binary translator), OI-DBT, using different Region Formation Techniques (RFTs) on constant-time and non-constant-time implementations of cryptographic algorithms. We show that emulation can have a significant impact on secret leakages, even mitigating them in some cases. Moreover, our results indicate that the choice of RFT heuristic by the emulator does have an impact on these leakages.*

## 1. Introduction

Cryptography algorithms are designed to be secure against analytic attacks, but valuable information can still be extracted by peculiarities in their implementation and execution. This undesired leakage of information (like variations in execution time, system power consumption, branch prediction behavior) is statistically exploited by side-channel attacks aiming to infer secret information used by the algorithms [Kang et al. 2016]. Among the different types of side-channel attacks, timing attacks are a class of side-channel attacks that try to infer secrets based on the execution time behavior of the algorithm. Roughly speaking, if this behavior depends on a secret value, information can be leaked. These attacks against modern cryptography algorithms were demonstrated feasible even when including network noise in remote attacks [Brumley and Boneh 2005]. Since then, they have become more relevant to various applications in different scenarios, such as IoT, cloud computing, among others.

Several implementation techniques to achieve constant-time execution were proposed to protect against timing attacks [Käsper and Schwabe 2009], and have become widely used. It is possible to find these techniques in various libraries, such as OpenSSL (openssl.org) and BearSSL (bearssl.org). The rules for constant-time implementation are very conservative, and consist of avoiding: branching based on secret data, variable-latency instructions, and table look-ups with secret indexes, among others. Thus, the

Acknowledgement: the authors would like to thank CAPES (PROEX - 0487/2018), CNPq, FAPESP, Microsoft and Intel for their financial support and the Multidisciplinary High Performance Computing Lab (LMCAD) for its infrastructure and technical support.

binary code must be carefully generated and inspected to guarantee constant-time execution, which can be difficult for larger cryptographic libraries. Furthermore, a compiler can remove some of these modifications and reintroduce leakage points to the code, like the strength-reduction optimization can inserts branches that depends on the secret value [Cleemput et al. 2012]. This means that the final binary needs to be carefully reexamined every time a change is made and the code is recompiled, complicating the task even more. For this reason, many automatic countermeasures were introduced in the literature [Cauligi et al. 2017, De Mulder et al. 2018, Chen and Venkataramani 2014]. Some of them make use of (difficult to create) computation models to remove classes of leakage with soundness while others remove the complexity of setting up the computational model but only mitigate the leakage with no guarantee of removing it. In this scenario, dynamic execution systems such as emulators may also have an impact on these implementations, either by adding noise to mitigate leakages or by applying optimizations or code transformations that could create them. Anyway, little has been studied about timing leakage on emulators and so, in this work, we investigate the impact of multiple-target emulation in constant-time/leakage property of cryptographic implementations using the Dudect tool presented by Reparaz et. al [Reparaz et al. 2017].

An emulator is a piece of software that enables the execution of a binary compiled to one ISA (guest) into another (host) [Smith and Nair 2005]. The guest ISA can be a real one, like emulating an ARM binary into an x86 processor, or a virtual one (has no real hardware implementation) like Java Bytecode. One of the fastest and most common ways to implement an emulator is using dynamic compilation (a.k.a. Dynamic Binary Translation (DBT) or JIT Compilation), a technique which starts by interpreting code, then selects hot regions with a heuristic (Region Formation Techniques, RFTs) and lastly compiles/translates these regions to the host ISA for faster (native) execution. We executed our experiments using a Cross-ISA Multiple-target RISC-V/OpenISA DBT named OI-DBT varying between five RFTs techniques (NET, NET-R, NETPlus, NETPlus-e-r and MRET2) to test the timing leakage during emulation (compared to native execution timing leakage) of three algorithms (a memory comparison routine, AES block cipher, and Elliptic-curve Curve25519-donna algorithm) each one with a non-constant-time and a constant-time implementation. Finally, the main results and contributions of this paper are:

- We show that an emulator can interfere with the time-leakage property of some implementations, i.e. mitigating it. Moreover, although we did not see in our experiments the opposite result (an emulator adding leakage to a constant-time implementation), we argue that this is also possible to happen.
- We tested the impact of different RFTs during emulation and showed that they do have an influence on the amount of timing leakage.

This article is organized as follow: Section 2 presents a background in DBT, side-channel attacking and the OI-DBT infrastructure; Section 3 presents our timing leakage model; Sections 4 and 5 describe our methodology and results; and, finally, Sections 6 and 7 present the related works and our conclusion.

## 2. Background

This section aims to introduce some key concepts about DBT and information leakage through execution time. First, we introduce the concept of emulation and its implemen-

tation techniques (2.1), including RFTs (2.1.1), then we describe OI-DBT (2.2), which was the DBT used in our experiments. To the end, we introduce the main kinds of timing leakage (2.3) that are present in the literature and were tested in our experiments.

## 2.1. Emulation and DBT

Emulators allow the execution of binaries compiled to one ISA on another, and it has been useful in a handful of applications such as Virtual Machines (VMs), support of legacy code, simulators and others [Smith and Nair 2005]. The two main ways of implementing an emulator are by interpretation or by translation. The former is the slowest technique, but the most portable. The latter, on the other hand, is the one that results in faster emulators, but it is less portable and far more complicated to implement. The translation of an ISA can be done either statically, by translating the whole binary beforehand (Static Binary Translation, SBT), or dynamically, translating hot regions of code from the binary while interpreting it (Dynamic Binary Translation, DBT). Usually, SBT is simpler to implement than DBT, but given binary properties such as code discovery problem, self-modifying code and indirect branches, it cannot translate all programs, thus, DBT is more common in real scenarios.

DBT engines usually start by interpreting the binary code and only after some execution start translating code. While interpreting, they also collect execution frequency information which is used by heuristics (Region Formation Techniques, RFTs), further explained in Section 2.1.1, to detect regions of code that form a cycle or have a high probability of being executed many times in the future. Once selected, these regions of code are sent to a compiler to produce native code (in the target ISA) mimicking the behavior of the guest ISA region. These translated regions are put in a cache, known as Translated Code Cache (TCC), and every time the emulator needs to execute one instruction that is in the start address of one of the TCC regions, the emulator jumps to the native code in the TCC, executing the translated code. Translation (or compilation) in a DBT happens together with binary emulation and so, its execution time impacts directly to the final emulation time. However, as the execution of translated code is faster than interpretation, if a region is executed enough times, the translation cost is paid off by the speedup from using the translated code instead of interpretation.

As most of the emulation time is spent executing translated code, quality of region translation is extremely important for the final emulation performance [Hong et al. 2012] and RFTs impact directly in this quality because they are responsible for defining the compilation unit of a DBT engine. For instance, in OI-DBT, RFTs that select larger regions have a better performance as they open opportunities for more optimization and, most importantly, reduce the transitions between regions. In multiple-target DBT engines, such as OI-DBT and HQEMU [Hong et al. 2012], transitions are expensive because it is not trivial to map register banks between all pairs of architectures they support and then the DBT needs to apply register allocation separately for each region, creating different mappings and forcing each one of the regions to load and save the used guest registers context when entering and exiting (see Figure 1). Thus, multiple-target DBT engines add several extra store and loads which are executed every time a region-to-region (as arrows (c), (d) and (e) in Figure1)) or a region-to/from-interpreter (as arrows (a), (b)) transition happens.

The transition cost affects the overall execution and is not associated with a spe-

cific information. For constant time implementations this cost could add a random noise into the code execution. In Figure 2, for example, we have a memory access in OpenISA (represented by OI_array[]) that is indexed by the secret. After translating this region of code, not only the memory access is translated to x86, but also new stores and loads are inserted to load and save the guest register context, adding a random noise to the execution time and changing the memory access pattern. Moreover, as different RFTs have a different level of code fragmentation (more or fewer regions and transitions), we believe that they would impact in the information leakage level and this was exactly what we end up observing in our results. In fact, in some cases, these extra loads and stores changed the cache and pipeline behavior in such a way that completely mitigates the information leakage to a point of becoming undetectable in our experiments. This happened in test cases of leakage with varying-delay instructions and memory accesses depending on the secret.

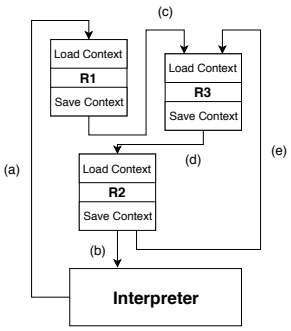



Figure 1. Guest machine context load and store when transiting between regions and interpreter.

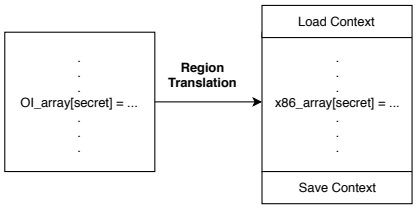



Figure 2. Extra overhead and leakage noise (context load and save) inserted by multiple-target DBT engines.

### 2.1.1. Region Formation Techniques (RFTs)

A DBT engine needs to decide which regions of code it is going to translate and optimize. This is a responsibility of the RFT which determines when to start recording a region, what to record and when to stop and send it to be translated/compiled. Different techniques have been proposed in the literature to address this challenge.

**NET [Duesterwald and Bala 2000]** creates super-blocks composed of instructions that are executed in sequence. It considers that every target of backward branches or super-blocks exit is a candidate for starting a region. A region starts to be recorded when the execution of one these points passes an established execution threshold. When recording, every instruction emulated is added to the new region and it only stops when a backward branch is executed or another region entrance is found. **MRET2 [Wang et al. 2007]** executes NET two times and select the code from the intersection of the result of the two executions. The goal is to reduce tail-duplication. **NET-R [Hong et al. 2012]** is a modified version of NET, a relaxation of it. NET-R does not end recording a region when a backward branch is found, instead, it ends when a cycle is found (repeated instruction address), creating larger regions. **NETPlus [Davis and Hazelwood 2011]** first runs NET and then instead of just finishing the super-block formation, also runs a forward search looking for paths which could extend the region. The search looks for paths which ex-

its the NET region and which returns to its entrance with less than a given number of steps (branches). This technique tends to select larger regions of code and reduce region fragmentation. **NETPlus-e-r [Hong et al. 2012]** is an extended and relaxed version of NETPlus. It uses NET-R to form regions and when expanding, it does not only include paths which return to the entrance of the region but all paths which return to any part of the region.

For multi-target DBT, reducing the number of transitions between regions is essential to achieve good performance (because of the need to save and restore register context). Techniques such as NETPlus-e-r and NET-R have demonstrated to have the least amount of transitions and the best performance with this kind of DBT engines [Hong et al. 2012].

### 2.2. OI-DBT

OI-DBT[1] is a multiple-source and multiple-target DBT engine based on LLVM 7.0. It supports emulating RISC-V and OpenISA binaries in both x86/ARM (32 and 64 bits) processors and it is also able to change its RFT dynamically supporting all aforementioned RFTs. In our experiments, we used and tested all these RFTs, however, in spite of having support for RISC-V and OpenISA, the RISC-V front-end is still under development and the OpenISA front-end is far more mature, so we only used the OpenISA front-end. Moreover, for the simplification of our experiments, we solely executed tests in an x86 processor, therefore, emulating OpenISA in an Intel x86 processor.

As can be seen in Figure 3, the OI-DBT architecture is composed of two major components: one that interprets, profiles and executes dynamically compiled RISC-V/OpenISA code and another that dynamically compiles selected RISC-V/OpenISA regions to native architecture binary, using the LLVM On-Demand ORC JIT infrastructure. Each one of these components runs in a different thread and communicates using a compilation-wait queue in a producer-consumer manner. Thus, translation and emulation are done concurrently.

OI-DBT starts by emulating code with an interpreter that also profiles branches and calls and the collected profile information is used to feed the dynamically selected RFT algorithm. From time to time, the RFT selects potential hot regions of code and adds them to the compilation wait-queue. Every time the wait-queue is not empty the compilation thread wakes up, removes one region of code from the queue and starts emitting an LLVM IR with an equivalent semantic. After finishing emitting the IR, the result is sent to the ORC JIT Compiler which optimizes and compiles the IR to the host ISA, in this case, x86. The generated native code is then put into the TCC and every time a branch or call is going to be emulated by the interpreter, it checks if the target is not in the cache and if it is, it jumps to its respective translated/native code. With this infrastructure, OI-DBT is able to achieve better performance then DBTs such as QEMU.

### 2.3. Timing Leakage

Timing leakage happens when the execution time of a program or the emulation of a program depends on secret information. In practice, if there is a data dependence between a secret and the execution time, an attacker executing a program, even remotely, with different inputs and measuring its execution time, can infer secret information from the

[1]URL to be released after publication

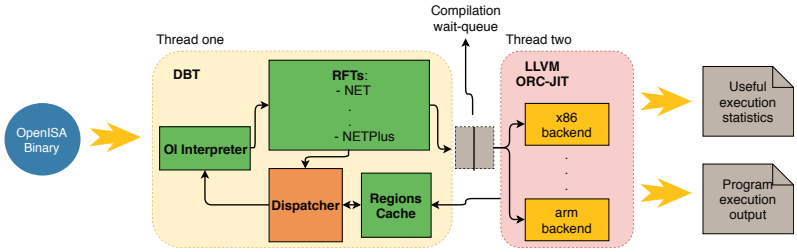



Figure 3. The OpenISA Dynamic Binary Translator (OI-DBT) architecture.

program using statistical methods. The three most common implementation/architecture characteristics which end up creating a dependence between data and time are: (1) having control flow depending on secret information, which will lead to different executions paths with a different number of instructions or memory pattern access when using different secrets; (2) changing the memory access pattern depending on the secret information, as it could lead to different cache performance, for instance, indexing a array access with secret data; (3) manipulating secret information with processor instructions that vary their execution time depending on the processed data.

## 3. Timing Leakage Detection Model

Our leakage detection model was implemented inside OI-DBT based on the Dudect tool from Reparaz et. al (2017). The tool uses a Test Vector Leakage Assessment (TVLA) methodology which is commonly used to assess the security of a system against side-channel attacks. This methodology consists in maintaining two vectors with executions times from two different classes of inputs and then comparing the two vectors using some statistical test to evaluate if they represent the same measurement of time. If not, both vectors have a different execution time for two different classes of input and, therefore, the implementation is not constant time.

For each test, what we will call an execution, a cryptography key is randomly chosen and $N$ inputs are randomly generated with an uniform distribution of two classes of inputs. The first class, $C_1$, consists in only one random plain text which is repeated among all $C_1$ inputs. The second class, $C_2$, consists of different random plain texts, one for each input of its class. In other words, $N$ inputs are generated with near to half of them fixed ($C_1$) and another half of them varying. We measured the time to execute each one of these inputs and inserted each measurement into its respective vector depending on its class. Finally, we run a Welch's $t$-test [Welch 1947] to infer if both vectors are measuring the same execution time. As statistical tests such as Welch's $t$-test can have their results accumulated, we can run this test (of $N$ inputs) $E$ times (which we call executions), each time cleaning the two time vectors, generating a new cryptography key and a new fixed plain text for $C_1$ and running the $t$-test to accumulate knowledge about the algorithm being constant or not. This fixed-vs-random kind of test is very popular in the literature and is considered one of the most powerful schemes for detecting timing leakages [Standaert 2017]. For the three algorithms tested in our experiments, we used different values for $N$ and $E$ based on their execution time and leakage detection speed. For Memory Comparison we used 30000 executions, for AES, 3000, and for Curve25519 we used 400. Now for the number of inputs, $N$, for memory comparison, we used 1000, for AES, 10000, and for Curve25519, 10000. The input size can make the hypothesis test

converge faster and these values were sufficient to analyze the behaviour of constant-time implementations.

Moreover, after every execution and before running the $t$-test, a post processing is applied in the two vectors, removing some time outliers. Timing distributions are skewed towards larger execution times. This may be caused by measurement artifacts such as the main process being interrupted by the OS or other extraneous activities. Thus, we removed any execution time which exceeds the average by 100%. Furthermore, both the generation of all inputs, the time collection, and the statistical test were implemented together with the cryptographic algorithm and compiled into one unique OpenISA binary, allowing us to discount from the execution time the time for starting the emulator and the time to construct all random inputs. To summarize, we only measure the time to emulate the algorithm for each input and remove everything else.

## 4. Experimental Setup

All experiments were executed in a system running the GNU/Linux Ubuntu 16.04.4 LTS (kernel 4.13.0) operating system and a 4-core 64-bit processor Intel(R) Core(TM) i7-7700 CPU 3.60GHz with 32GB of RAM. Experiments with OpenISA were compiled using Clang 3.7 and emulated using OI-DBT 0.1. Finally, the x86 native binaries (the ones not emulated), referenced during the experimental result section as 'native', were compiled using Clang 6.0. Both x86 and OpenISA binaries were all compiled with the O3 optimization set.

The source code of the experiments was implemented in C and then compiled to OpenISA and x86. Their implementations are all divided into four main parts: (1) a routine which generates all random inputs and allocates the time vectors, (2) a loop which iterates over each of the inputs calling the cryptographic routine, (3) the cryptographic routine and (4) the data-processing routine. We only measured the execution time of the third part, the cryptographic routine. For this, we added a special instruction to get the execution cycle before and after the routine execution. In x86 we used the `rdtsc` instruction[2] and in OpenISA we use a special 'shadow' syscall added to the emulator. After the loop consumes all the inputs, the data-processing routine is called to processes the time-vectors and applies the statistical test in a cumulative online fashion. All these steps can be repeated indefinitely and, after each execution, the statistical test updates its assumptions about the algorithm.

Welch's $t$-test statistical test outputs two numbers, the $t$-value and the $p$-value. The first shows the magnitude of the difference between the execution time of the two classes (time-vectors) and the second describes the exponential tendency for the $t$-value, defining a confidence interval for the test. A $t$-value higher than 4 is usually considered a strong evidence that the two classes have different execution time and the implementation is not constant. In our experiments, we tested three algorithms with two implementations each, one known to be non-constant and another known to be constant.

### 4.1. Cryptographic Algorithms

In our experiments, we tested three algorithms each one with a well-known non-constant-time implementation for each of the classes of timing leakage discussed in Section 2.3. We

[2]Intel Assembly instruction: read timestamp counter (`rdtsc`)

also executed tests with a well-known constant-time implementation for each algorithm. The algorithms and implementations are described in Table 4.1.

| Algorithm | Description | Implementation | Timing Leakage |
|---|---|---|---|
| Memory comparison | A very common algorithm for checking message authenticated codes. It compares two 256-bit strings. | Non-constant time memory comparison [3] | Aborts on pair of different bytes, thus control-flow behavior depends on secret data |
| | | Constant-time memory comparison [4] | None |
| AES | A popular block cipher, based on Substitution-Permutation Network (SPN) model. It encrypts blocks of 128-bit of data. | T-Table Implementation [5] | Table look-ups access indexed with secret data |
| | | Bitsliced [6] | None |
| Curve25519 | An elliptic curve algorithm for key agreement protocols (ECDHE). It uses a Montgomery curve defined over prime a field defined by $2^{255} - 19$. | Non-Constant time Curve25519-donna [7] | Likely comes from variable latency instructions |
| | | Constant-time Curve25519-donna [8] | None |

Table 1. List of tested algorithms and implementations

## 5. Experimental Results

We executed our leakage detection model over the native implementations of the algorithms shown in Table 4.1. The native executions showed to always leak more and faster than when emulating the same code (OpenISA to x86). This can observed in Figures 4, 6 and 8 with native $t$-value growing quicker. Concerning the leakage during emulation, we tested it with 5 RFTs and their $t$-values showed to behave differently (Figure 4), in other words, they have different levels of leakage. For instance, NETPlus always has a $t$-value higher than NET, the same happens with NETPlus-e-r compared with NET-r, and in both cases, the only difference between the two algorithms is the application of the static forward search to expand the region. Thus, this expansion or the regions it creates somehow affect the leakage level during the emulation. It is not clear yet why these differences in the RFTs leakage levels occur, but this evidence that the phenomenon does happen should be further investigated.

Both the native and emulated tests behave as expected with constant implementations, the result in Figures 5, 7 and 9 shows that there is no leakage ($t$-value lower than 4) with native execution and, interestingly, there was also no leakage while emulating. However, this does not mean that a DBT cannot add leakage during emulation for an initially constant implementation. It is already proved that optimization (that OI-DBT and some others DBT engines apply) could add leakage points to the code. Therefore, a DBT, including OI-DBT could add leakage points to an emulation. On the other hand,

[3] Glibc `memcmp` function

[4] Compare inputs by performing XOR over all elements.

[5] [Rijmen et al. 2000]

[6] [Käsper and Schwabe 2009]

[7] Curve25519-donna implementation following the prescriptions of informational RFC 7748 [Kaufmann et al. 2016].

[8] https://code.google.com/archive/p/curve25519-donna/

and now more unexpected, OI-DBT emulation mitigated the leakage of the AES-Table and the non-constant time Curve25519-donna implementations (Figures 6 and 8). When we looked for the generated binary by the DBT, we found that the leakage causes were still there, but several others instructions to save and load the register context (explained in Section 2.1) and for type conversions were added. For instance, in Figure 10 and 11 we have the leakage point for the AES algorithm in the native and translated code. Adding these extra overheads result in removing cache related and vary-latency instruction leakages with the price of the emulation overhead, which was 10-fold slower on average than the native execution.

Figures 4, 5, 6, 7, 8 and 9 show the $t$-value difference of the execution time of the two classes of inputs. The executions presented in the x axis, shows the number of individual measurements performed in the algorithm for each batch of plain texts ($N$), tested $E$ times. We tested the different implementations both with native x86 compiled version and the emulated one, varying the RFT of the emulator. The higher the $t$-value, the more evident is the inconstancy in the execution time of the implementations or emulation. Almost all variable-time implementations are obfuscated when emulating and constant-time implementations don't show any leakage.

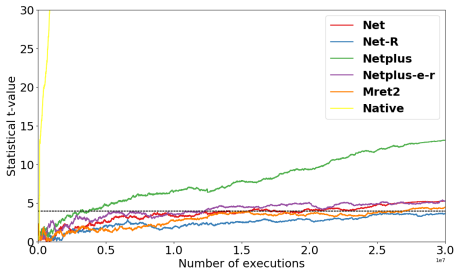



Figure 4. Variable-time memory comparison

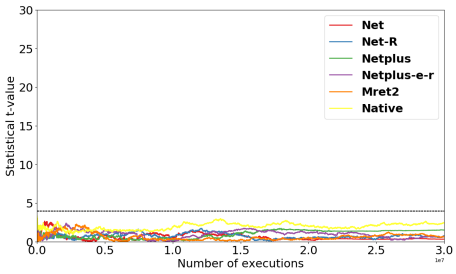



Figure 5. Constant-time memory comparison

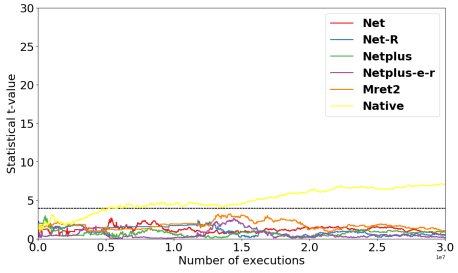



Figure 6. T-Table implementation of AES

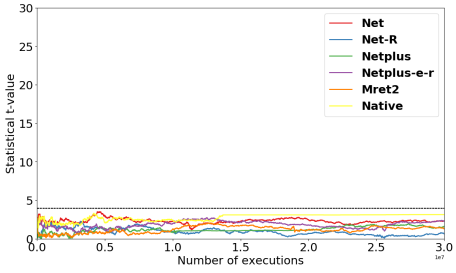



Figure 7. Bitsliced implementation of AES

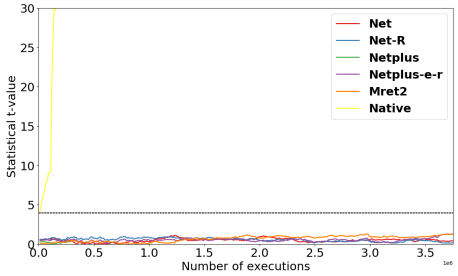



Figure 8. Variable-time Curve25519-donna

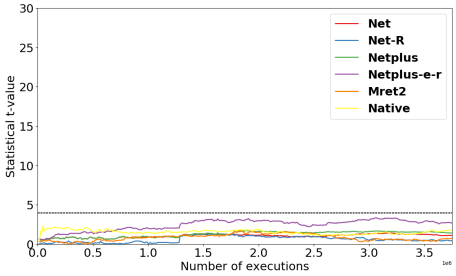



Figure 9. Constant-time Curve25519-donna

Table 2 shows the average $t$ and $p$ values for all the executions. We also present the percentage time (compare to an entire execution) that the OI-DBT used to heat (select and compile all regions), the low percentage indicates that most of the execution time was spent executing translated regions and not interpreting. Furthermore, we can observe that only MRET2 converges faster than the others RFTs, showing that the results we collected are not related to this metric. It's important to stress that, the noise introduced by the emulation process could be well-defined and characterized. By doing so, it can be removed with the appropriated pos-processing methods.

```
cmpb    $0, (%rdi)
jne     35 <do_one_computation+0x28>
leaq    1(%rdi), %rdx
addq    $1024, %rdi
jmp     15 <do_one_computation+0x21>
nopw    (%rax,%rax)
addq    $1, %rdx
cmpq    %rdi, %rdx
je      15 <do_one_computation+0x30>
movzbl  (%rdx), %eax
testb   %al, %al
je      -16 <do_one_computation+0x18>
movl    $1, %eax
retq
```

Figure 10. Native x86 code.

```
movl    12(%rdi), %ecx
addl    %eax, %ecx
movslq  %ecx, %rcx
movzbl  (%rsi,%rcx), %ecx
movl    %ecx, 4(%rdi)
movl    8(%rdi), %edx
addl    %eax, %edx
movslq  %edx, %rdx
movzbl  (%rsi,%rdx), %edx
movl    %edx, 24(%rdi)
incl    16(%rdi)
movl    $1, 20(%rdi)
cmpl    %ecx, %edx
jne     37
incl    8(%rdi)
incl    12(%rdi)
xorl    %ecx, %ecx
cmpl    $1024, 16(%rdi)
setb    %cl
movl    %ecx, 4(%rdi)
movq    $0, 20(%rdi)
jb      -75
movl    $1072, %eax
retq
```

Figure 11. Translated x86 code.

| Algorithm | Implementation | Statistics | Native | Net | Net-R | NetPlus | NetPlus-e-r | Mret2 |
|---|---|---|---|---|---|---|---|---|
| Memory Comparasion | Non-Constant | Mean t-value | 144.409 | 3.615 | 2.457 | 7.932 | 4.059 | 2.934 |
| | | Mean p-value | 0.0009 | 0.017 | 0.032 | 0.005 | 0.007 | 0.028 |
| | | RFT Convergence | 0 | 0.05% | 0.05% | 0.05% | 0.05% | 0.05% |
| | Constant | Mean t-value | 1.9633 | 0.6086 | 0.7311 | 0.9685 | 0.9728 | 0.61002 |
| | | Mean p-value | 0.0339 | 0.2900 | 0.2444 | 0.1940 | 0.1884 | 0.29175 |
| | | RFT Convergence | 0 | 0.05% | 0.05% | 0.05% | 0.05% | 0.05% |
| AES | Non-Constant | Mean t-value | 5.0036 | 1.2296 | 1.0349 | 0.7056 | 0.5685 | 1.7602 |
| | | Mean p-value | 0.0027 | 0.1232 | 0.1851 | 0.2548 | 0.3166 | 0.0578 |
| | | RFT Convergence | 0 | 12.22% | 12.49% | 12.23% | 12.22% | 9.59% |
| | Constant | Mean t-value | 2.7792 | 2.3035 | 0.9311 | 1.2049 | 1.8294 | 1.1921 |
| | | Mean p-value | 0.0056 | 0.0139 | 0.1975 | 0.1281 | 0.0466 | 0.1371 |
| | | RFT Convergence | 0 | 15.62% | 1.71% | 15.63% | 15.71% | 15.68% |
| Curve25519-donna | Non-Constant | Mean t-value | 653.7874 | 0.8038 | 0.5352 | 0.2115 | 0.5635 | 0.7063 |
| | | Mean p-value | 1.0271e-7 | 0.2154 | 0.3006 | 0.4165 | 0.2945 | 0.2542 |
| | | RFT Convergence | 0 | 5.48% | 5.61% | 5.49% | 5.61% | 4.22% |
| | Constant | Mean t-value | 1.5453 | 1.0375 | 0.6000 | 1.2530 | 2.4301 | 0.8275 |
| | | Mean p-value | 0.0676 | 0.1553 | 0.2889 | 0.1196 | 0.0267 | 0.2230 |
| | | RFT Convergence | 0 | 14.57% | 13.44% | 14.58% | 13.47% | 9.81% |

Table 2. Memory Comparison Statistic.

## 6. Related Work

Although not being a recent problem [Kocher 1996], side-channel analysis is still relevant and, because of new attacks appearing for time to time, it is frequently in the spotlight. For example, when Bernstein demonstrated a successful key-recovery side-channel attack against the AES implementation of OpenSSL 0.98 in a network scenario or with the new Spectre attacks [Kocher et al. 2018]. However, on the other hand, these attacks also motivated researchers to analyze the security of implementations and design automatic mechanisms to detect and mitigate timing leakages.

To quantify and mitigate timing leakages, several methodologies were proposed in the literature coming from diverse areas. Becker, [Becker et al. 2013] uses statistical tests

with different vector assignments. Gianvecchio and Wang (2011) showed an entropy-based model using a Kolmogorov-Smirnov test to detect covert timing channels based on estimation. Chen and Venkataramani (2014) presents an algorithm to detect the existence of a covert timing channel tracking contention patterns on shared processors and memory. Nevertheless, to our experiments, we used the Dudect approach [Reparaz et al. 2017], mainly because of its simplicity to reproduce.

Once detected the leakages, there are approaches to remove or mitigate them. Cleemput, [Cleemput et al. 2012] studied some of these approaches and their effectiveness. Most of these approaches affect the overall performance by more than 8-fold. In their posterior study [Van Cleemput et al. 2017], the authors extended the enforcement of invariable latency paths with a profile-based JIT protection by applying a selective if-conversion and transformations to regions with leakages. However, none of these works studied and analyzed the potential for multiple-target DBT to change the leakage from a program, adding or removing it, during emulation of a binary.

## 7. Conclusion

In this work, we analyze the impact of emulating a binary with and without timing leakage using OI-DBT, a multiple-target dynamic binary translator. The results showed that an emulator can add enough noise into the execution to the point in which it can actually mitigate the timing leakage. In our case, we could mitigate the timing leakage with an overhead of 10-fold the native execution time by simply emulating it. We also argue that the opposite scenario can be true, one emulator can insert timing leakage to a constant implementation by applying some optimization to the code. Moreover, we evaluate the emulation of constant and non-constant implementations using multiple types of region formation techniques, showing that they do have an impact on the final result.

This is the first work of its kind in analyzing a multiple-target DBT impact on side-channel leakage. Given the growing importance of emulators in an IoT, fog and cloud computing world with multiple ISAs, this problem will become even more important. We are already investigating these same questions in different emulators and implementing a new technique to mitigate side-channel during emulation with near-zero overheads.

## References

Becker, G., Cooper, J., DeMulder, E., Goodwill, G., Jaffe, J., Kenworthy, G., Kouzminov, T., Leiserson, A., Marson, M., Rohatgi, P., et al. (2013). Test vector leakage assessment (tvla) methodology in practice. In *ICMC*, volume 1001, page 13.

Brumley, D. and Boneh, D. (2005). Remote timing attacks are practical. *Computer Networks*, 48(5):701–716.

Cauligi, S., Soeller, G., Brown, F., Johannesmeyer, B., Huang, Y., Jhala, R., and Stefan, D. (2017). Fact: A flexible, constant-time programming language. In *2017 SecDev*, pages 69–76. IEEE.

Chen, J. and Venkataramani, G. (2014). An algorithm for detecting contention-based covert timing channels on shared hardware. In *HASP*, page 1. ACM.

Cleemput, J. V., Coppens, B., and De Sutter, B. (2012). Compiler mitigations for time attacks on modern x86 processors. *TACO*, 8(4):23.

Davis, D. and Hazelwood, K. (2011). Improving region selection through loop completion. In *ASPLOS*, volume 4, pages 7–3.

De Mulder, E., Eisenbarth, T., and Schaumont, P. (2018). Identifying and eliminating side-channel leaks in programmable systems. *IEEE Design & Test*, 35(1):74–89.

Duesterwald, E. and Bala, V. (2000). Software profiling for hot path prediction: Less is more. *ACM SIGOPS*, 34(5):202–211.

Hong, D.-Y., Hsu, C.-C., Yew, P.-C., Wu, J.-J., Hsu, W.-C., Liu, P., Wang, C.-M., and Chung, Y.-C. (2012). Hqemu: a multi-threaded and retargetable dynamic binary translator on multicores. In *CGO*, pages 104–113. ACM.

Kang, Y.-J., Bruce, N., Park, S., and Lee, H. (2016). A study on information security attack based side-channel attacks. In *ICACT*, pages 61–65. IEEE.

Käsper, E. and Schwabe, P. (2009). Faster and timing-attack resistant aes-gcm. In *CHES 2009*, pages 1–17. Springer.

Kaufmann, T., Pelletier, H., Vaudenay, S., and Villegas, K. (2016). When constant-time source yields variable-time binary: Exploiting curve25519-donna built with msvc 2015. In *International Conference on Cryptology and Network Security*, pages 573–582. Springer.

Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2018). Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*.

Kocher, P. C. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer.

Reparaz, O., Balasch, J., and Verbauwhede, I. (2017). Dude, is my code constant time? In *DATE*, pages 1697–1702. IEEE.

Rijmen, V., Bosselaers, A., and Barreto, P. (2000). Optimised ansi c code for the rijndael cipher (now aes). *Public domain software*.

Smith, J. E. and Nair, R. (2005). *Virtual Machines: Versatile Platforms for Systems and Processes*. The Morgan Kaufmann Series. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Standaert, F.-X. (2017). How (not) to use welch's t-test in side-channel security evaluations. *IACR*, 2017:138.

Van Cleemput, J., De Sutter, B., and De Bosschere, K. (2017). Adaptive compiler strategies for mitigating timing side channel attacks. *TDSC*.

Wang, C., Zheng, B., Kim, H., Jr., M. B., and Wu, Y. (2007). Two-pass mret trace selection for dynamic optimization. Patent number 20070079293.

Welch, B. L. (1947). The generalization of student's' problem when several different population variances are involved. *Biometrika*, 34(1/2):28–35.

# A Methodology for Optimization of Interpreters

**Vanderson Martins do Rosario[1], Mario Mikio Hato[1], Rodolfo Azevedo[1], Edson Borin[1]**

[1]Institute of Computing – UNICAMP – Campinas – SP – Brasil

vanderson.rosario@ic.unicamp.br, mario.hato@lsc.ic.unicamp.br

{rodolfo, edson}@ic.unicamp.br

***Abstract.*** *Interpretation is a simple and portable technique that enables emulation of instruction set architectures (ISAs). Even though other techniques, such as binary translation, still provide superior emulation performance, interpreters are easier to implement and debug. Because of that, they are often used to complement binary translation techniques or to design reference instruction set simulators (ISS). In this work, we characterize the main causes of overhead in an interpreter and present a methodology to analyze and optimize interpreters code. The methodology was designed having in mind the most common practices in the development of interpreters, the operation of modern processors architectures and memory hierarchy. Finally, to evidence the importance of the methodology we present a case study in which we used it to optimize an interpreter-based ISS, improving its performance by 48%.*

## 1. Introduction

According to Smith and Nair [Smith and Nair 2005], interpretation is a common way of implementing an emulator and it is used in several applications such as in architectural simulation and Virtual Machines (VMs), i.e. Android VM [art 2018] and the Java VM [Häubl and Mössenböck 2011]. Interpreters are normally used because they provide better portability and are easy to implement and debug, but when compared to more sophisticated emulation techniques, such as Dynamic Binary Translation (DBT) and JIT compilers, its performance can be much lower.

Interpretation performance can be even worse when the code is created by an automatic Instruction-Set Simulator (ISS) generator. So, in order to mitigate this issue, engineers manually analyze and optimize the interpreter code. For example, the Android VM has two hand-crafted interpreters written in assembly, one for x86 and another for ARM. However, this can be a hard engineering process, especially without guidelines. In this work, we propose a methodology to analyze and optimize the performance of an interpreter code. This methodology can be applied to any of the most common techniques available to implement interpreters and, to evidence its power, we follow its guidelines to optimize an interpreter-based ISS generated by ArchC [Azevedo et al. 2005]. As our results show, the methodology-guided optimizations reduced the pressure on the cache system and the average number of host instructions used to emulate each guest instruction, improving the interpreter performance by 48%.

Acknowledgement: the authors would like to thank CAPES (PROEX - 0487/2018), FAPESP (2013/08293-7), CNPq and Microsoft for their financial support and the Multidisciplinary High Performance Computing Lab (LMCAD) for its infrastructure and technical support.

This text is organized as follows: Section 2 presents the most common issues that hurt the performance of interpreters; Section 3 describes the proposed methodology step-by-step and Section 4 shows how it is applied to optimize an interpreter-based ISS; Finally, Section 5 discusses the related work and Section 6 presents our conclusions.

## 2. Characterization of Interpretation Overhead

Interpretation is a technique that provides means for executing a program compiled for a guest ISA on different ISA processor, called host ISA [Smith and Nair 2005]. This technique mimics the behavior of a simple CPU by fetching, decoding and executing the guest instructions one by one. Its main advantage is that the behavior of the guest instructions can be easily implemented in High-Level Language (HLL) and compiled to several architectures. So, an interpreter written in an HLL, i.e. C, is not only small but also extremely portable and easy to debug.

A typical implementation of an interpreter contains a loop that fetches and decodes instructions from the guest code and a dispatcher that selects a segment of code, usually a routine, to emulate the instruction behavior. This approach, known as the classic or switch-based interpretation [Rossi and Sivalingam 1996], is one of the simplest and most common techniques. It is, for instance, the technique present in automatically generated simulators, such as ArchC [Azevedo et al. 2005], and in the portable version of the Android ART interpreter. This classic interpretation technique contains a loop that a) fetches the guest instructions indexed by the virtual program counter (vPC), b) decodes the instruction to extract the opcode `OP`, and c) selects the proper segment of code to emulate the guest instruction behavior using a switch statement for every emulated instruction.

In addition to the classic approach, there are also variations that were proposed to improve the interpretation performance, such as the Indirect Threaded Code [Dewar 1975, Klint 1981, Smith and Nair 2005], the Context Threading [Berndl et al. 2005], and the Replication [Casey et al. 2007] approach. Despite the differences among the techniques, the issues that affect their performance are similar and our methodology works without modifications in all of them.

Opposing to the high-level implementation, an interpreter can also be implemented in assembly, normally handcrafted. This type of interpreter is able to achieve lower levels of overhead, but they are usually specialized to a single architecture and platform. On that account, if one wants an interpreter to support three different architectures, she would need to implement and optimize three versions, one to each; much more expensive and far less productive. Thus, for interpreters, there is a clear trade-off between low overhead and portability.

As a hand-craft interpreter written in Assembly is normally done by specialists and focuses on performance, in our research, we focus on HLL implementations. We observed that the performance of a portable interpreter, written in an HLL, is usually affected by the following factors:

- **Host/Guest Instruction Proportion**: in order to emulate every instruction from the guest application, the interpreter must fetch the instruction, decode it and invoke the proper code to emulate its behavior. This process may require the execution of several host instructions. The Host/Guest instruction proportion, or **IH/IG**

proportion, represents the average number of host instructions required to execute each guest instruction.

- **Cache Pressure:** the code to mimic the behavior of an instruction is, in most cases, implemented with more than one instruction and usually causes more pressure on the CPU instruction cache than the single guest instruction. Moreover, since the interpreter loads the guest instruction from memory as data, there is also an additional pressure over the CPU data cache.
- **Branch Prediction Pressure:** the dispatcher executes several branches. For instance, a classical dispatcher executes at least two branches for each instruction emulated: one to select the proper emulation code and another to return to the head of the loop. These extra branch instructions impose more pressure over the branch predictor.
- **Initialization overhead**: before starting the execution, the interpreter needs to instantiate its internal data structures, and only then it loads the guest binary. Thereby, it takes some time to initialize, adding an extra overhead to all executions.

## 3. The Methodology for Optimization of Interpreters

In this section, we present a methodology to address the main causes of overhead stated in the previous section and improve the performance of interpreters. The methodology leverages micro benchmarks to expose the main performance issues and guide the developer to the segments of code that are responsible for the overhead. The methodology can be summarized in three steps: (1) selection and creation of micro benchmarks; (2) investigation of optimizations based on the results from the micro benchmarks; and (3) selection of the best combination of optimizations to generate the final optimized interpreter. Each one of these three steps is further divided, resulting in eight steps that are illustrated in Figure 1. The eight steps are the following:

1. **Filter out short-running benchmarks**: once the sets of benchmarks that will be used to evaluate the performance of the interpreter are chosen, it is necessary to filter them by keeping only the ones with a significant execution time. This is important to make sure that the information collected by the profiler reflects the interpretation performance instead of the loader initialization process. To this end, we recommend removing any benchmarks for which the initialization time takes more than 10% of the total execution time.
2. **Find the hot instructions**: one should run the benchmarks (already filtered by the first step) and profile their execution to identify which are the most frequent executed guest instructions. The goal is to select the most significant ones, which together represents 90% of all execution frequency, also known as the 90% cover set, for each benchmark. Then, having the sets in hand, one should rank the instructions by the number of times each one appeared in the 90% cover sets, creating a sorted list of the most executed instructions for these benchmarks. Probably reflecting also the most common instructions in that architecture and for the used compiler.
3. **Create the micro benchmarks**: for each one of detected hot instructions, one should create a micro benchmark containing a loop to execute this instruction millions of times. The goal is to make sure that micro benchmark causes the

interpreter to spend most of its execution time on the fetch-decode-execute loop of this instruction, instead of the load and initialization process.

4. **Analyze the micro benchmarks overhead**: having the micro benchmarks in hand, one should run them and measure: a) number of guest instructions executed per second in MIPS[1], b) the Instructions from Host/Instructions from Guest (IH/IG) proportion, c) the percentage of cache misses, and d) the branches miss prediction rate. Then, compare the micro benchmarks performance results in order to identify discrepancies and formulate hypotheses about the potential performance issues. In this step, it is important to keep in mind the main sources of performance overhead discussed in Section 2.
5. **Find possible optimizations**: based on the sources of overhead identified in the previous step, list the possible optimizations for the interpreter and create different versions of it for each optimization. Some of the possible optimizations which could be applied are discussed in Section 4.
6. **Generate optimization sequences**: with different versions of the code, use diff files to apply the optimizations in the original code, choosing randomly which optimizations to apply or using any preferable heuristic (random search, genetic algorithm...). Save each sequence applied and the resulted code. Note that one can generate all possible sequences, but the possibilities grow fast and become unfeasible even with a small number of optimizations. Autotuning techniques could be used in this step.
7. **Analyze the performance of each sequence**: for all generated version of the interpreter, measure their performances.
8. **Choose the best sequence**: since the optimization space is very large, it is important to have a heuristic to decide when to stop the exploration. For instance, one can choose to generate a fixed number of sequences and chose the best at the end. Lastly, having the best interpreter code, run it for all benchmarks, including the filtered ones, to calculate the final speedup.

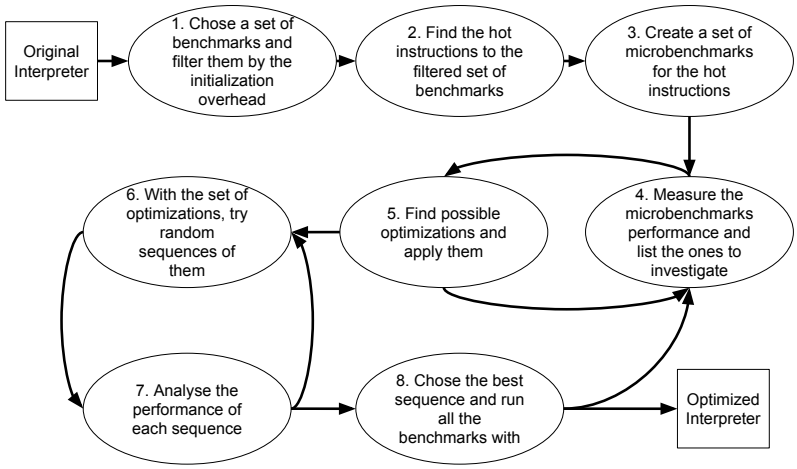



**Figure 1. Methodology flowchart.**

Steps four, five, six and seven are iterative, and the number of iterations should be defined depending on the available time to be spent on the project or the speedup goal.

[1]Millions of Instructions Per Second

One can also find, at the end of step eight, that the results are not good enough and decide to return to step 4.

In step five, it is important to have in mind that there is a recurrent way to reduce each one of the performance metrics (IH/IG, cache misses, and branch miss prediction). Usually, a high IH/IG is related with excessive computation in the emulation of an instruction; cache misses are related with an excessive use of memory; and branch miss prediction with the excessive use of `if` instruction or the dispatcher method used.

Finally, for each metric, one can have two possible results: all instructions (micro benchmark) with the same value or mostly equal but with some outliers. In the first case, it is plausible that the optimization needed will be in a common code to the emulator of all instructions, i.e. the dispatcher or the decoder. In the second case, the problem is likely to be in the specific code to mimic the behavior of the instruction or in functions called in this part of the code.

Of course, the performance of the iteration between the instructions and not only the performance of the individual instructions are important, but testing this iteration is problematic as the possibilities are endless. So, we focus only on optimizing each individual instruction in our methodology, and this approach has demonstrated to be enough to obtain high-performance gains in our case study.

## 4. Case Study: Optimizing the ArchC Auto-generated MIPS Interpreter

In this section, we present a case study in which we apply the proposed methodology to optimize an interpreter-based ISS. The simulator emulates the 32-bit MIPS ISA and was automatically generated by the `acsim` ArchC tool [Azevedo et al. 2005]. The `acsim` tool takes as input a description of an ISA, in this case, a 32-bit MIPS (big-endian) with 59 instructions and three formats of instruction encoding, and then it generates a simulator, for the given specification, using an interpreter with a classical dispatcher. To accomplish the experiments, we used ArchC 2.2 and the performance was measured using the Linux Perf tool (version 4.18.g94710ca). All measurements we present during the section for micro benchmarks and benchmarks were computed taking the median from 10 executions. We used 55 benchmarks from MiBench [mibench 2018] and MediaBench [mediabench 2018], all compiled using GCC 4.4.3 in a machine with Ubuntu Server 10.4 x64 (Linux 2.6.32), 16 GB of RAM and an Intel Core i7 980X with 6 cores with the frequency fixed at 3.3 GHz.

### 4.1. Filtering the Benchmarks by the Initialization Overhead

The first step consists in computing the initialization overhead of the interpreter. To this end, we developed a set of micro benchmarks containing a loop with a specified number of *add* instructions. We varied the number of iterations and the quantity of *adds*, executing all variations and collecting the number of host instructions for each with Perf. Then, for each variation of the micro benchmark we created a new version with the first instruction being an instruction to finalize the program. Thus, executing these new versions and comparing the results with the first ones, we got the initialization time, as they will have the same number of static instruction (therefore same load cost) but they will finish as soon as they start. We observed with this experiment that the initialization cost is around six millions host instructions.

Then, we measured the number of instructions executed by all benchmarks and filtered out the ones executing less than 100 million instructions as the initialization would have a significant impact on them ($\gtrsim 6\%$ of the total of instructions). As discussed before, this is important to ensure that the profiler will be measuring mainly the interpretation overhead and not the initialization cost, cleaning and facilitating the analysis of the results. In the end, from the initial 55 benchmarks (Table 1), only 27 met the requirements, 24 from MiBench and 3 from MediaBench (the ones marked with a tick in the table). Finally, for a better representation in the graphics, for now on, we will only use an id to refer to each benchmark. Table 1 shows all ids.

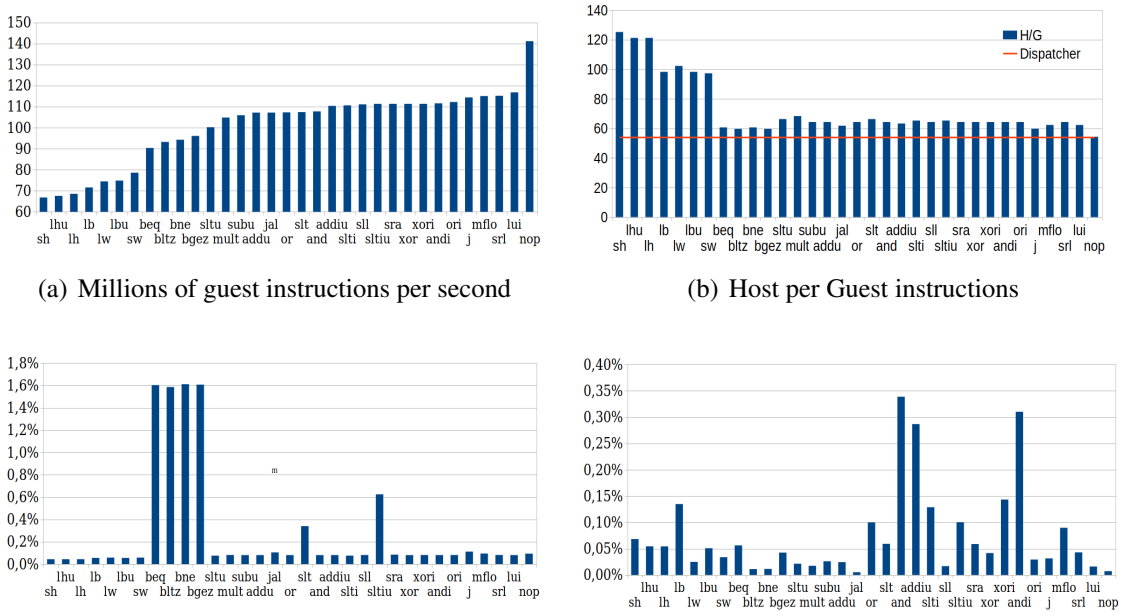
**Table 1. MiBench and MediaBench benchmarks.**

| Suite | Id | *Benchmark* | |
|---|---|---|---|
| MiBench | 0 | basicmath_small | ✓ |
| | 1 | basicmath_large | ✓ |
| | 2 | bitcount_small | |
| | 3 | bitcount_large | ✓ |
| | 4 | qsort_small | |
| | 5 | qsort_large | ✓ |
| | 6 | susan_small_corners | |
| | 7 | susan_small_edges | |
| | 8 | susan_small_smoothing | |
| | 9 | susan_large_corners | |
| | 10 | susan_large_edges | ✓ |
| | 11 | susan_large_smoothing | ✓ |
| | 12 | jpeg_small_encode | |
| | 13 | jpeg_small_decode | |
| | 14 | jpeg_large_encode | ✓ |
| | 15 | jpeg_large_decode | |
| | 16 | lame_small | ✓ |
| | 17 | lame_large | ✓ |
| MiBench | 18 | dijkstra_small | |
| | 19 | dijkstra_large | ✓ |
| | 20 | patricia_small | ✓ |
| | 21 | patricia_large | ✓ |
| | 22 | stringsearch_small | |
| | 23 | stringsearch_large | |
| | 24 | rijndael_small_encode | |
| | 25 | rijndael_small_decode | |
| | 26 | rijndael_large_encode | ✓ |
| | 27 | rijndael_large_decode | ✓ |
| | 28 | sha_small | |
| | 29 | sha_large | ✓ |
| | 30 | adpcm_small adpcm | |
| | 31 | adpcm_small pcm | |
| | 32 | adpcm_large adpcm | ✓ |
| | 33 | adpcm_large pcm | ✓ |
| | 34 | CRC32_small | |
| | 35 | CRC32_large | ✓ |
| | 36 | FFT_small | ✓ |
| MiBench | 37 | FFT_small_inv | ✓ |
| | 38 | FFT_large | ✓ |
| | 39 | FFT_large_inv | ✓ |
| | 40 | gsm_small_encode | |
| | 41 | gsm_small_decode | |
| | 42 | gsm_large_encode | ✓ |
| | 43 | gsm_large_decode | ✓ |
| MediaBench | 44 | adpcm_encode | |
| | 45 | adpcm_decode | |
| | 46 | gsm_encode | ✓ |
| | 47 | gsm_decode | |
| | 48 | jpeg_encode | |
| | 49 | jpeg_decode | |
| | 50 | mpeg2_encode | ✓ |
| | 51 | mpeg2_decode | ✓ |
| | 52 | pegwit_generate | |
| | 53 | pegwit_encrypt | |
| | 54 | pegwit_decrypt | |

Once the benchmarks were filtered, we performed, for each one, an instruction coverage analysis and selected the minimal set of instructions that together covered 90% of the benchmark execution. The union of all these sets gave us a total of 34 (out of 59) instructions which are the most frequently executed instructions and, therefore, important to the final performance of the two-benchmark suites. Thus, from now on, we focus our analysis on them, analyzing and improving their individual performance.

For each one of these 34 instructions (listed in the X-axis in Figure 2(a)), we created a micro benchmark that executes 100 million times the instruction. We then used Perf to measure: a) the number of instructions emulated per second, b) the average number of host instructions executed per guest instruction, c) the branch miss prediction rate, and d) the cache miss rate. The results are shown in Figure 2.

In this step of the methodology, we want to understand where it is worth to apply optimizations. The four graphics give us the first insights. From Figure 2(a), there are three sets of results drawing attention: (1) the performance of the `nop` instruction, which exposes only the overhead of fetching, decoding and dispatching since it does not involve any execution, as their is no behavior to mimic; (2) the poor performance of load and store micro benchmarks (`lb`, `lbu`, `lh`, `lhu`, `lw`, `sh`, `sw`), which are slower than other micro benchmarks and indicate a poor performance in memory manipulation; and (3) the poor performance of micro benchmarks for branch instructions (`beq`, `bgez`, `bne`, `j`), which also suggests that these instructions may suffer from issues associated with their behavior.




(a) Millions of guest instructions per second

(b) Host per Guest instructions

(c) Branch miss prediction rate

(d) Cache miss rate

**Figure 2. Overhead measurement for the micro benchmarks for each one of the instruction by their MIPS.**

Figure 2(b) shows the average number of host instructions executed for each guest instruction (IH/IG). These results suggest that loads and stores had poor performance because they execute, on average, more host instructions to emulate each guest instruction. As the process of decoding and dispatching can be estimated by the `nop` result, we separate the cost associated with the behavior of the instruction and the cost associated with the fetch-decode-dispatch process by drawing a red horizontal line in the graphic.

Figure 2(c) indicates that the micro benchmarks designed for the branch instructions suffer from more branch miss predictions than the other micro benchmarks. These results suggest that the code that emulates the behavior of these instructions may be putting extra pressure on the CPU branch predictor. The micro benchmarks for the `slt` and `sltiu` instructions also suffered from higher branch miss rates, due to their intrinsic internal conditional test. Finally, Figure 2(d) indicates that there is some variation on the cache miss rate; however, we could not find any correlation with the performance of the micro benchmarks.

Having these results in mind, we developed the optimizations further explained in sections 4.2 and 4.3.

## 4.2. Investigating and Optimizing Loads and Stores

Given the poor performance results of load and store instructions and its strong correlation with the IH/IG instruction proportion metric, we investigated the code that emulates their behavior. In this process, we identified four possible optimizations in the code:

- **No-Local Copy**: interpreter methods used to support the access of the guest memory, both to read and write, do an unnecessary copy of data to an auxiliary local structure. By removing this copy we are able to reduce memory manipulation.

- **Endianness**: the interpreter checks if there was an endianness swap during the execution to treat the data differently, but MIPS does not have any instruction to change the endianness, making the verification useless. So, we remove it.
- **Routines Specialization**: the interpreter uses a single function to deal with different sizes of data, using a switch statement to select the correct code for each instruction. However, this approach increases the complexity and size of the function and may inhibit compiler optimizations as inlining. To solve this issue, we implemented specialized functions for each data size.
- **Function Return**: when loading data, the result is returned using a parameter, which is preventing the compiler to store the value in a register and making it difficult to optimize it. Therefore, we changed the load function to return the value as a simple function return value.

To test the impact of each one of these optimizations, we created five versions of the interpreter, each one containing a different sequence of optimizations, as described in Table 2.

**Table 2. Optimizations applied to each one of the interpreter versions.**

| Optimization | v0 | v1 | v2 | v3 | v4 |
|---|---|---|---|---|---|
| No-Local Copy | | ✓ | ✓ | ✓ | ✓ |
| Endianness | | | ✓ | ✓ | ✓ |
| Routines Specialization | | | | ✓ | ✓ |
| Function Return | | | | | ✓ |

The main goal of those optimizations is to reduce the IH/IG metric for load and store instructions, which was achieved as it is shown in Figure 3(a). Figure 3(b) shows the load and store micro benchmarks performance in millions of guest instructions executed per second. Notice that when all optimizations are applied (v4) the IH/IG proportion is reduced by an average of 32% and the performance improved by 20%.


(a) Host per guest instructions

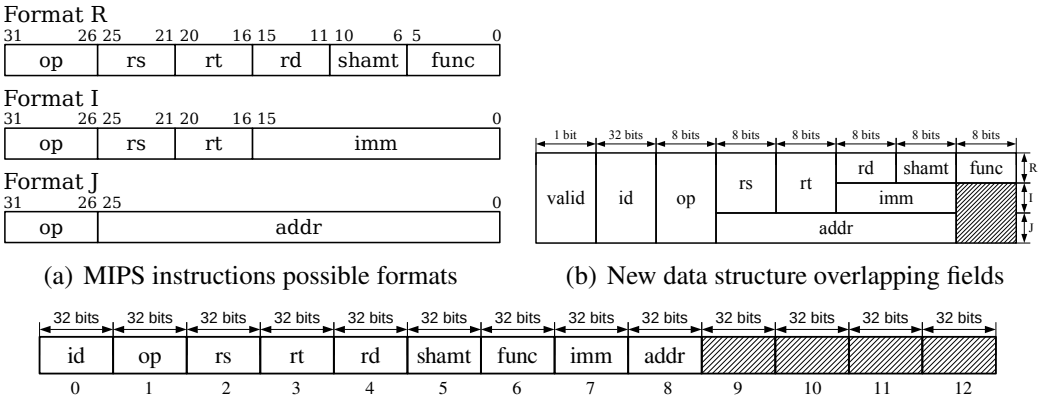
(b) Millions of guest instructions per second

**Figure 3. Performance of load and store micro benchmarks for each version of the interpreter.**

## 4.3. Reducing the Pressure on the Cache System

One of the main causes of high cache miss rate in an interpreter is a poor representation of decoded instructions in memory. After investigating this data structure in the ArchC auto-generated MIPS interpreter, we noticed an excessive use of memory. To easily generate

a representation for any architecture, the ArchC `acsim` tool creates a vector with one 32-bit field for each possible field in the guest instruction plus an extra field to identify the instruction. For instance, as illustrated in Figure 4(a), there are three formats in the MIPS ISA, one with six fields, another with four and the last one with two, resulting in 12 distinctive fields. In this case, `acsim` creates a 32-bit field for each one of these 12 fields and an extra one for the instruction identification, which results in a 416-bit data structure to represent a single guest instruction, as illustrated in Figure 4(c). In order to reduce the use of memory to represent decoded instructions, we manually modified the data structure to be more compact by overlapping fields that are never used together. The new data structure is illustrated in Figure 4(b).




(a) MIPS instructions possible formats (b) New data structure overlapping fields

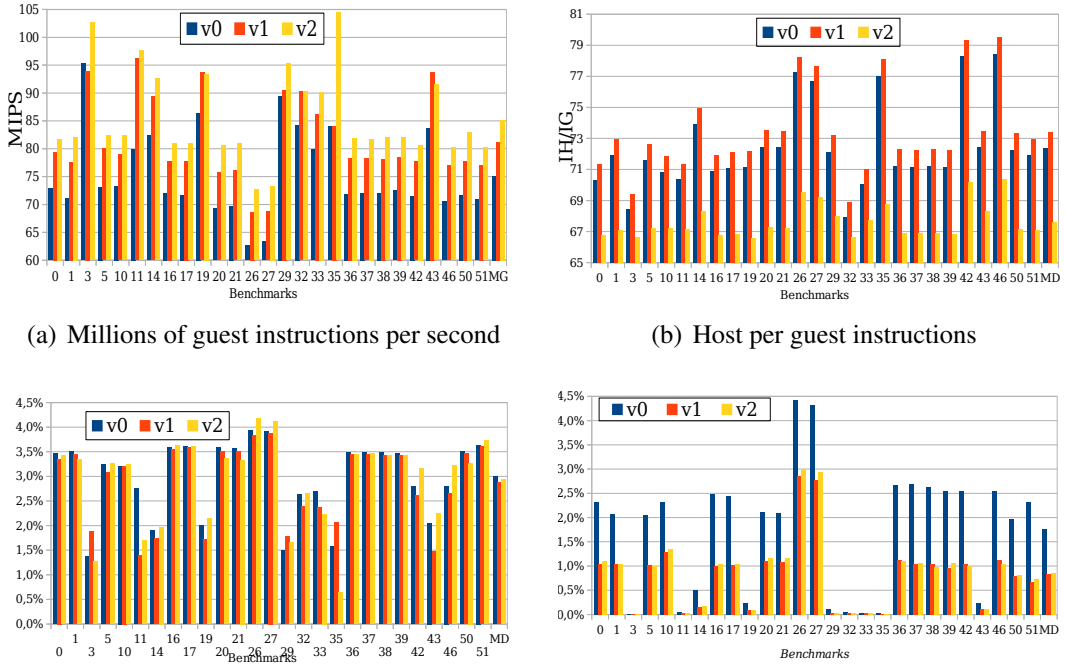
(c) ArchC MIPS interpreter representation

**Figure 4. Memory representation of the guest instruction.**

With the new data structure, the fields of the guest instruction require only 81 bits, more than 5 times smaller than the original. Consequently, the interpreter accesses less memory and imposes less pressure on the cache system, which may reduce the cache miss rate and improve the general performance. Figure 5 presents the performance and profiling results for three versions of the interpreter: **v0** is the interpreter without optimizations, **v1** is the interpreter with the new decoded instruction data structure, and **v2** combines the new data structure with the full set of optimizations presented in Section 4.2. As can be seen in Figure 5(a), **v1** is faster than **v0**, indicating that the new data structure improves the interpreter performance. Also, **v2** is faster than **v1**, which indicates that the interpreter performance can be improved even further when combining these different optimizations. Using both optimizations (V2) we improved the performance by 13%.

It is also possible to see in Figure 5(b) that the average number of host instructions required to emulate each guest instruction increases when the new data structure is included in the interpreter (**v1**). However, despite this increment on IH/IG proportion, Figure 5(a) shows that the performance is improved by 8.14%. The main reason for the performance improvement is the 52% reduction in cache misses due to the new data structure, as indicated by Figure 5(d).

As can be seen in Figure 5(c), the three interpreter versions present little or no difference in the branch miss prediction rate. This result is expected since the optimizations performed so far are not targeted at reducing this metric.




(a) Millions of guest instructions per second

(b) Host per guest instructions

(c) Branch miss predictions

(d) L1 cache misses

**Figure 5. Overhead measurements of the interpreter with and without the new structure for representing decoded instructions. Benchmarks id can be seen in Table 1 and MG/MD is used for the geometric mean of the results.**

### 4.4. Automatic Exploration of the Optimization Sequence Space

We iterate three more times in analyzing the performance and looking for new optimizations. After these iterations, we found several modifications that could improve the interpreter performance. For a matter of space, we present all of them briefly in Table 3.

As we did before, these optimizations can be combined together, however, in our experiments we verified that some combinations are harmful to performance. Hence, it is important to find a good set of optimizations that, when combined, maximize the performance gains. In order to search for this set, we fixed what we think are the 6 most important optimizations (marked with a tick in Table 3). We called them the base set of optimization. All others sets were randomly created by removing or inserting other optimizations from the table. We generated 245 different configurations. For each configuration, we executed all benchmarks with the interpreter optimized for them. In the end, we select the top 9 optimization sets that provided the best results for at least one of the benchmarks. These sets and their respective speedup are summarized in Table 4.

The base optimization set provides, by itself, strong results, reducing the IH/IG proportion by 29%, the cache misses by 67% and increasing the performance by **23%**. Our exploration for a better set of optimization than the base gave us C0 ( New_Stop, Syscall_Jump, Threading, No_Wait, No_PC_Ver and Full_Decode), which is able to further improve the performance (when compared to base optimization set) by **20.55%**. Hence, when combining the best sequence of extra optimizations with the base optimization set, the performance of the original interpreter is improved by **48%** ($1.23 \times 1.2055 = 1.48$).

**Table 3. Summary of the designed optimizations**

| | Optimizations | Fixed | Description |
|---|---|---|---|
| A | *LD_ST_Mem* | ✓ | Optimizations presented in Section 4.2 |
| B | *New_Dec_Cache* | ✓ | New structure for decoded instruction (Section 4.3) |
| C | *New_Annul* | ✓ | Redefinition of the flag system |
| D | *No_Wait_Sig* | ✓ | Remove the wait signal verification |
| E | *New_Config* | ✓ | Change the way the interpreter is initialized |
| F | *No_Ident_Inst* | ✓ | Disabling the verification of self-modified code |
| G | *New_Stop* | | Remove the check for the end of emulation |
| H | *Force_Inline* | | Add directive to force inline in critical functions |
| I | *Syscall_Jump* | | Direct jumping to a syscall coroutine (threading) |
| J | *Threading* | | Support for a new dispatcher based in threading |
| K | *No_Wait* | | Disabling multi-thread synchronization |
| L | *No_PC_Ver* | | No PC border check |
| M | *Index_Fix* | | Use fixed index in the decoder cache |
| N | *Full_Decode* | | Decode the entire binary before execution |
| O | *No_Save_ID* | | Remove the state information used in warnings |
| P | *New_Syscall* | | Reducing operations to use a syscall |

**Table 4. The best found optimizations sequences.**

| Version | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 |
|---|---|---|---|---|---|---|---|---|---|
| G–*New_Stop* | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| H–*Force_Inline* | | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ |
| I–*Syscall_Jump* | ✓ | ✓ | ✓ | ✓ | | ✓ | | | |
| J–*Threading* | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ | ✓ |
| K–*No_Wait* | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| L–*No_PC_Ver* | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| M–*Index_Fix* | | | | ✓ | | ✓ | | ✓ | |
| N–*Full_Decode* | ✓ | ✓ | ✓ | ✓ | | ✓ | | | ✓ |
| Speedup | 1.205 | 1.205 | 1.204 | 1.193 | 1.192 | 1.189 | 1.179 | 1.177 | 1.177 |
| Best for | 3 | 8 | 7 | 1 | 3 | 1 | 1 | 2 | 1 |

## 5. Related Work

There have been several works investigating the overhead of specific interpreters, such as the work of Barany [Barany 2014] or Wang, Wu and Padua [Wang et al. 2014] which analyzed the code and proposed handcrafted optimizations to improve the performance of the R interpreter or Edwards [Edwards 2006] which argues that simulations either run quickly or are implemented quickly, and propose a specialization on some SystemC fix-point operations to gain performance at the cost of losing generality. All these optimizations are useful to learn and understand, but none of the cited works presented a methodology to systematically guide an optimization process in any given interpreter. Hence, as far as we are concerned, this is the first work in the literature to propose a methodology to guide developers to analyze and optimize interpreters in a systematic and generic way.

## 6. Conclusions

In this work, we discussed the main causes of overhead in interpreters and proposed a methodology to guide developers to analyze and optimize their code. We also applied this methodology to optimize an interpreter-based ISS generated by ArchC [Azevedo et al. 2005], which reduced the pressure on the cache system and the average number of host instructions used to emulate each guest instruction and improved the interpreter performance by

48%. We expect this methodology to help other scientists and engineers in the process of handcrafting or manually optimizing other interpreters. We believe that this methodology is an important tool to be learned by interpreter developers and shows the importance of profiling constantly to understand overhead causes and only then looking for optimizations.

## References

(2018). Art and dalvik. https://source.android.com/devices/tech/dalvik/ . Last Accessed: 07/2018.

Azevedo, R., Rigo, S., Bartholomeu, M., Araujo, G., Araujo, C., and Barros, E. (2005). The ArchC architecture description language and tools. *International Journal of Parallel Programming*, 33.

Barany, G. (2014). Python interpreter performance deconstructed. Dyla'14, pages 5:1–5:9, New York, NY, USA. ACM.

Berndl, M., Vitale, B., Zaleski, M., and Brown, A. D. (2005). Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. CGO '05, pages 15–26, Washington, DC, USA. IEEE Computer Society.

Casey, K., Ertl, M. A., and Gregg, D. (2007). Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Trans. Program. Lang. Syst.*, 29(6).

Dewar, R. B. K. (1975). Indirect threaded code. *Commun. ACM*, 18:330–331.

Edwards, S. A. (2006). Using program specialization to speed systemc fixed-point simulation. PEPM '06, pages 21–28, New York, NY, USA. ACM.

Häubl, C. and Mössenböck, H. (2011). Trace-based compilation for the java hotspot virtual machine. In *PPPJ*, pages 129–138. ACM.

Klint, P. (1981). Interpretation techniques. *Software — Practice & Experience*, 11(9).

mediabench (2018). Mediabench. http://euler.slu.edu/ fritts/mediabench/mb1/. Last Accessed: 07/2018.

mibench (2018). Mibench. http://www.eecs.umich.edu/mibench/index.html. Last Accessed: 07/2018.

Rossi, M. and Sivalingam, K. (1996). A survey of instruction dispatch techniques for byte-code interpreters. Technical report, Seminar on Mobile Code, Number TKO-C-79, Laboratory of Information Processing Science, Helsinki University of Technology, 1995.

Smith, J. and Nair, R. (2005). *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Wang, H., Wu, P., and Padua, D. (2014). Optimizing r vm: Allocation removal and path length reduction via interpreter-level specialization. CGO '14, pages 295:295–295:305, New York, NY, USA. ACM.

# Towards a High-Performance RISC-V Emulator

**Leandro Lupori**[1], **Vanderson Martins do Rosario**[1], **Edson Borin**[1]

[1]Institute of Computing – UNICAMP – Campinas – SP – Brasil

leandro.lupori@gmail.com, {vanderson.rosario, edson}@ic.unicamp.br

***Abstract.*** *RISC-V is an open ISA which has been calling the attention worldwide by its fast growth and adoption, it is already supported by GCC, Clang and the Linux Kernel. Moreover, several emulators and simulators for RISC-V have arisen recently. However, none of them with good performance. In this paper, we investigate if faster emulators for RISC-V could be created. As the most common and also the fastest technique to implement an emulator, Dynamic Binary Translation (DBT), depends directly on good translation quality to achieve good performance, we investigate if a high-quality translation of RISC-V binaries is feasible. To this, we used Static Binary Translation (SBT) to test the quality that can be achieved by translating RISC-V to x86 and ARM. Our experimental results indicate that our SBT is able to produce high-quality code when translating RISC-V binaries to x86 and ARM, achieving only 12%/35% of overhead when compared to native x86/ARM code. A better result than well-known RISC-V DBT engines such as RV8 or QEMU. Since DBTs have its performance strongly related with translation quality, our SBT engine evidence the opportunity towards the creation of RISC-V DBT emulators with higher performance than the current ones.*

## 1. Introduction

RISC-V is a new, open and free ISA initially developed by the University of California [Waterman et al. 2014] and now maintained by the RISC-V foundation [RV-foundation 2018a] with a handful of companies supporting its development. It is a small RISC-based architecture divided in multiples modules that support floating-point computation, vector operations, and atomic operations, each one focusing on different future computing targets such as IoT embedded devices and cloud servers. RISC-V is calling attention worldwide by its fast growth and adoption. By now, it is supported by the Linux Kernel, GCC, Clang, not to mention several RISC-V simulators [Ta et al. 2018, Ilbeyi et al. 2016] and emulators [Clark and Hoult 2017, Bartholomew 2018]. However, at the current time, all emulators for RISC-V suffer from poor performance.

Having a high-performance RISC-V emulator for common architectures, i.e. x86 and ARM, would not only facilitate its adoption and testing but also would show it as a useful virtual architecture to ease software deployment. One approach to implement a high-performance emulator is by using Dynamic Binary Translation (DBT) [Smith and Nair 2005], a technique that selects and translates regions of code dynamically during the emulation. This technique has been used to implement fast virtual machines (VMs), simulators, debuggers, and high-level language VMs. For example, it

Acknowledgements: the authors would like to thank CAPES (PROEX - 0487/2018), CNPq, Microsoft and Instituto de Pesquisas Eldorado for their financial support and LMCAD for its infrastructure and technical support.

has been used to facilitate the adoption of new processors and architectures, such as the Apple's PowerPC to x86 migration Rosetta software [Apple 2006], to enable changes in microarchitecture without changing the architecture itself, as with the Transmeta VLIW processor [Dehnert et al. 2003] that implements x86, or in the deployment of high-level languages in several platforms such as with the Java VM [Häubl and Mössenböck 2011]. A DBT engine usually starts by interpreting the code and then, after heating (translating all hot regions), it spends most of the time executing translated regions instead. Thus, the quality and performance of these translated regions are responsible for most of the DBT engine performance [Borin and Wu 2009] and there are two DBT design choices which affect most of the quality of translation: (1) the DBT's Region Formation Technique (RFT) which defines the shape of the translation units [Smith and Nair 2005] and (2) the characteristics of the guest and host ISA which can difficult or easy the translation [Auler and Borin 2017]

While RFT design choice is well explored in the literature, the translation quality of each pair of guest and host ISA needs to be researched and retested for every new ISA. One approach to understanding the quality and difficulty of code translation for a pair of ISAs is by implementing a Static Binary Translation (SBT) engine [Auler and Borin 2017]. SBTs are limited in the sense that they are not capable of emulating self-modifying code and may have difficulty differentiating between data and code, however, its design and implementation is usually much simpler than a DBT. Since the translation mechanisms in a DBT and an SBT are very similar and creating an SBT engine which can emit high-quality code implies that the same could be done with a DBT engine, we implement and evaluate a LLVM-based SBT to investigate whether or not it is possible to produce high-quality translations from RISC-V to x86 and ARM. Our SBT is capable of producing high-quality translations, that execute almost as fast as native code, with only 1.21x/1.39x slowdown in x86/ARM. These results suggest that it is possible to design and implement a high-performance DBT to emulate RISC-V code on x86 and ARM platforms. The main contributions of this paper are the following:

- We show that it is possible to perform a high-quality translation of RISC-V binaries to x86 and ARM.
- We compare the performance of our SBT with the performance of state-of-the-art RISC-V emulators and argue that there is a lot of room for performance improvement on RISC-V emulators.

The rest of this paper is organized as follows. Section 2 further describes DBT and SBT techniques, the challenges to implement both and discusses ISA characteristics that are difficult to translate. Section 3 describes the main characteristics of the RISC-V ISA and Section 4 presents other emulators for RISC-V. Then, in Sections 6 and 7 we discuss our SBT for RISC-V and the results we have obtained with it. Finally, Section 8 presents our future work and conclusion.

## 2. Binary Translation and Challenges

Interpretation, SBT and DBT are well known methods used to implement ISA emulators. Interpretation is a technique that relies on a fetch-decode-execute loop that mimics the behavior of a simple CPU, a straightforward approach. Nonetheless, it usually requires the execution of tens (or hundreds) of native instructions to emulate each guest

instruction. On the other hand, dynamic and static translators translate (maps) pieces of guest code into host code and usually obtain greater performance with the cost of being more complex and hard to implement. Because of this, DBT is commonly used on high-performance emulators, such as QEMU [Bellard 2005]. A DBT engine uses two mechanisms to emulate the execution of a binary, one with a fast-start but slow-execution and another with a fast-execution but a slow-start. The former is used to emulate cold (seldom executed) parts of the binary, normally implemented using an interpreter. The latter is used to emulate hot (frequently executed) parts of the code by translating the region of code and executing it natively. A translated region of code normally executes more than 10x faster than an interpreter [Böhm et al. 2011]. It is important to notice that the costs associated with the translation process impacts directly on the final emulation time. As a consequence, DBTs usually employ region formation techniques (RFTs) that try to form and translate only regions of code that the execution speedup (compared to interpretation) pays off the translation time cost.

The majority of a program execution is spent in small portions of code [Smith and Nair 2005], thus DBT engines spend most of their time executing small portions of translated code. Therefore, the quality of these translated regions is crucial to the final performance of a DBT engine. In fact, this is evidenced by the low overhead of Same-ISAs DBT engines [Borin and Wu 2009], also known as binary optimizers, as they always execute code with the same or better quality than the native binary (this happen because same-ISA do not actually impose translations, but only optimizations). Designing and implementing high-performance Cross-ISA DBT engines, on the other hand, is more challenging because the performance of translated code depends heavily on the characteristics of the guest (source) and the host (target) ISA. For instance, the ARM has a conditional execution mechanism that enables instructions to be conditionally executed depending on the state of the status register, however, since x86 does not have this feature, it may require several instructions to mimic this behavior in x86 [Salgado et al. 2017]. Experience has shown that emulating a guest-ISA which is simpler than the target-ISA is normally easier [Auler and Borin 2017].

In the end, implementing a high-performance DBT would be the final proof of concept of whether an ISA is simple to translate, but implementing a DBT is a complex project and a challenge by itself to construct. Another possibility, which we use in this paper, is to implement an SBT engine to translate the binary. An SBT engine translates statically the whole binary at once. SBT is not usually used to emulate binaries in industry, despite being easier to implement than a DBT engine, because an SBT cannot execute all kind of applications. Self-modifying code, code discovery problem and indirect branches, are some problems that cannot be addressed statically [Cifuentes and Malhotra 1996]. However, for the purpose of testing the difficulty of translating code with high-quality, an SBT is enough. This same approach was used by Auler and Borin [Auler and Borin 2017] to test the OpenISA emulation performance.

## 3. RISC-V

In terms of ISA design, RISC-V is reaching a mature and stable state only by now [Waterman et al. 2014]. RISC-V was developed in 2010, but the user-level ISA base and extensions MAFDQ (Multiply/divide, Atomic, single-precision Floating-point, Double-precision floating-point and Quadruple-precision floating-point: the main standard exten-

sions) were frozen only in 2014 [RV-foundation 2018a]. For the privileged ISA, at the time of this writing, it is still a draft, albeit at an advanced stage. For the physical implementations, there are several open-sourced RISC-V CPU designs available. While these open-sourced designs are a great step towards having plenty RISC-V CPU chips available, that is not the case for now, given the research/experimental state of such CPU designs. Therefore, at the present time, there are few platforms that implement the RISC-V architecture. It usually takes some time until hardware implementing a new ISA becomes widely available. Until then, emulation plays a crucial role, because it enables the use of a new ISA while there are no (or few) physical CPUs available for it. The main job of an ISA emulator is to translate guest instructions to host instructions, with the goal of making the host perform a computational work that is close enough to what would be achieved by the guest instructions being executed on the guest platform. As the majority of available hardware is either X86 or ARM based, targeting these ISAs as host platforms for RISC-V emulators seems reasonable. While there are already some RISC-V emulators available for those, e.g., Spike and QEMU, they are unable to achieve near-native performance — as shall be discussed in the next section — which limits their scope by excluding them from use cases where performance plays a major role. Thus, this work aims at designing mechanisms for fast RISC-V emulation. It shows the viability of running RISC-V applications with near-native performance on X86 and ARM-based processors, proving that RISC-V is an easy to translate ISA, at least to these architectures.

## 4. Related Work

According to Auler and Borin [Auler and Borin 2017], it is possible to achieve near-native performance in cross-ISA emulation if the guest architecture is easy to be emulated. They showed this to be possible with OpenISA, an ISA based on MIPS but modified with emulation performance in mind. Using SBT to emulate OpenISA on X86 and ARM, they were able to achieve less than 1.2x of slowdown on most benchmarks. RISC-V has most of the characteristics pointed by the authors to be easy to emulate: it is simple, it hardly uses the status register and it has a small number of instructions, with 66% of them being equivalent to the OpenISA instructions, all indicating that RISC-V is also an easy to emulate ISA. For same-ISA emulation, the best performance achieved is from StarDBT by Borin and Wu [Borin and Wu 2009]: 1.09x slower than native (x86) emulation.

Spike [RV-foundation 2018b], a RISC-V ISA simulator, is considered by the RISC-V Foundation to be their "golden standard" in terms of emulation correctness. As expected from an interpreted simulator, its performance is not very high, although quite higher than other simulators in some cases, varying from 15 to 75 times slower than native on SPECINT2006 benchmarks. This performance is due to several DBT-like optimizations, such as instruction cache, software TLB, and unrolled PC-indexed interpreter loop to improve host branch prediction. ANGEL [RV-foundation 2018b] is a Javascript RISC-V ISA (RV64) Simulator that runs riscv-linux with BusyBox. Our simple run achieved $\approx$ 10 MIPS in Chrome, on an Intel Core i7-2630QM CPU running at 2.0GHz, or about 200 times slower than native.

In the case of RISC-V DBT engines, B. Ilbeyi *et al.* [Ilbeyi et al. 2016] showed that Pydgin can achieve better performance than Spike, by means of more sophisticated techniques, mainly, DBT. Pydgin is able to achieve a performance between 3.3x to 4x slower than native. Clark and Hoult [Clark and Hoult 2017] presented the RV8 emulator

a RISC-V high-performance DBT for x86. Using optimizations such as macro-op fusion and trace formation and merge, RV8 is able to achieve a performance 3.16x slower than native. QEMU [Bellard 2005], a famous DBT with multiple sources and targets, also gained support for RISC-V. QEMU is 7x slower than a native execution and one of its main performance disadvantages comes from floating-point emulation, as its Intermediate Representation (IR) does not have any instruction of that kind and it needs to simulate them by calling auxiliary functions.

## 5. RISC-V SBT

Static translation of a RISC-V binary into a native binary for other architecture, such as ARM and x86, involves several steps. As mentioned earlier, the goal is to translate RISC-V machine instructions into machine instructions for another ISA, that performs equivalent computational work. Modern compilers, such as LLVM/Clang, compile from source to machine code in several, independent, steps. It helps to modularize, organize and decouple the parts that compose a compiler. That is why we chose to take advantage of an existing modular compiler: LLVM 7.0. This enabled us to reuse several parts of it, write only the missing parts and assemble everything to perform the complete translation.

Our Static Binary Translator starts by reading the RISC-V executable file and disassembling each instruction in it, with the help of LLVM libraries. Then, for each RISC-V instruction, our translator emits equivalent, target independent, LLVM Intermediate Representation (IR) instructions or bitcode. This is practically the same as the IR produced by Clang after compiling a source code file. After that, the produced LLVM IR is written to a file, concluding the first translation stage.

The remaining steps are performed with existing software. LLVM tools are used to optimize the IR and to generate assembly code for x86 or ARM. After that, a standard assembler and linker for the target platform, such as GNU as and ld, can be used to produce the native binary for the host architecture. The code generation flows used in our experiments are further detailed in Section 6. Finally, all these steps for SBT are summarized in the diagram from Figure 1.

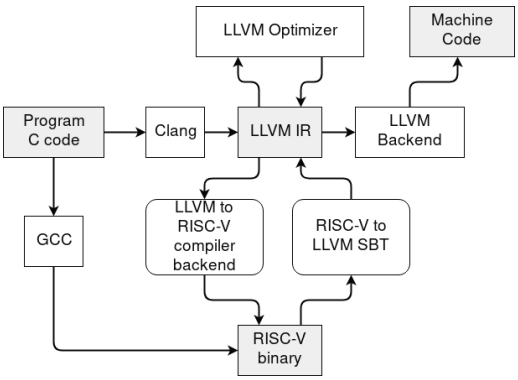



**Figure 1. Our RISC-V SBT Architecture.**

### 5.1. Unlinked Objects as Input

Instead of translating final linked binaries, we choose to translated object binaries before linking to avoid having to deal with some issues. It enables us to translate only the benchmark code, leaving C runtime out. In this way, we avoid C runtime translation quality

from interfering in benchmark performance measurements and also save a considerable amount of work that would be required if the SBT needed to be able to translate all C runtime libraries.

However, with this approach, the translator must now be able to identify C library calls in guest code and forward these to the corresponding ones on native code. This was done by listing all C functions needed by the benchmarks we used, together with their types and arguments and then, at the call site, copying RISC-V registers corresponding to arguments to the appropriate host arguments' locations, as defined by their ABIs.

### 5.2. Register Mapping

Regarding register mapping between architectures during the translation, our SBT implements two techniques:

- **Globals** – RISC-V registers are translated to global variables. The main advantages of this approach is that it is simple and it does not need any kind of inter-function synchronization. The main disadvantage of it, however, is that the compiler is unable to optimize most accesses to global variables.
- **Locals** – RISC-V registers are translated to function's local variables. The main advantage of this approach is that the compiler is able to perform aggressive optimizations on those. The main disadvantage is that the values of these local variables need to be synchronized with those of other functions at function calls and returns, what can impact performance significantly on hot spots. We implement the synchronization by copying local register variables from/to global register variables when entering or leaving functions.

## 6. Experimental Setup and Infrastructure

In order to quantify the performance overhead introduced by the SBT, we compare the performance of benchmarks emulated with SBT against the performance of their native execution. Also, we designed several experiments to investigate the performance overhead on both x86 and ARM platforms and the effect of different compilers on the performance of the SBT. As a consequence, we employed multiple compilation flows in our experiments. These compilation flows are depicted on Figure 2.

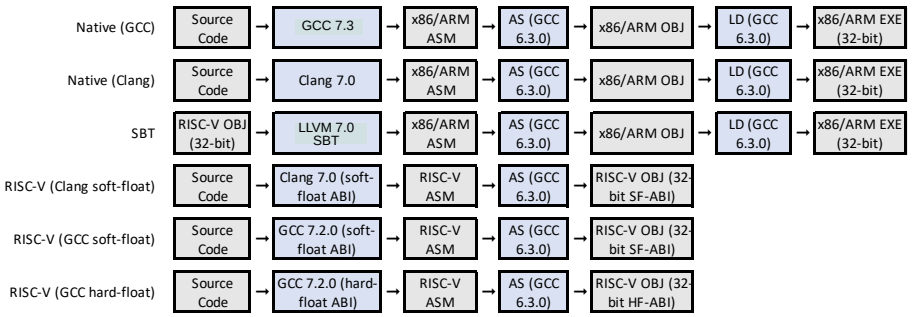



**Figure 2. Code generation flows.**

The first compilation flow, Native (GCC), was used to produce native x86 and ARM binaries using the GCC compiler. The second compilation flow, Native (Clang), was used to produce native x86 and ARM binaries using the Clang compiler. In this case,

the assembly code was generated by Clang 7.0 and the final binary was assembled and linked by GCC 6.3.0. We used this combination because LLVM's assembler and linker do not support RISC-V binaries. However, as we discuss later, differences in libc versions do not matter in our experiments because we factor out time spent in it. Now, in order to measure the performance of our SBT engine, we combine the following flows: three to produce RISC-V binaries (RISC-V OBJ) from benchmarks' source code (Clang soft-float and GCC Soft or Hard float) with another to translate the RISC-V binaries to native code, using our SBT based on LLVM 7.0 (SBT).

To minimize performance differences that may be introduced by using different compiler versions and flags, we have used the same compilers and optimization flags (-O3 was used in all cases) for flows and experiments. Moreover, currently, Clang supports generating only RISC-V assembly code, not the full linked binary. Thus, for all targets, we used the same approach: use Clang (7.0) or GCC (7.3) to compile the source code (C) to ASM and then GCC (6.3) to assemble and link. Furthermore, for x86, the AVX extensions were enabled and, to avoid issues with legacy x86 extended precision (80-bit) floating point instructions, we also used the -mfpmath=sse flag. As for ARM, we targeted the armv7-a processor family, with vfpv3-d16 floating point instructions, as this is a perfect match with Debian 9 distribution for *armhf*.

### 6.1. Measurement Technique

To perform the experiments, after compiling and translating all needed binaries, each one was run 10 times. Their execution times were collected using Linux Perf and summarized by their arithmetic mean and standard deviation (SD). The execution times showed to follow a normal distribution with a very small SD. Thus, as it would not be relevant and in order to have clear graphics, we choose to not present the SD data.

Moreover, we also decided to factor out from the results the time spent on libc functions. We followed the same methodology aforementioned, executing the benchmarks 10 times and calculating the libc portion of the execution time arithmetic mean (percentage of the execution time). The final runtime of each benchmark is then multiplied by this percentage, so that time spent in parts other than the main benchmark code, such as libc, libm, and dynamic loader, are factored out.

### 6.2. GCC vs Clang and Soft vs Hard Float backend

To compile the benchmarks, our initial plan was to use Clang for every target: ARM, RISC-V and x86. However, during the experiments, we found out that Clang's support for RISC-V is still incomplete and considerably behind GCC's. For instance, some of LLVM optimizations need to be performed in collaboration with the target back-end or they may otherwise be skipped. But the major inefficiency we have noticed so far is that LLVM does not support RISC-V hard-float ABI. Although it is able to generate code that makes use of floating point instructions, function arguments are always passed through integer registers and stack, instead of using floating point registers whenever possible. This causes unnecessary copies from floating point registers to integer registers and vice-versa. This is further aggravated by the fact that, on RISC-V 32-bit, there is no instruction to convert a double value to a pair of 32-bit integer registers or to do the opposite conversion; this needs to be done in multiple steps, using the stack. Because of this, we also performed the same experiments using GCC to compile the code, so that we could have

a higher quality RISC-V input code, especially on benchmarks that make heavy use of floating point operations.

As mentioned above, we observed that Clang produced RISC-V code with a considerably worse quality than GCC in some cases, especially on floating point benchmarks. Because of this, for x86, we performed the same experiments with both compilers. For ARM, we used only GCC and hard-float ABI in the experiments, as these gave the best results.

### 6.3. RISC-V Configuration

In our experiments and SBT implementation, we chose to use the RISC-V 32-bit instruction set, mainly to: facilitate comparisons with OpenISA, that is also 32-bit; facilitate translation to ARM 32-bit. The standard RISC-V extensions used by us were: M (integer multiplication/division instructions), F and D (single and double precision floating point instructions). Except for the A (atomic instructions) extension that we left out, these extensions compose the general-purpose RISC-V instructions. The reason for leaving the A extension out is that we have used only single-threaded benchmarks, in which case atomic instructions are not needed.

## 7. Experimental Results

In this section, we present the performance of our RISC-V SBT in terms of slowdown when compared to native execution (GCC RISC-V compared to GCC native and Clang RISC-V compared to Clang native). Hence, the higher the value the worse the emulation performance. A slowdown equal to 1 means that the translated binary is as fast as the native. In all cases, the guest binaries were translated using both the Globals and the Locals translation schemes.

### 7.1. GCC vs Clang

Figure 3 shows the performance of the SBT when emulating RISC-V binaries produced by Clang and GCC soft-float ABI on x86. In general, the performance of the Locals translation scheme produces better code than the Globals one. The only exception is bitcount. In this case, we profiled the code using *perf* and observed that what caused the poor emulation performance with Locals was the register synchronization overhead. This synchronization occurs at bitcount's main loop, when entering and leaving the main benchmark functions through indirect calls, making it harder for the compiler to optimize/inline these. On average, the Locals translation scheme achieved a slowdown with GCC/Clang of 1.20x/1.34x while the Globals scheme produced a slowdown of 2.06x/2.18x when emulating RISC-V code produced with soft-float ABI.

Figure 3 also indicates that in some cases (adpcm-decode, rijndael-decode, and sha) the SBT performs better when emulating code produced by Clang while in others (dijkstra, crc32, adpcm-encode, stringsearch, basicmath, and patricia) it performs better with GCC. Now, for benchmarks that do not use floating point operations (from dijkstra to blowfish-decode), we can see that emulation performance stays close to native performance, at least in Locals mode. Basicmath and FFT performed poorly, but the following figures make it clear that this is due to the lack of a hard-float ABI for RISC-V code in Clang, as these two benchmarks are among the ones that make the heaviest usage of

floating point operations and have their performance improved when using the hard-float ABI.

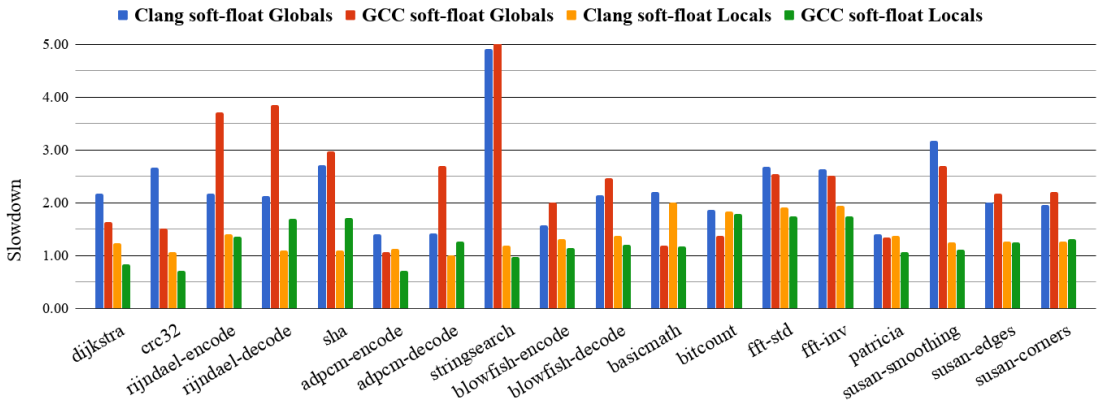



**Figure 3. Slowdown benchmarks compiled with Clang and GCC soft-float RISC-V backends. Both Globals and Locals register mapping results are presented.**

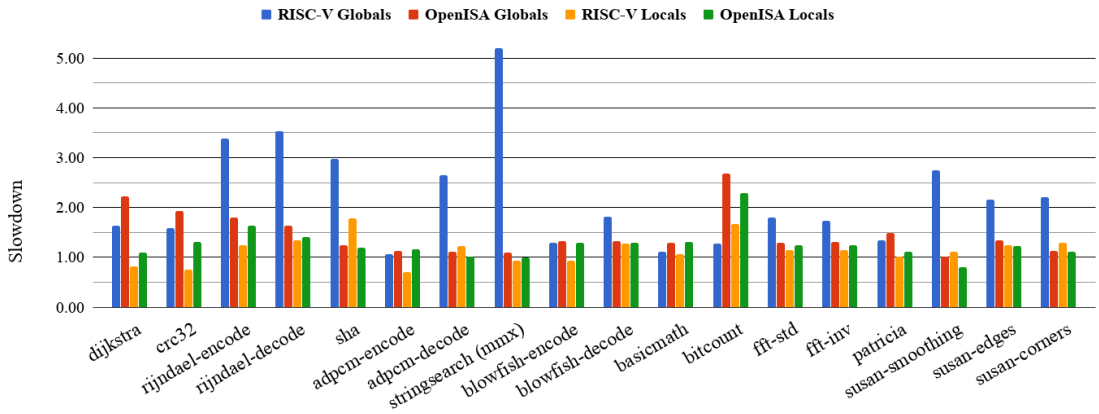
This GCC vs Clang experiment showed that although GCC has a more mature backend than Clang, our SBT performance when translating RISC-V binaries with soft-float ABI produced by both compilers was close. However, as the Clang RISC-V backend does not have, until the date, support to hard-float ABI, we are going to use solely GCC in the next experiments. Moreover, the Locals performance outstands the Globals performance in almost 2-fold, showing the importance of the register mapping approach.

### 7.2. RISC-V vs OpenISA

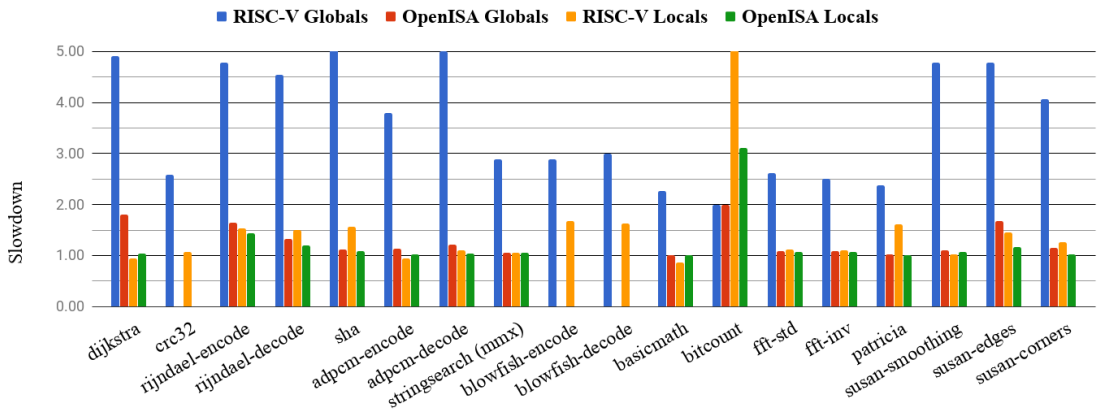
In both Figure 4(a) and 4(b) we compared the performance of our SBT with the ones obtained by the OpenISA SBT translating OpenISA to x86 and ARM [Auler and Borin 2017] using the hard-float ABI. Translating RISC-V for x86, we obtained an average of 1.99x slowdown using Globals mapping and 1.12x using Locals, even better than the one obtained by the OpenISA SBT translating OpenISA to x86, which was 1.41x and 1.23x respectively. The ARM support for our SBT was the last added, so it still needs improvements, but its performance was close to the OpenISA: 3.49x slowdown with Globals and 1.35x with Locals, while with OpenISA it was 1.21x with Globals and 1.17x with Locals.

Figure 4(a) shows the results obtained using GCC hard-float ABI to compile RISC-V binaries and translating them to x86. Besides the results already discussed above for soft-float ABI (Figure 3), we can see a great improvement in FFT and some smaller improvements in basicmath and patricia when using hard-float ABI. Figure 4(b) shows the same binaries translated to ARM. One thing that stands out is that Globals results are much worse for the ARM and this needs further investigation. Moreover, we can see good performance for a handful of benchmarks, similar to x86's, such as dijkstra, crc32, adpcm, stringsearch, basicmath, FFT, and susan when using Locals. Others, on the other hand, suffered a much greater slowdown than x86, such as rijndael, blowfish, bitcount, and patricia. Further analysis is required to understand such differences. Nevertheless,

despite having slightly worse results than when translating to x86, ARM results were only 23% worse than the results from OpenISA.




(a) x86




(b) ARM

**Figure 4. Slowdown for our RISC-V SBT and for the OpenISA SBT. All binaries were compiled using the hard-float ABI.**

### 7.3. Our SBT engine vs DBT engines available

Finally, we compared our best approach, translating binaries generated with hard-float from GCC to x86, with the two most known DBT engines for RISC-V available: RV8 (github 1d4d1ee commit) and QEMU (v2.12.0-835-g360a7809d2-dirty). We can clearly see in the chart from Figure 5 that our RISC-V SBT was the one with the best performance for all tested programs. Our RISC-V SBT achieved, on average, 1.12x slowdown, while RV8 and QEMU achieved 3.16x and 7.07x slowdown, respectively. However, we were not able to run all the benchmarks with RV8 and QEMU. With RV8, during the emulation of some benchmarks, it finished with a segment fault. With QEMU, the benchmarks which manipulated files did not run because some of the syscalls were missing.

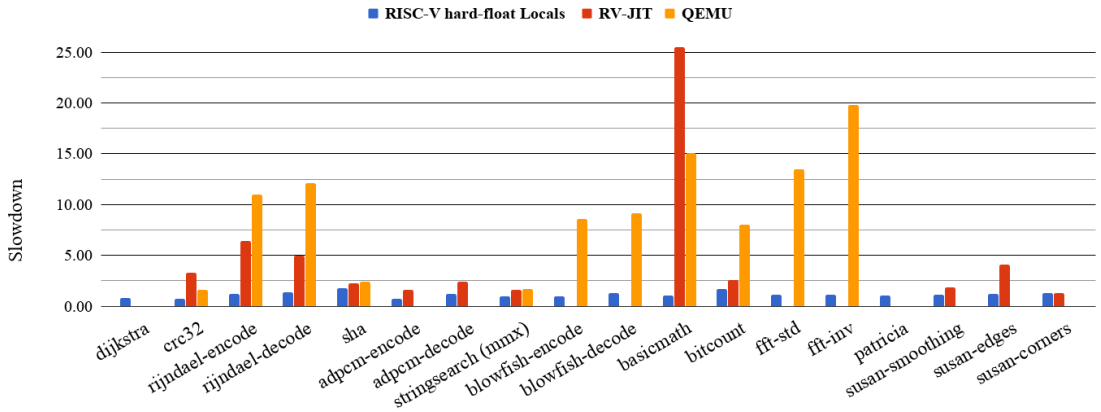



**Figure 5. Slowdown comparison between our RISC-V SBT, RV8 DBT and QEMU-RISCV DBT. All tests emulating RISC-V binaries on a x86 processor.**

Table 1 lists emulation results presented so far in the literature. As can be noticed, other RISC-V emulators suffer from high overheads (more than 3x) when emulating RISC-V code on x86 and ARM platforms. Also, the low overheads achieved by our SBT suggest that it is possible to design and implement high-performance DBTs to emulate RISC-V code on x86 and ARM platforms.

**Table 1. Comparison Between Binary Translator Approaches**

| Name | Guest-ISA | IR | Target-ISA | Technique | Avg. Slowdown |
|---|---|---|---|---|---|
| StarDBT | x86 | None | x86 | DBT | 1.09x |
| OpenISA-SBT | OpenISA | LLVM 3.7 | x86 & ARM | SBT | 1.23x/1.17x |
| ANGEL | RISC-V | None | x86 | Interpreter | 200x |
| Spike | RISC-V | None | x86 | Interpreter | 50x |
| Pydgin | RISC-V | None | x86 | DBT | 4x |
| QEMU | RISC-V | QEMU IR | x86, ARM... | DBT | 7.07x |
| RV8 | RISC-V | None | x86 | DBT | 3.16x |
| **Our SBT** | RISC-V | LLVM 7.0 | x86 & ARM | SBT | **1.12x/1.35x** |

## 8. Conclusion

RISC-V is having the attention globally from the industry and academia. Thus, it is probable that RISC-V is going to have a significant impact in the future of IoT and cloud. However, by now, there is no RISC-V emulation with low overhead available. In this work, we demonstrated that RISC-V is an architecture that enables its code to be translated into high-quality x86 and ARM code. An strong evidence that DBT engines with high-performance can be built for RISC-V. We did this by building a RISC-V static translator which is able to translate RISC-V to x86 and ARM with an execution overhead lower than 12% in the former and 35% in the latter, being the fastest RISC-V emulator presented so far in the literature.

In future work, we plan to modify the OpenISA DBT (OI-DBT) to also support RISC-V, enabling high-performance emulation of RISC-V code with DBT and acceler-

ating its adoption. We also expect to show that RISC-V can be used as an IR for easy software deployment.

## References

Apple (2006). Rosetta: the most amazing software you'll never see. *Archived. Url: http://www.apple.com/asia/rosetta/. Last Accessed 07/2018.*

Auler, R. and Borin, E. (2017). The case for flexible isas: unleashing hardware and software. In *SBAC-PAD*, pages 65–72. IEEE.

Bartholomew, D. (2018). Risc-v qemu. *Github. Url: https://github.com/riscv/riscv-qemu. Last Accessed 07/2018.*

Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46.

Böhm, I., Edler von Koch, T. J., Kyle, S. C., Franke, B., and Topham, N. (2011). Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *ACM SIGPLAN Notices*, volume 46, pages 74–85. ACM.

Borin, E. and Wu, Y. (2009). Characterization of dbt overhead. In *IISWC 2009.*, pages 178–187. IEEE.

Cifuentes, C. and Malhotra, V. M. (1996). Binary translation: Static, dynamic, retargetable? In *icsm*, pages 340–349.

Clark, M. and Hoult, B. (2017). rv8: a high performance risc-v to x86 binary translator.

Dehnert, J. C., Grant, B. K., Banning, J. P., Johnson, R., Kistler, T., Klaiber, A., and Mattson, J. (2003). The transmeta code morphing™ software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO*, pages 15–24. IEEE Computer Society.

Häubl, C. and Mössenböck, H. (2011). Trace-based compilation for the java hotspot virtual machine. In *Proceedings of the 9th PPPJ*, pages 129–138. ACM.

Ilbeyi, B., Lockhart, D., and Batten, C. (2016). Pydgin for risc-v: A fast and productive instruction-set simulator. In *Extended Abstract for Presentation at the 3rd RISC-V Workshop*.

RV-foundation (2018a). Risc-v foundation — instruction set architecture (isa).

RV-foundation (2018b). Risc-v software tools. *Url: https://riscv.org/software-tools. Last Accessed 07/2018.*

Salgado, F., Gomes, T., Pinto, S., Cabral, J., and Tavares, A. (2017). Condition codes evaluation on dynamic binary translation for embedded platforms. *IEEE Embedded Systems Letters*, 9(3):89–92.

Smith, J. and Nair, R. (2005). *Virtual machines: versatile platforms for systems and processes*. Elsevier.

Ta, T., Cheng, L., and Batten, C. (2018). Simulating multi-core risc-v systems in gem5.

Waterman, A., Lee, Y., Patterson, D., and Asanovic, K. (2014). The risc-v instruction set manual. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*.

# Análise de Desempenho de Rede para Aplicações MPI em Infraestruturas SDNs Convergentes para HPC e Big Data

**Alexandre T. Oliveira[1], Alex B. Vieira[1], Antônio Tadeu A. Gomes[2], Artur Ziviani[2]**

[1] Departamento de Ciência da Computação – Universidade Federal de Juiz de Fora (UFJF)

[2]Laboratório Nacional de Computação Científica (LNCC)

alexandre.tavares@ice.ufjf.br,alex.borges@ufjf.edu.br

{atagomes,ziviani}@lncc.br

***Abstract.*** *Extraction of knowledge from large volumes of data imposes challenges to current computing platforms. Although it appears to be natural, the use of HPC platforms by Big Data applications involves the suitability of several of its elements. For instance, the data network infrastructure needs to be efficient and flexible to fit the typical applications of these environments. The SDN paradigm is adequate in this context because of its global vision and its higher level of network programmability. As a consequence, network management is simplified, making it more flexible. In this article, we present an SDN-based communication platform capable of supplying, in a convergent way, HPC and Big Data application requirements. We highlighted the importance of defining the most appropriate network configurations for each traffic profile. We present four routing strategies in a convergent environment and analyze the performance of MPI applications. The results of simulations show that the data throughput in the convergent environment can be up to 39% better, depending on the strategy selected.*

***Resumo.*** *A extração de conhecimento de grandes volumes de dados impõe desafios às plataformas computacionais atuais. Embora pareça ser natural, a utilização de plataformas HPC por aplicações de Big Data envolve a adequação de vários de seus elementos. Por exemplo, a infraestrutura de rede de dados precisa ser eficiente e flexível para se ajustar às aplicações típicas desses ambientes. O paradigma SDN é adequado a esse contexto por sua visão global e seu maior nível de programabilidade da rede. Como consequência, a gerência da rede é simplificada, tornando-a mais flexível. Neste artigo, apresentamos uma plataforma de comunicação baseada em SDN capaz de suprir, de forma convergente, os requisitos de aplicações HPC e Big Data. Evidenciamos a importância da definição das configurações de rede mais adequadas a cada perfil de tráfego. Apresentamos quatro estratégias de roteamento em um ambiente convergente e analisamos o desempenho de aplicações MPI. Os resultados de simulações mostram que a vazão de dados no ambiente convergente pode ser até 39% melhor, dependendo da estratégia selecionada.*

## 1. Introdução

*Big Data* vem se consolidando como um dos maiores fenômenos em termos de modelo de negócios e de ciência em quase todos os domínios [Economist 2010]. Empresas, órgãos

públicos e instituições de pesquisa vêm se beneficiando amplamente das abordagens de *Big Data Analytics*, extraindo informações valiosas a partir dos grandes volumes de dados gerados no âmbito dessas organizações. Naturalmente, manipular esses dados impõe enormes desafios, tanto em termos de gerenciamento de memória quanto de processamento e acesso aos dados.

Plataformas de computação de alto desempenho (*High Performance Computing* – HPC) oferecem grande capacidade de processamento que pode ser aproveitada por uma vasta gama de aplicações, incluindo soluções para tratar grandes volumes de dados. Embora a utilização dessas plataformas HPC por aplicações de *Big Data* pareça ser natural, o que se observa, de fato, é um imenso descasamento entre essas arquiteturas. Realmente, explorar recursos de HPC para atender soluções típicas de *Big Data* demanda o tratamento de inúmeros problemas. A integração entre esses mundos envolve o estudo e a adequação de vários elementos desses sistemas. A infraestrutura de comunicação de dados, por exemplo, precisa ser eficiente e flexível para se adequar aos diferentes perfis das aplicações que operam sobre a rede. No entanto, as redes de dados tradicionais possuem controles complexos, o que as tornam "ossificadas". Logo, o ajuste das redes de comunicação aos requisitos de cada uma dessas aplicações demanda um alto custo administrativo, o que inviabiliza, na prática, o uso dessas infraestruturas de forma convergente e eficiente. Nesse aspecto, há novos paradigmas de redes que podem favorecer a integração dos ambientes HPC e *Big Data*, como as Redes Definidas por Software (*Software-Defined Networks* – SDN).

Nesse cenário, a busca por uma plataforma de comunicação convergente pressupõe o atendimento das exigências de rede de aplicações paralelas que utilizam as APIs típicas de ambientes HPC (e.g., *Message Passing Interface* – MPI), e de aplicações típicas de ambientes *Big Data*, executadas sobre arcabouços MapReduce (e.g., Hadoop, Spark). Espera-se também que essa plataforma implemente mecanismos de classificação de tráfego, de modo que a infraestrutura se ajuste aos respectivos padrões de comunicação através de configurações proativas ou reativas. Assim, a rede torna-se ciente das aplicações, adequando sua configuração apropriadamente a cada perfil de tráfego.

Neste artigo, apresentamos uma plataforma baseada em SDN cujo objetivo é fornecer uma infraestrutura de rede de dados flexível capaz de suprir, de forma convergente, os requisitos de desempenho de aplicações *Big Data* e de HPC. A plataforma atende esses requisitos otimizando a comunicação dos dados provenientes dos processos paralelos. Para tanto, o controlador SDN identifica o tráfego da aplicação e utiliza a estratégia de roteamento mais adequada ao perfil desse tráfego. Dessa maneira, o controlador seleciona o melhor caminho para os pacotes conforme essa estratégia e configura as respectivas regras de encaminhamento nas tabelas de fluxos dos dispositivos de rede. Nessa circunstância, nós propomos quatro algoritmos simples de seleção de rotas para ambientes de rede multicaminhos, característicos de topologias HPC.

Nós simulamos um ambiente SDN Ethernet com uma topologia multicaminhos para analisar o desempenho da rede perante o tráfego gerado por um *benchmark* MPI. A partir de diferentes medições utilizando um cenário específico, nós constatamos diferenças de vazão de até 39% na comparação entre as estratégias de roteamento propostas, ao mesmo tempo que a latência dos pacotes permaneceu estável. Os resultados atuais realçam a influência do processo de seleção de rotas na eficiência da rede e a

importância da definição da melhor configuração de rede para cada perfil de tráfego. Embora ainda não tenham sido realizados testes de forma conjunta, ou seja, considerando ambas as aplicações MPI e *Big Data*, as análises mostram a aplicabilidade do paradigma SDN em infraestruturas Ethernet como solução para maximizar a comunicação em redes de alto desempenho, mitigando problemas comuns dessa tecnologia como variação estatística do retardo e baixa capacidade efetiva. Nesse contexto, este trabalho contribui também para que se abra uma nova perspectiva no estudo de soluções convergentes, no sentido de se ponderar sobre a implantação de infraestruturas Ethernet, significativamente mais baratas, para atender soluções de HPC.

No restante deste artigo, a Seção 2 discute os trabalhos relacionados. Na Seção 3, mostramos alguns dos conceitos de computação avançada. A Seção 4 explora os aspectos da infraestrutura convergente. A descrição da análise de desempenho das aplicações MPI é apresentada na Seção 5. A Seção 6 mostra as conclusões do artigo e os trabalhos futuros.

## 2. Trabalhos Relacionados

Com o amadurecimento do paradigma SDN, o estudo das redes de dados nos ambientes de computação avançada ganhou novos contornos, especialmente na otimização das aplicações típicas de HPC e dos *frameworks Big Data*. Como exemplo, os trabalhos de [Bhatia et al. 2017] e [Alsmadi et al. 2016] empregam SDN para aprimorar o processamento MPI. No primeiro, o controlador utiliza estatísticas de fluxo para atualizar um grafo de topologia de rede usado por um algoritmo de roteamento adaptativo na seleção dinâmica dos caminhos. No segundo, informações de rede obtidas pelo controlador são usadas pelo orquestrador de recursos para selecionar os nós mais adequados ao processamento de cada tarefa.

Há ainda na literatura algumas pesquisas focadas na melhoria dos ambientes típicos de *Big Data*. Por exemplo, [Qin et al. 2015] discutem o processamento de dados em larga escala no Hadoop, uma das mais conhecidas implementações *open source* do modelo de programação MapReduce. No artigo, os autores propõem um agendador de tarefas consciente da largura de banda utilizando o modelo SDN. Essa abordagem, chamada BASS, é capaz não somente de garantir a localidade dos dados a partir de uma visão global, mas também pode eficientemente atribuir tarefas de maneira otimizada.

A convergência entre HPC e *Big Data* é um tópico atual de pesquisa. Em parte, essa convergência é tratada por trabalhos como o de [Ponce et al. 2018], que apresentam uma extensão do modelo de programação paralela e distribuída COMP Superscalar (COMPSs) para o processamento de dados massivos. Nesse estudo, o COMPSs é integrado ao HDFS, sistema de arquivos distribuído bastante utilizado nos cenários de *Big Data*. Há também trabalhos cujos princípios fornecem diversas percepções em torno dessa convergência em camadas mais baixas no contexto de redes, embora não abordem esse tema de forma explícita. Por exemplo, [Webb et al. 2011] formalizam a possibilidade de uso simultâneo de múltiplos mecanismos de roteamento em um *data center*, permitindo às aplicações defini-los e implantá-los conforme suas necessidades. Ainda nesse contexto, [Zhang et al. 2014] propõem um algoritmo ECMP (*Equal-Cost MultiPath*) otimizado por meio de SDN que torna possível ajustar dinamicamente o encaminhamento de fluxos, com o objetivo de melhorar a vazão nas redes de *data center*.

De forma geral, os trabalhos que exploram as capacidades de SDN nos ambientes

HPC e *Big Data* o fazem de maneira independente, como evidenciam os artigos de [Bhatia et al. 2017], [Alsmadi et al. 2016] e [Qin et al. 2015]. Nossa proposta, por sua vez, tira proveito da flexibilidade proporcionada pelo paradigma SDN para prover uma infraestrutura de comunicação verdadeiramente convergente e, acima de tudo, eficiente. Além disso, nossa proposta baseia-se em um modelo onde a rede é ciente da aplicação, com a rede ajustando-se aos requisitos das aplicações, em contraponto aos estudos de [Alsmadi et al. 2016] e [Qin et al. 2015], que propõem soluções sob um ponto de vista no qual a aplicação se torna consciente da rede, ou seja, a aplicação se adapta às condições da rede. Por fim, no contexto da definição dos métodos de roteamento mais apropriados a cada padrão de comunicação, fundamental na proposta de convergência delineada neste artigo, nossa análise de desempenho considera o perfil de tráfego das aplicações MPI, com foco na plataforma convergente, ao contrário dos artigos de [Webb et al. 2011] e [Zhang et al. 2014], que avaliam padrões de tráfego genéricos.

## 3. Ecossistemas de Computação Avançada

Esta seção apresenta alguns dos conceitos por trás dos ecossistemas de computação avançada, incluindo suas arquiteturas e modelos de programação paralela.

### 3.1. Arquiteturas Paralelas

Sistemas que demandam grande poder computacional, como aplicações científicas e até aplicações/ferramentas de negócios, utilizam computação paralela para processarem seus dados no menor tempo possível. Sistemas paralelos são criados combinando múltiplos elementos de processamento em um único sistema maior. Desde a metade dos anos 1990, tecnologias como *multithreading* e *multicore* tornaram esses sistemas amplamente disponíveis. A maioria dos sistemas paralelos modernos encaixam-se na arquitetura MIMD (*Multiple Instruction, Multiple Data*), tipicamente classificada de acordo com a organização de memória: compartilhada e distribuída [Mattson et al. 2004].

Em sistemas de memória compartilhada, processos compartilham um único espaço de memória. Esses sistemas também são classificados em SMP (*symmetric multiprocessors*), NUMA (*nonuniform memory access*) e sua variação ccNUMA (*cache-coherent* NUMA). Sistemas de memória distribuída são aqueles onde cada processo tem seu próprio espaço de endereçamento e a comunicação com os demais processos ocorre mediante o envio e o recebimento de mensagens (*message passing*). Eles são tradicionalmente divididos em MPP (*massively parallel processors*) e *clusters*, que são sistemas mais baratos e, portanto, cada vez mais comuns. Arquiteturas paralelas também podem se apresentar como sistemas híbridos e *grids* [Mattson et al. 2004].

As atuais arquiteturas de computação de alto desempenho (HPC) possuem a tendência de migrar dos tradicionais sistemas SMP e MPP para sistemas de *cluster* de memória distribuída, com nós de computação conectados através de interconexões de alta largura de banda e baixa latência [Date et al. 2016]. As redes desses *clusters* são implantadas com diversas tecnologias e sobre topologias físicas multicaminhos como *fat-tree* e Torus, que oferecem ao mesmo tempo desempenho e redundância. Nesses ambientes, os nós compartilham o acesso aos dados através de um sistema de arquivos distribuído como, por exemplo, o sistema de arquivos paralelos *open source* Lustre.

No ecossistema *Big Data*, os dados a serem processados, a princípio, não cabem na memória principal de um único computador. Assim, o sistema de *cluster* de servidores

é visto como a plataforma padrão para esses ambientes [Porto 2017]. Suas aplicações típicas executam sob um modelo onde os arquivos são distribuídos pelos nós e os processos alocados a esses nós, através do qual se busca minimizar a transferência de dados. Para tanto, esses ambientes utilizam tradicionalmente o sistema de arquivos distribuídos HDFS (*Hadoop Distributed File System*). Considera-se que esses sistemas se encaixam em uma arquitetura sem compartilhamento, onde a tecnologia de rede predominante é a Ethernet.

### 3.2. Programação Paralela

Em programação paralela, cada arquitetura tem um modelo de programação apropriado. Os modelos mais comuns são baseados nas arquiteturas de memória compartilhada, de memória distribuída com passagem de mensagem, ou em uma combinação híbrida das duas [Mattson et al. 2004]. O OpenMP é um dos principais ambientes de programação para as arquiteturas de memória compartilhada, sendo frequentemente usado para adicionar paralelismo ao código sequencial, sobretudo em arquiteturas SMP.

O MPI é o modelo de programação paralela padrão para arquiteturas de memória distribuída, especialmente para os ambientes HPC. Por definição, MPI é uma especificação de interface de bibliotecas de passagem de mensagem. Nesse modelo, um processo empacota informação em uma mensagem e a envia a um outro processo. MPI possui um conjunto de APIs de alto nível que fornece rotinas para gerenciamento de processos e operações de comunicação coletiva. Criado nos anos 1990, sua especificação mais recente é a versão 4.0. Atualmente, MPI possui implementações *open source* como OpenMPI e MPICH, e implementações proprietárias como Intel MPI e Bull MPI.

Nos ambientes *Big Data*, i.e. em arquiteturas sem compartilhamento, é utilizado o modelo de programação funcional MapReduce. Nele, usuários especificam a computação em termos de funções *map* e *reduce*. A função *map* processa cada item do conjunto de entrada, enquanto a função *reduce* processa um conjunto de elementos da entrada. Com base nessa programação, o sistema de tempo de execução subjacente automaticamente paraleliza a computação através do *cluster*. A partir do modelo MapReduce, surgiram os principais *frameworks* de computação intensiva de dados, entre eles o Hadoop, o Spark, e mais recentemente o Flink, todos atualmente mantidos pela fundação Apache.[1]

## 4. Infraestrutura de Comunicação Convergente

A proposta de plataforma de comunicação convergente é concebida sobre uma arquitetura HPC baseada em SDN capaz de atender, além das próprias aplicações HPC, aplicações típicas de *Big Data*. Essas aplicações são executadas em paralelo pelos nós da arquitetura. Esses nós trocam dados/mensagens através da rede. O controlador SDN, por sua vez, procura otimizar a comunicação na rede identificando o tráfego das aplicações e aplicando as estratégias de roteamento mais adequadas a cada perfil de tráfego. Por fim, a rede torna-se consciente da aplicação.

O termo SDN refere-se a uma arquitetura de rede fundamentada em quatro pilares: (i) desacoplamento dos planos de dados e de controle; (ii) decisões de encaminhamento baseadas em fluxos; (iii) transferência da lógica de controle para um

[1] https://www.apache.org

elemento controlador externo logicamente centralizado; e (iv) programação da rede através de aplicações de *software* executando no topo do modelo [Kreutz et al. 2015]. Conforme mostra a Figura 1, a interface entre o controlador SDN e os dispositivos da infraestrutura de rede (*southbound* API) é fornecida por protocolos seguros como o OpenFlow [McKeown et al. 2008]. O OpenFlow é considerado um padrão *de facto*, sendo utilizado pela maioria das plataformas atuais. A *northbound* API não é padronizada ainda. Assim, em um ambiente SDN típico, os elementos de encaminhamento de dados são habilitados com o OpenFlow, através dos quais podem ser gerenciados por um controlador compatível. Uma implementação do protocolo bastante comum é o Open vSwitch.

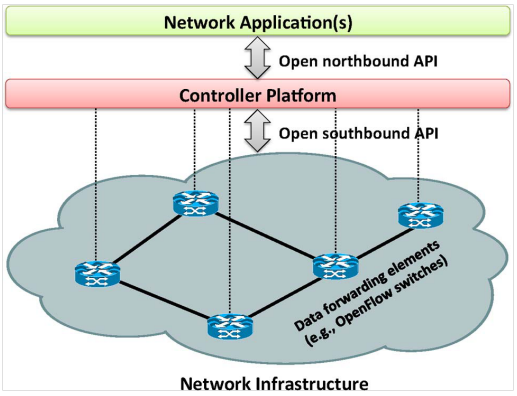



**Figura 1. Visão simplificada de uma arquitetura típica SDN [Kreutz et al. 2015].**

Na nossa concepção, a identificação do tráfego pode seguir tanto uma abordagem proativa quanto reativa. No modo proativo, as aplicações informam suas características ao controlador da rede através de APIs de programação disponibilizadas pela plataforma, o que exige modificações nas aplicações. No modo reativo, a rede toma ciência dos requisitos das aplicações de forma transparente. Nesse caso, o controlador efetua o reconhecimento de assinaturas, ou seja, de padrões explícitos dessas aplicações como, por exemplo, os números de portas dos protocolos da camada de transporte dos pacotes. O controlador pode também realizar a classificação do tráfego mediante metodologias baseadas em inspeção profunda de pacotes ou aprendizado de máquina, embora esses métodos sejam mais custosos computacionalmente [Qazi et al. 2013].

A escolha da estratégia de roteamento tem papel fundamental na rede. Definir os caminhos mais adequados a cada padrão de tráfego pode implicar em aumento de vazão e diminuição da latência dos pacotes. Para cada tipo de classificação, decidir corretamente a rota pode fazer com que se eleve o desempenho da plataforma convergente. Por exemplo, aplicações que utilizam o modelo MapReduce possuem um padrão de comunicação bem definido, com destaque para a movimentação dos dados entre os nós que computam as funções *map* e os que computam as funções *reduce* (fase de "embaralhamento"). No modelo MPI, o padrão depende da aplicação, com operações de comunicação coletiva geralmente prevalecendo sobre operações ponto-a-ponto. Apesar da dependência da aplicação, o sistema de passagem de mensagem apresenta um perfil peculiar.

Considerando, portanto, uma plataforma convergente SDN sobre um *cluster* com topologia multicaminhos, nós propomos quatro mecanismos de roteamento a serem utilizados na infraestrutura de rede desses ambientes. Eles são idealizados com a

finalidade de maximizar a comunicação dos dados ao mesmo tempo que tentam minimizar o custo computacional. Em termos gerais, as políticas de roteamento são baseadas em algoritmos comumente utilizados em redes de comutação de pacotes com múltiplas rotas.

O **Algoritmo 1**, aqui referenciado como **"stp"**, seleciona sempre o mesmo caminho (por exemplo, o primeiro) dentre o conjunto de caminhos mínimos disponíveis entre uma origem e um destino. Ele é inspirado no protocolo *spanning tree* no sentido de que ignora as demais rotas, considerando-as como redundantes. Seu custo computacional é baixo, porém desperdiça os múltiplos caminhos da topologia. Um exemplo de pseudocódigo para o algoritmo "stp" é apresentado a seguir:

**Algoritmo 1:** "stp"

```
   Entrada: Conjunto de caminhos mínimos
1  início
2  |  caminho mínimo ← primeiro caminho
3  fim
4  retorna caminho mínimo
```

O **Algoritmo 2**, aqui referenciado como **"traffic"**, seleciona o caminho mínimo menos congestionado entre uma origem e um destino. Para tal, dentre o conjunto de caminhos mínimos, é calculada a menor taxa de tráfego total instantânea dos seus enlaces. Seu custo computacional é alto, entretanto usufrui das rotas com maior largura de banda disponível. A seguir, é mostrado um pseudocódigo para esse algoritmo:

**Algoritmo 2:** "traffic"

```
   Entrada: Conjunto de caminhos mínimos
1  início
2  |  caminho mínimo ← primeiro caminho
3  |  taxa total menor ← ∞
4  |  para cada caminho ∈ caminhos mínimos faça
5  |  |  taxa do caminho ← 0
6  |  |  para cada comutador ∈ ao caminho faça
7  |  |  |  taxa do comutador ← bytes enviados/recebidos na porta de saída
8  |  |  |  taxa do caminho ← taxa do caminho + taxa do comutador
9  |  |  fim
10 |  |  se taxa do caminho ≤ taxa total menor então
11 |  |  |  taxa total menor ← taxa do caminho
12 |  |  |  caminho mínimo ← caminho
13 |  |  fim
14 |  fim
15 fim
16 retorna caminho mínimo
```

O **Algoritmo 3**, aqui referenciado como **"ecmp"**, seleciona o caminho mínimo através de uma função de *hash*. A chave da função pode ser formada por uma n-tupla composta por campos de cabeçalho do fluxo de pacotes — por exemplo, (IP de origem, IP de destino, ID do protocolo) — ou pelo próprio valor do campo identificador do fluxo que o SDN fornece. Para mapear a chave a um índice associado ao caminho, pode ser utilizada uma função simples como a de módulo. Seu custo computacional é baixo e os

multicaminhos são bem aproveitados, porém o uso de funções de *hash* não perfeitas pode gerar colisões e, consequentemente, caminhos ociosos. Esse algoritmo é inspirado no algoritmo ECMP [Thaler and Hopps 2000]. Um pseudocódigo é exibido adiante:

**Algoritmo 3:** "ecmp"

```
   Entrada: Conjunto de caminhos mínimos
1  início
2  |  índice do caminho ← ID do fluxo % nº de caminhos mínimos
3  |  caminho mínimo ← caminhos mínimos [índice do caminho]
4  fim
5  retorna caminho mínimo
```

Por fim, o **Algoritmo 4**, aqui referenciado como **"isolated"**, é uma variação do Algoritmo 3. Ele seleciona o caminho mínimo por meio de uma função de *hash* e o "isola" temporariamente, de forma que a rota permaneça exclusiva para o fluxo, ficando indisponível para seleção pelos demais fluxos. Seu custo computacional também é baixo. Um pseudocódigo para esse algoritmo é apresentado a seguir:

**Algoritmo 4:** "isolated"

```
   Entrada: Conjunto de caminhos mínimos
1  início
2  |  enquanto ∃ caminhos mínimos faça
3  |  |  índice do caminho ← ID do fluxo % nº de caminhos mínimos
4  |  |  caminho mínimo ← caminhos mínimos [índice do caminho]
5  |  |  se caminho mínimo ∉ caminhos isolados então
6  |  |  |  caminho mínimo ∪ caminhos isolados
7  |  |  |  vai para 14
8  |  |  fim
9  |  |  senão
10 |  |  |  caminhos mínimos ← caminhos mínimos - caminho mínimo
11 |  |  fim
12 |  fim
13 fim
14 retorna caminho mínimo
```

A proposta desses algoritmos baseou-se, essencialmente, em trabalhos publicados no domínio das redes de *data center*, como os artigos de [Webb et al. 2011] e [Zhang et al. 2014], descritos na Seção 2. Inclusive, [Webb et al. 2011] apontam que aplicações MapReduce adequam-se melhor a estratégias de seleção de caminhos que privilegiam a largura de banda disponível.

## 5. Análise de Desempenho de Aplicações MPI

No contexto da avaliação da plataforma de comunicação convergente entre HPC e *Big Data*, nós descrevemos nesta seção os detalhes de uma análise de desempenho de aplicações MPI, considerando os algoritmos descritos na Seção 4.

### 5.1. Cenário Considerado

A Figura 2 apresenta o cenário de avaliação considerado neste trabalho. Ele representa uma arquitetura HPC SDN formada por um *cluster* de servidores de memória distribuída

com uma topologia física multicaminhos *fat-tree 4-port, 3-tree*, isto é, com *switches* de quatro portas e uma árvore de três níveis (acesso, distribuição e núcleo). A rede L2 possui 20 *switches* ($s1..s20$) gerenciados por um controlador SDN (não exibido na Figura 2). O cenário ainda é composto por 16 *hosts* ($h1..h16$) e um servidor (srv1), o qual exerce o papel de servidor de arquivos compartilhado pelos nós. Nessa figura também é possível observar a representação dos quatro PODs (*Point of Delivery*).

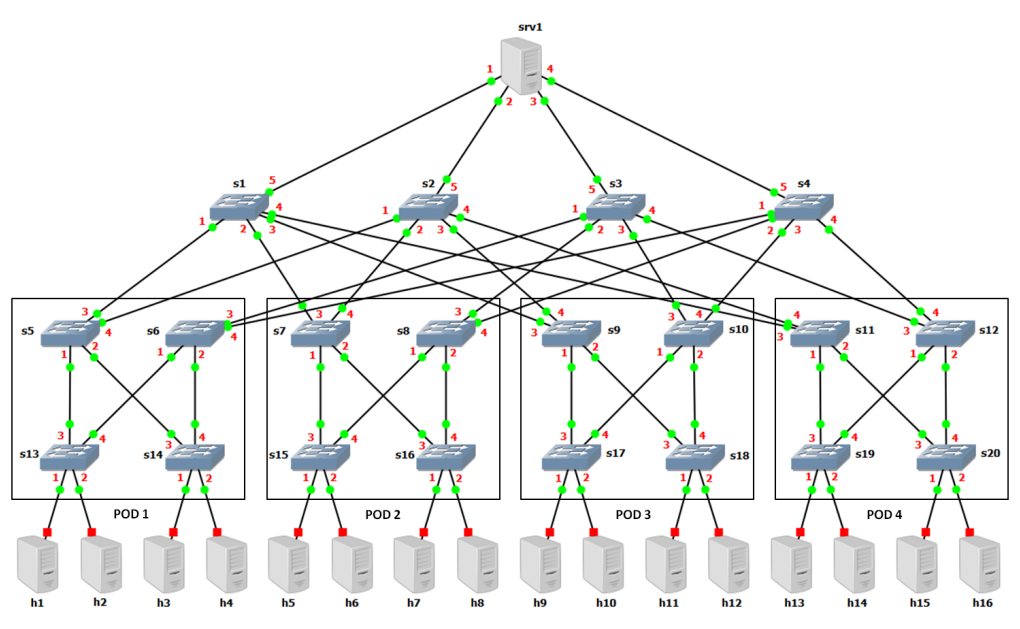



**Figura 2. Cenário de avaliação considerado.**

Esse cenário retrata um ambiente HPC reduzido mas, ainda assim, capaz de capturar todos os aspectos relevantes à nossa avaliação. Nele, as aplicações MPI são executadas em paralelo pelos nós ($h1..h16$). A comunicação entre os processos dispõe, quase sempre, de múltiplos caminhos entre os pares de nós. O número de caminhos mínimos entre esses pares varia em relação às suas posições na rede: nós conectados ao mesmo *switch* de acesso não possuem caminhos redundantes; nós conectados a *switches* de acesso diferentes dentro do mesmo POD dispõem de dois caminhos mínimos entre eles; nós conectados a PODs distintos possuem quatro caminhos mínimos diferentes.

## 5.2. Metodologia de Avaliação

O ambiente de rede descrito na Seção 5.1 foi implementado no emulador Mininet versão 2.3.0d1 com os comutadores executando o Open vSwitch 2.0.2 e enlaces limitados a 1 Gbps. Como controlador SDN foi empregado o POX *carp* utilizando o OpenFlow 1.0. Todo o ambiente foi montado em um servidor 64 bits com duas CPUs Intel Xeon E5520 2,27 GHz, 16 núcleos, memória RAM de 48 GB, HD de 1 TB e SO Ubuntu 14.10.

O modelo de testes foi preparado visando avaliar os quatro algoritmos previamente propostos. Um *benchmark* MPI foi executado em paralelo a partir do *host* $h1$ utilizando nós de diversos PODs. A ferramenta utilizada foi o HPCC (HPC *Challenge Benchmark*)[2] versão 1.5.0. A *suite* HPCC consiste de basicamente sete testes que incluem diferentes

[2]http://icl.cs.utk.edu/hpcc

métricas de desempenho. Entre eles, nós focamos em um conjunto de testes baseado no b_eff (*effective bandwidth benchmark*), o qual mede a vazão e a latência de vários padrões de comunicação simultâneos.

Para o projeto das simulações foram definidas duas variáveis de resposta, ambas medidas sob o ponto de vista do *host* $h1$: (i) vazão utilizada na comunicação; e (ii) latência dos pacotes. Tais métricas são importantes no âmbito das redes de ambientes HPC, as quais exigem interconexões de alta largura de banda (grande vazão) e baixa latência. Para as simulações paralelas, foram utilizados três tamanhos de grupos de *hosts*: (a) 4 nós ($h1$, $h3$, $h5$ e $h7$ – PODs 1 e 2); (b) 8 nós (*hosts* ímpares – todos os PODs); e (c) 16 nós (todos os *hosts*). A partir desse planejamento foram executadas três sessões de testes, onde para cada sessão somente um grupo de nós foi usado. Cada configuração foi executada utilizando os 4 algoritmos propostos: (1) "stp"; (2) "traffic"; (3) "ecmp"; e (4) "isolated". Foram utilizadas as configurações padrão do HPCC: medições de vazão com mensagens de 2000000 *bytes*; medições de latência com mensagens de 8 *bytes*; e comunicação lógica por anel realizada em ambas as direções. Em cada rodada de testes foram efetuadas 30 execuções do HPCC, com intervalos de 30 segundos entre elas. Os resultados são médias obtidas com intervalos de confiança de 95%.

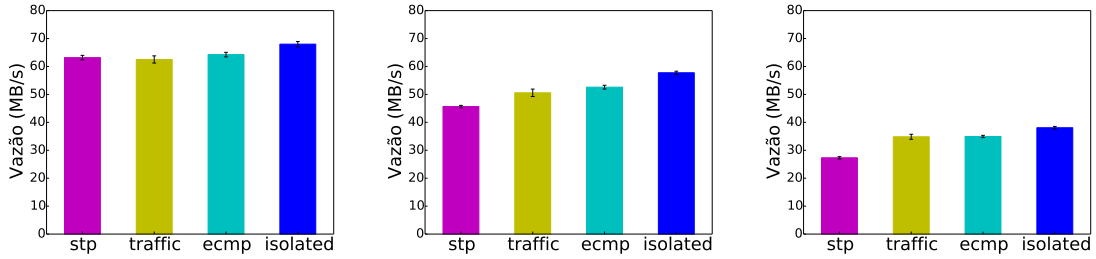
### 5.3. Resultados

A Figura 3 apresenta, para cada grupo de nós, a vazão média e os respectivos intervalos de confiança em função do algoritmo escolhido[3]. Podemos notar que há uma clara influência do processo de seleção de caminhos no desempenho da comunicação MPI, para todos os cenários avaliados. Essa influência é menos acentuada na sessão de testes com 4 nós (Figura 3(a)), com os algoritmos "stp" e "traffic" apresentando, inclusive, taxas equivalentes de vazão (63,17 MB/s e 62,51 MB/s, respectivamente). Os algoritmos "ecmp", com 64,25 MB/s, e "isolated", com 67,99 MB/s, alcançaram melhores resultados nessa sessão. O ganho do melhor algoritmo em relação ao pior foi de cerca de 9%, o que sugere uma correlação direta entre o número de mensagens na rede e os possíveis benefícios do uso de estratégias de roteamento distintas.

De fato, os gráficos das sessões de testes com 8 e 16 nós mostram mais claramente a relevância da definição de um método de roteamento apropriado na melhoria do desempenho da rede. Como indica a Figura 3(b), na sessão de testes com 8 nós, o ganho relativo do "isolated" (vazão de 57,75 MB/s) em comparação com o "stp" (vazão de 45,65 MB/s) foi superior a 26%. Ainda, os algoritmos "traffic", com 50,57 MB/s, e "ecmp", com 52,59 MB/s, também apresentaram melhores taxas de vazão em relação ao "stp". A sessão de testes com 16 nós (Figura 3(c)), embora apresente menores médias, revelou diferenças expressivas de taxas de vazão entre os algoritmos "stp" (27,31 MB/s) e "isolated" (38,07 MB/s), o que representa uma melhoria de aproximadamente 39%. Nesse mesmo cenário, os algoritmos "traffic" e "ecmp" também proporcionaram melhores desempenhos, com taxas de vazão de 34,83 MB/s e 34,94 MB/s, respectivamente.

Por fim, avaliamos a latência fim-a-fim dos pacotes em todas as sessões de testes. A Figura 4 mostra, para cada conjunto de nós, as respectivas funções de distribuição de probabilidade acumulada em relação aos algoritmos utilizados. Os gráficos demonstram

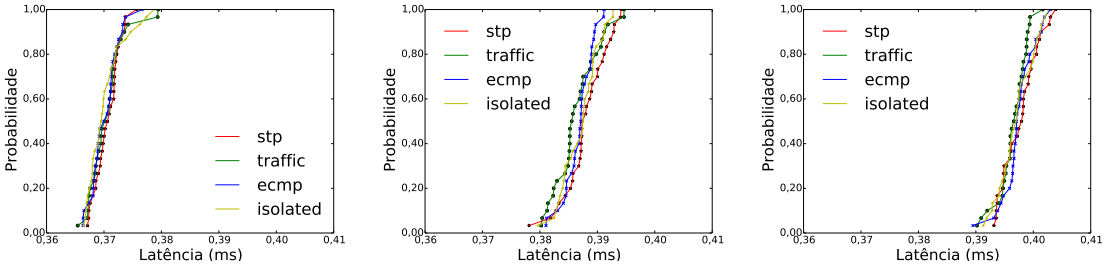
[3]Os dados resultantes das simulações e os códigos-fontes dos protótipos dos componentes de *software* desenvolvidos para a avaliação estão disponíveis em https://github.com/netlabufjf/hpc-sdn




(a) 4 Nós (b) 8 Nós (c) 16 Nós

**Figura 3. Médias das taxas de vazão na comunicação MPI.**

que o desempenho da rede permaneceu estável no que se refere a esse parâmetro, mesmo diante do uso de métodos de roteamento distintos. Realmente, os valores médios apresentaram-se muito próximos entre si: 0,37 ms (4 nós); 0,39 ms (8 nós); e 0,40 ms (16 nós). Os resultados também evidenciam que não houve contenção de rede no cenário.




(a) 4 Nós (b) 8 Nós (c) 16 Nós

**Figura 4. Distribuição de latência dos pacotes.**

Em suma, o algoritmo de roteamento "isolated", que utiliza a estratégia de isolamento de caminhos, apresentou o melhor desempenho em todos os cenários.

## 6. Conclusões e Trabalhos Futuros

Neste artigo, apresentamos a concepção de uma plataforma de comunicação baseada em SDN cujo propósito é suprir, de forma integrada, os requisitos de aplicações típicas de ambientes HPC e *Big Data*. Evidenciamos a importância da definição das configurações de rede mais adequadas aos perfis de tráfego e, por isso, apresentamos quatro estratégias de roteamento. Nós simulamos a arquitetura proposta e a avaliamos a partir de um *benchmark* bem conhecido. Os resultados mostram que a estratégia de isolamento de caminhos de rede apresenta taxas de vazão superiores em até 39%, dependendo da configuração de rede selecionada. Nós concluímos, portanto, que a exploração das capacidades de SDN pode maximizar a comunicação em uma infraestrutura de rede convergente, tornando-a mais eficiente.

Como trabalhos futuros, nós pretendemos replicar a análise deste estudo no contexto das aplicações *Big Data*, inclusive ratificando os resultados aqui encontrados em cenários de avaliação maiores. Mais adiante, nós esperamos desenvolver os mecanismos de classificação de tráfego de rede para ambas as aplicações. Planeja-se, ainda,

avaliar a utilização de ferramentas de "fatiamento" de rede como solução alternativa à implementação da plataforma convergente.

## Agradecimentos

Os autores agradecem o apoio de CAPES, CNPq, FAPEMIG, FAPERJ e FAPESP.

## Referências

Alsmadi, I., Khamaiseh, S., and Xu, D. (2016). Network parallelization in HPC clusters. In *Proc. of the IEEE CSCI*.

Bhatia, S., Sinha, Y., Chalapathi, G., and Kumar, R. (2017). MPI aware routing using SDN. *Poster Presented at the 26th International Symposium on High Performance Parallel and Distributed Computing (HPDC)*.

Date, S., Abe, H., Khureltulga, D., Takahashi, K., Kido, Y., Watashiba, Y., U-chupala, P., Ichikawa, K., Yamanaka, H., Kawai, E., and Shimojo, S. (2016). SDN-accelerated HPC infrastructure for scientific research. *International Journal of Information Technology*, 22(1).

Economist (2010). The data deluge. *Special Supplement*.

Kreutz, D., Ramos, F., Veríssimo, P., Rothenberg, C., Azodolmolky, S., and Uhlig, S. (2015). Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76.

Mattson, T., Sanders, B., and Massingill, B. (2004). *Patterns for Parallel Programming*. Pearson Education.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74.

Ponce, L., Santos, W., Meira-Jr, W., and Guedes, D. (2018). Extensão de um ambiente de computação de alto desempenho para o processamento de dados massivos. In *Proc. of the SBRC*.

Porto, F. (2017). Algoritmos e modelos de programação em big data, XXXVI Jornada de Atualização em Informática (JAI). In *Proc. of the SBC Congress*.

Qazi, Z., Lee, J., Jin, T., Bellala, G., Arndt, M., and Noubir, G. (2013). Application-awareness in SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):487–488.

Qin, P., Dai, B., Huang, B., and Xu, G. (2015). Bandwidth-aware scheduling with SDN in Hadoop: A new trend for big data. *IEEE Systems Journal*.

Thaler, D. and Hopps, C. (2000). Multipath issues in unicast and multicast next-hop selection. RFC 2991, RFC Editor. `http://www.rfc-editor.org/rfc/rfc2991.txt`. Accessed: July, 2018.

Webb, K., Snoeren, A., and Yocum, K. (2011). Topology switching for data center networks. *Hot-ICE*, 11.

Zhang, H., Guo, X., Yan, J., Liu, B., and Shuai, Q. (2014). SDN-based ECMP algorithm for data center networks. In *Proc. of the IEEE ComComAp*.

# Suporte ao Processamento Paralelo e Distribuído em uma DSL para Visualização de Dados Geoespaciais

**Endrius Ewald, Adriano Vogel, Cassiano Rista, Dalvan Griebler, Isabel Manssour, Luiz Gustavo Fernandes**

[1] Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)

Email:adriano.vogel@acad.pucrs.br

***Abstract.*** *The amount of data generated worldwide related to geolocalization has exponentially increased. However, the fast processing of this amount of data is a challenge from the programming perspective, and many available solutions require learning a variety of tools and programming languages. This paper introduces the support for parallel and distributed processing in a DSL for Geospatial Data Visualization to speed up the data pre-processing phase. The results have shown the MPI version with dynamic data distribution performing better under medium and large data set files, while MPI-I/O version achieved the best performance with small data set files.*

***Resumo.*** *Tecnologias de geolocalização tem gerado uma grande quantidade de dados. No entanto, o processamento desse volume de dados não é uma tarefa trivial, pois os métodos de geolocalização exigem a aprendizagem de várias ferramentas ou linguagens de programação. Esse artigo apresenta o suporte ao processamento paralelo e distribuído para uma DSL de visualização de grandes volumes de dados geoespaciais. O sistema aproveita o processamento distribuído para acelerar o pré-processamento de dados. A versão com distribuição dinâmica dos dados (MPI-D) apresentou os melhores resultados com arquivos de dados grandes e médios. Enquanto isso, com arquivos de dados pequenos, o desempenho foi superior na versão MPI-I/O.*

## 1. Introdução

Nos últimos anos, ocorreu um crescimento significativo no volume de dados digitais gerados em todo o planeta. Em dezembro de 2012, um estudo realizado pelo IDC (*International Data Corporation*) [IDC 2012] estimou o tamanho do universo digital em cerca de 2.837 exabytes e com previsão de crescimento para incríveis 40,000 exabytes até 2020. Então, em 2020, de acordo com o IDC, o universo digital seria capaz de disponibilizar mais de 5 terabytes para cada pessoa do planeta. Esse estudo demonstra que estamos vivendo na era dos dados, ou seja, a era do *big data*.

Essa situação tem gerado demandas por novas técnicas e ferramentas que transformem os dados armazenados e processados em conhecimento. Nesse contexto, o uso e o aprimoramento das técnicas de visualização de dados geoespaciais surgem como uma alternativa. As técnicas de visualização usualmente são aplicadas à informações geoespaciais, usando Sistemas de Informação Geográfica (SIG), bibliotecas de programação e *frameworks*. As técnicas de visualização de dados geoespaciais permitem a visualização de variáveis associadas a uma localização espacial, tais como a população e o índice de

qualidade de vida de diferentes cidades, ou as vendas de uma empresa em uma região. Entre as técnicas de análise de dados geoespaciais, a visualização de dados se destaca por auxiliar os usuários a obterem informações rapidamente [Kraak and Ormeling 2011].

É importante destacar que as ferramentas disponíveis atualmente não fornecem as abstrações necessárias para a etapa de pré-processamento do *pipeline* de visualização [Moreland 2013]. Assim, mesmo que a visualização de dados ofereçam muitos benefícios, sua geração continua sendo um desafio [Zhang et al. 2015]. Os usuários têm dificuldades em lidar com a grande quantidade de dados, uma vez que exige custos elevados de processamento e um grande esforço de programação para manipular os dados brutos e o suporte ao paralelismo.

O pré-processamento tem um papel fundamental para a verificação de dados e erros, identificação de inconsistências e possíveis incompletudes. Na literatura (Seção 2), o pré-processamento de grandes quantidades de dados é considerado um problema de desempenho e citado em diferentes trabalhos. Dessa forma, o uso de arquiteturas de processamento paralelo e distribuído aparecem como uma alternativa capaz de atenuar esse problema. Isso é possível, graças ao uso de técnicas de programação paralela, que permitem a aceleração do processamento para as aplicações através dessas arquiteturas de alto desempenho. No entanto, esta não é uma tarefa trivial, pois requer habilidades específicas em ferramentas, metodologias e modelagem [Rünger and Rauber 2013].

Este trabalho avalia estratégias para o pré-processamento distribuído, buscando melhorar a experiência de criação de visualizações de dados geoespaciais, reduzindo o tempo necessário para o pré-processamento dos dados e a implementação da visualização de dados. O objetivo é permitir o processamento paralelo e distribuído na DSL GMaVis, que até o presente momento, era capaz de realizar apenas o processamento paralelo em sistemas multi-core [Ledur et al. 2017]. Assim, o trabalho apresenta as seguintes contribuições:

- Um estudo e implementação do processamento paralelo e distribuído para DSL GMaVis.
- Um conjunto de experimentos com diferentes implementações distribuídas do pré-processamento e tamanhos de arquivos baseados em cargas realísticas.

O artigo está organizado da seguinte forma. A Seção 2 apresenta os trabalhos relacionados e a Seção 3 apresenta a linguagem da DSL GMaVis. A Seção 4 descreve a implementação do suporte ao pré-processamento distribuído. Os experimentos, resultados e discussões são mostrados na Seção 5. As conclusões deste trabalho são apresentadas na Seção 6.

## 2. Trabalhos Relacionados

Nessa seção são apresentadas diferentes abordagens de processamento paralelo e distribuído e DSLs para a visualização e pré-processamento de grandes quantidades de dados para aplicações de geovisualização, incluindo também abordagens para o aprimoramento de desempenho com base nas características de Entrada ou Saída (E/S).

No trabalho de [Seo et al. 2015], foram abordadas operações de E/S coletivas sem bloqueio em MPI. A implementação em MPICH foi baseada no algoritmo coletivo de

E/S ROMIO, substituindo as operações de bloqueio coletivo de E/S ou contrapartes não-bloqueantes. Os resultados indicaram um melhor desempenho que o bloqueio de E/S coletivo em termos de largura de banda de E/S, sendo capaz de sobrepor E/S com outras operações. [Latham et al. 2017] propõem uma extensão para o MPI-3, permitindo determinar quais nós do sistema compartilham recursos comuns. A implementação realizada em MPICH fornece um mecanismo portátil para a descoberta de recursos, possibilitando determinar quais nós compartilham dispositivos locais mais rápidos. Os resultados obtidos com testes de *benchmarks* demonstraram a eficiência da abordagem para investigar a topologia de um sistema. [Mendez et al. 2017] apresentam uma metodologia para avaliar o desempenho de aplicações paralelas com base nas características de E/S da aplicação, requisitos e níveis de severidade. A implementação definiu o uso de cinco níveis de severidade considerando requisitos de E/S de aplicações paralelas e parâmetros do sistema de HPC. Resultados mostraram que a metodologia permite identificar se uma aplicação paralela é limitada pelo subsistema de E/S e identificar possíveis causas do problema.

[Ayachit 2015] descreve o projeto e as características do *ParaView*, que é uma ferramenta de código aberto multiplataforma que permite a visualização e análise de dados. No *ParaView* a manipulação de dados pode ser feita interativamente em 3D ou através de processamento em lote. A ferramenta foi desenvolvida para analisar grandes conjuntos de dados usando recursos de computação de memória distribuída. [Wylie and Baumes 2009] por sua vez, apresentam um projeto de expansão para a ferramenta de código aberto *Visualization Toolkit* (VTK). O projeto foi denominado *Titan*, e oferece suporte a inserção, processamento e visualização de dados. Além disso, a distribuição de dados, o processamento paralelo e a característica cliente/servidor da ferramenta VTK fornecem uma plataforma escalável.

[Steed et al. 2013] descrevem um sistema de análise visual, chamado Ambiente de Análise de Dados Exploratório (EDEN), com aplicação específica para análise de grandes conjuntos de dados (*Big Data*) inerentes à ciência do clima. EDEN foi desenvolvido como ferramenta de análise visual interativa permitindo transformar dados em *insights*, melhorando assim a compreensão crítica dos processos do sistema terrestre. Resultados foram obtidos com base em estudos do mundo real usando conjuntos de pontos e simulações globais do modelo terrestre (CLM4). [Perrot et al. 2015] apresentam uma arquitetura para aplicações de *Big Data* que permite a visualização interativa de mapas de calor em larga escala. A implementação feita em *Hadoop*, *HBase*, *Spark* e *WebGL* inclui um algoritmo distribuído para calcular um agrupamento de *canopy*. Os resultados comprovam a eficiência da abordagem em termos de escalabilidade horizontal e qualidade da visualização produzida.

O estudo apresentado por [Zhang et al. 2015] exploram o uso de tecnologias de análise geovisuais e computação paralela para problemas de otimização geoespacial. O desenvolvimento resultou em um conjunto de ferramentas geovisuais interativas capazes de direcionar dinamicamente a busca por otimização de forma interativa. Os experimentos revelam que a análise visual eficiente e a busca através do uso de árvores paralelas são ferramentas promissoras para a modelagem da alocação do uso da terra.

Nesta seção foram apresentadas diferentes abordagens relacionadas com o pré-processamento e aprimoramento de desempenho de grandes quantidades de dados. Algumas abordagens [Seo et al. 2015], [Latham et al. 2017] são focadas no aprimoramento de

desempenho de aplicações de E/S, enquanto outras [Mendez et al. 2017] permitem identificar se a aplicação é limitada pelo subsistema de E/S. Algumas abordagens [Ayachit 2015], [Wylie and Baumes 2009] estavam preocupadas com a visualização e análise dos conjuntos de dados e outras em permitir a visualização interativa de aplicações *Big Data* [Steed et al. 2013], [Perrot et al. 2015]. Por fim, [Zhang et al. 2015] demonstram o uso de tecnologias de análise geovisuais através de árvores paralelas.

É possível observar que a literatura não apresenta estudos que otimizam o pré-processamento de grandes quantidades de dados levando em consideração uma DSL para um ambiente de processamento paralelo e distribuído. Além dessa lacuna observada, nesse estudo a GMaVis é estendida para suportar pré-processamento distribuído de dados. Ainda, diferentes implementações e tamanhos de arquivos são testados nesse estudo.

## 3. GMaVis DSL

GMaVis [Ledur et al. 2017] é uma DSL que fornece uma linguagem de especificação de alto nível e tem como objetivo simplificar a criação de visualizações para dados geoespaciais em larga escala. A DSL permite aos usuários filtrar, classificar, formatar e especificar a visualização dos dados. Além disso, a GMaVis tem expressividade específica para reduzir a complexidade e automatizar decisões e operações, tais como o pré-processamento de dados, o zoom e a localização do ponto inicial [Ledur et al. 2015].

```
1 visualization: clusteredmap;
2 settings {
3   latitude: field 7;
4   longitude: field 8;
5   marker-text: field 1 field 2;
6   page-title: "Airports in World";
7   size: full;
8 }
9 data {
10  file: "vis_codes/airports.data";
11  structure {
12    delimiter: ',';
13    end-register: newline;
14  }
15 }
```

**Listagem 1. Código da DSL GMaVis para visualizar dados em *clusters*.**

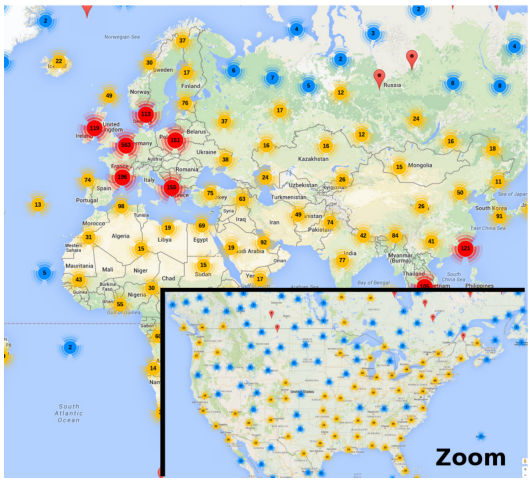



**Figura 1. Visualização dos dados dos aeroportos pelo mundo.**

A Listagem 1 ilustra um exemplo da linguagem de alto nível da GMaVis, e a visualização resultante é apresentada na Figura 1. A primeira declaração na Linha 1 é um *visualization*: declaração, na qual o tipo de visualização escolhido para aparecer na visualização é o *clusteredmap*. Na Linha 2, o bloco *settings* começa com as declarações usadas para especificar detalhes dos aspectos visuais e recebe os campos onde os atributos importantes estão localizados. Por exemplo, a *latitude* e a *longitude* estão sendo declaradas nas Linhas 3 e 4, quando são informados seus valores, que correspondem à posição no conjunto de dados onde essas informações podem ser encontradas. A Linha 5 possui um *marker-text* para ser exibido como um texto no marcador quando o usuário clica nele. A declaração *page-title* na Linha 6 informa o título que será colocado na visualização. Na

Linha 7, uma declaração *size* é usada para definir o tamanho que a visualização ocupará na página Web. A declaração permite também valores como *small* ou *medium*.

Há também um bloco *data* com declarações entre as Linhas 9 a 15 que especificam os arquivos e filtros de entrada. Os usuários também podem incluir sub-blocos para a estrutura e classificação dos dados. O arquivo de entrada é declarado na Linha 10, recebendo uma *string* com o caminho do sistema para o arquivo. Esta declaração pode ser repetida várias vezes até que seja incluído todo o conjunto de dados. Além disso, um sub-bloco *structure* é declarado na linha 11, com um *delimiter* e uma declaração *end-register* especificando que a vírgula separa os valores do conjunto de dados de entrada e o caractere de nova linha separa os registros. Esta declaração pode receber qualquer caractere ou palavras-chave definidas, tais como: *tab*, *comma*, *semicolon* ou *newline*.

### 3.1. Pré-processamento de Dados

O modulo de pré-processamento de dados é responsável pela transformação dos dados de entrada, aplicando operações de filtragem e classificação. Este módulo permite à DSL abstrair a primeira fase do *pipeline* para a criação da visualização [Moreland 2013], evitando que os usuários tenham que tratar manualmente com grandes conjuntos de dados. O módulo funciona recebendo os dados de entrada, processando e salvando em um arquivo de saída os dados formatados e estruturados. As principais operações são *Read*, *Process* e *Write*, sendo definidas juntamente com outras operações na Tabela 1.

**Tabela 1. Operações de pré-processamento de dados e suas definições.**

| Definição | Descrição |
|---|---|
| $F = \{\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n\}$ | $F$ é um conjunto de arquivos de entrada a serem processados e $\alpha$ representa um único arquivo de um conjunto de dados particionado. |
| $Split(\alpha)$ | Divide um arquivo do conjunto de dados de $F$ em $N$ blocos. |
| $D = \{d_1, d_2, d_3, ..., d_n\}$ | $D$ é um conjunto de blocos de um único arquivo. Pode-se dizer que $D$ é o resultado de uma função $Split(\alpha)$. |
| $Process(D)$ | Processa um único arquivo $D$ de $F$. |
| $Read(d)$ | Abre e lê um bloco de dados $d$ de um $\alpha$ em $F$. |
| $Filter(d)$ | Filtra um determinado bloco de dados $d$ em $D$, produzindo um conjunto de registros para criar a visualização. |
| $Classify(...)$ | Classifica os resultados de $Filter(...)$. |
| $Write(...)$ | Salva os resultado de $\sum_{i=1}^{n} Process(F)$, onde $F$ representa um conjunto de arquivos ($\alpha$) em um arquivo de saída a ser usado na geração da visualização. |

O compilador da DSL usa detalhes do código-fonte para gerar esse pré-processamento de dados através da linguagem de programação C++ [Ledur et al. 2017]. A escolha do C++, é justificada pelo fato de permitir a criação de uma aplicação usando um vasto conjunto de interfaces de programação paralela, além de possibilitar aprimoramentos de baixo nível no gerenciamento de memória e leitura de disco. Assim, o compilador gera um arquivo chamado *data_preprocessor.cpp*. Sendo todo código gerado e executado sequencialmente, incluindo bibliotecas, constantes e tipos de dados. A execução da aplicação de geovisualização usando a GMaVis pode ser paralela usando o suporte ao paralelismo em ambientes multi-core oferecido pela DSL SPar [Griebler et al. 2017]. Além

disso, as informações relevantes do código-fonte da DSL são transformadas e escritas nesse arquivo.

A Listagem 2 ilustra uma representação de codificação em alto nível da função de processamento da GMaVis, onde é realizada a leitura de cada bloco do disco, filtragem, classificação e, finalmente a gravação dos dados no arquivo de saída. Para que isso ocorra, as operações iniciais são executadas entre as Linhas 5 e 6 e definem o inicio da região de leitura de dados. Enquanto que nas Linhas 7 e 8 os diferentes blocos de arquivos são processados, gerando uma *string* de dados com base na operações de filtragem e classificação. A última etapa consome a *string*, com base nas operações executadas nas Linha 9 realizando a gravação no arquivo de saída.

```
1 function process(args...){
2   Open(in_file);
3   Open(out_file);
4   while(!in_file.eof()){
5     d_size = Split(in_file);
6     d = Read(in_file,d_size);
7     f = Filter(d);
8     c = Classify(f);
9     Write(c,out_file);
10  }
11  Close(in_file);
12  Close(out_file);
13 }
```

**Listagem 2. Representação da função de processamento em alto nível.**

O compilador do GCC é chamado para criar o executável de pré-processamento de dados. Após a compilação, o compilador da DSL chama o módulo de pré-processamento de dados e espera que sejam realizadas as transformações de dados, apresentando os dados pré-processados. Esta saída é carregada no gerador de visualização que usa as informações armazenadas a partir do código fonte. O analisador da DSL fornece informações sobre os detalhes especificados no bloco de configuração, como tamanho, título, texto do marcador e visualizações a serem criadas.

## 4. Modelagem Paralela do Pré-Processamento na DSL GMaVis

Conforme descrito por [Ledur et al. 2017], a DSL GMaVis permite suporte à visualização geoespacial somente em arquiteturas *multi-core* através da geração de anotações de código [Griebler et al. 2017]. Desse modo, para habilitar o suporte ao processamento paralelo e distribuído para o módulo de pré-processamento de dados, foram implementadas três variantes do modelo Mestre/Escravo. Uma versão em MPI estática, uma versão com distribuição dinâmica de tarefas e outra usando o MPI-I/O. Essas versões foram denominadas respectivamente: MPI-E, MPI-D e MPI-I/O.

O MPI-E implementa a versão estática do modelo Mestre/Escravo em MPI. Nas Listagens 3 e 4 são apresentados os trechos de código das implementações Mestre e Escravo da versão MPI-E, respectivamente. Note que o processo principal (Mestre) realiza uma contagem inicial da quantidade de conjuntos de dados a serem processados. Após essa contagem, é realizada a divisão proporcional dos conjuntos de dados disponíveis entre os processos (Escravos). Cabe destacar que o processo Mestre também exerce a

função de Escravo, além da sua tarefa usual de coordenação e envio de trabalho aos processos Escravos. Outro aspecto importante diz respeito aos conjuntos de dados, que são distribuídos entre os processos de forma aleatória, não havendo nenhuma análise previa.

```
MPI_E_MASTER(args...){
    files<path>;
    numFiles = files.size()/np;
  for(k=1;k<np;k++){
      MPI::Send(numFiles);
      for(i=0;i<numFiles;i++){
          MPI::Send(files.pop());
      }
  }
  while(!files.empty()){
      file = files.pop();
      process(file);
    }
}
```

**Listagem 3. MPI-E (Processo Mestre).**

```
MPI_E_SLAVE(args...){
    files<path>;
    MPI::Recv(numFiles);
  for(i=0;i<numFiles;i++){
      MPI::Recv(path);
      files.add(path)
  }
  while(!files.empty()){
      file = files.pop();
      process(file);
    }
}
```

**Listagem 4. MPI-E ( Processo Escravo).**

```
MPI_D_MASTER(args...){
    files<path>;
    numFiles = files.size()/np;
  for(k=1;k<np;k++){
      MPI::Send(files.pop());
  }
  for(i=0;i<numFiles;i++){
      MPI::Recv(free_slave);
      if(!files.empty()){
          MPI::Send(files.pop());
      }else{
          MPI::Send(end_message);
      }
    }
}
```

**Listagem 5. MPI-D (Processo Mestre).**

```
MPI_D_SLAVE(args...){
    MPI::Recv(file,tag);
    while(tag != end_message){
        function_process(file);
        MPI::Send(free_slave);
        MPI::Recv(file,tag);
    }
}
```

**Listagem 6. MPI-D (Processo Escravo).**

A versão dinâmica do módulo de pré-processamento de dados implementa o modelo Mestre/Escravo denominado MPI-D. A versão dinâmica se refere a habilidade do processo Escravo solicitar tarefas dinamicamente. Isso não deve ser confundido com a capacidade de criação de novos processos (dinamicamente) em tempo de execução, característica obtida através do uso da função *MPI_Comm_spawn* que não foi utilizada, presente desde a versão 2 do MPI. Assim, o processo principal (Mestre) do MPI-D realiza a distribuição dos conjuntos de dados entre os processos Escravos, conforme demonstrado na Listagem 5. Um vez realizada a fase de distribuição, inicia-se a fase de espera por parte do Mestre, que fica no aguardo do término da tarefa por parte de algum Escravo (demonstrado na Listagem 6), para somente então enviar uma nova tarefa para o Escravo. O aspecto dinâmico, está centrado no comportamento do Escravo, ou seja, a iniciativa

de solicitar uma nova tarefa parte dinamicamente do Escravo. Obviamente, isso ocorre somente se ele já finalizou sua tarefa anteriormente recebida.

Finalmente, a versão utilizando MPI-I/O é descrita. É importante ressaltar que a versão de MPI-I/O difere em alguns aspectos das demais versões (MPI-E e MPI-D), pelo fato de ter como principal forma de comunicação a troca de endereços entre os processos, pois há apenas um *único arquivo* compartilhado, com cada Escravo lendo e escrevendo em uma parte do arquivo. Assim, o processo principal (Listagem 7) calcula o tamanho total do conjunto de dados, realiza a divisão em partes (através da função *calculateJob()*) e envia o primeiro endereço do arquivo a ser processado para o primeiro escravo, representado na Listagem 8, e assim divide o arquivo com os demais processos, podendo ter novas iterações caso o número de *chunks* seja superior ao número de processos.

```
MPI_IO_MASTER(args ...){
  MPI::Send(primeiroPonteiro);
  while(hasJob){
    MPI::Recv(tag);
    if(tag==ponteiro){
        calculateJob();
        if(hasJob){
            getFreeSlave();
            MPI::Send(ponteiro);
        }else{
            MPI::Send(endSignal);
         }
     }else if(tag==endRead){
        if(hasJob){
            MPI::Send(ponteiro);
        }else{
            MPI::Send(endSignal);
         }
    }
  }
}
```

**Listagem 7. MPI-IO (Mestre).**

```
MPI_IO_SLAVE(args ...){
  MPI::File in;
  do{
      MPI::Recv(tag);
      if(tag==ponteiro){
        inicio=ponteiro;
        fim = ponteiro + chunk;
        buffer =
              in.Read(inicio,fim);
        ptrRetorno =
            ajustaRegistro(buffer);
        MPI::Send(ptrRetorno);
        process(buffer);
      }else if(tag==endSignal){
        MPI::Finalize();
        }
    } while (tag != endJob);
}
```

**Listagem 8. MPI-IO (Escravo).**

Nesse caso, cada processo escravo realiza a leitura, processamento e escrita na parte designada a cada um. O Mestre continua sendo responsável pela distribuição de tarefas. No entanto, durante interações envia endereços a serem processados para eventuais processos escravos disponíveis (através da função *getFreeSlave()*). Feito isso, realizam o processamento, classificação e filtragem dos dados. Em outras palavras, a implementação busca dividir o arquivo em *chunks* para cada escravo processar uma parte de forma independente e melhorar o desempenho a partir da leitura e escrita em partes. Os diversos arquivos de entrada são agrupados em um único arquivo compartilhado. Cada escravo lê e escreve no arquivo compartilhado através de troca de mensagens.

## 5. Experimentos

Para a realização dos experimentos, foram utilizados essencialmente três conjuntos de dados obtidos através do *Yahoo Flickr Creative Commons* [Thomee et al. 2016]: um conjunto de dados denominado de grande (4GB), um médio (1GB) e um pequeno (200MB).

Além disso, os experimentos executados para verificar o desempenho utilizaram um número diferente de réplicas (grau de paralelismo) dependendo do tipo de carga (balanceada ou desbalanceada), sendo realizadas 15 repetições para cada experimento, as diferentes em nenhum cenário apresentaram um desvio padrão superior a 2%. O tamanho de cada *chunk* usado em todos os experimentos foi definido como 100 MB, sendo um tamanho de *chunk* representativo para os cenários testados.

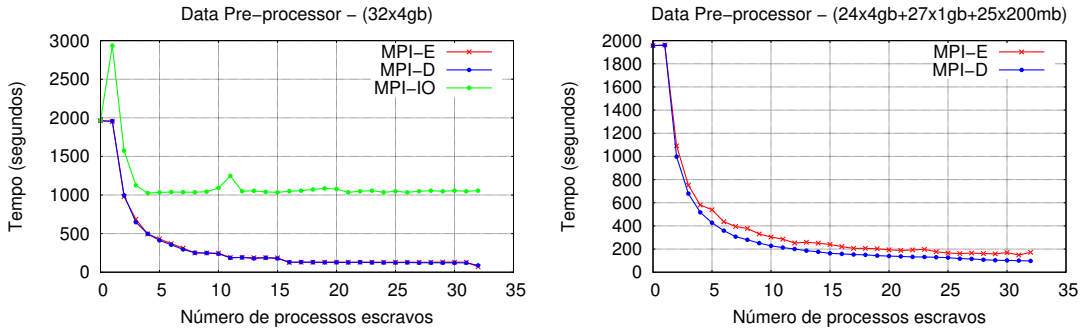
Os testes com carga balanceada utilizaram 32 réplicas de cada arquivo com carga grande (32x4GB), carga média (32x1GB) e carga pequena (32x200MB) para MPI-E, MPI-D e MPI-I/O. Por sua vez, os testes com carga desbalanceada combinaram o uso de diferentes cargas para MPI-E, MPI-D e MPI-I/O. Onde a carga grande (24x4GB + 27x1GB + 25x200MB), a carga média (6x4GB + 6x1GB + 10x200MB) e a carga pequena (1x4GB + 2x1GB + 2x200MB) foram definidas intencionalmente de modo a simular desbalanceamento dos conjuntos de dados. O número de arquivos bem como seus tamanhos foram definidos para que o resultado da soma dos arquivos totalizasse a mesma quantidade de dados (em Gigabytes) da carga balanceada.

Em relação ao ambiente de execução, a infraestrutura consistiu de quatro nodos, sendo que cada nodo possui uma configuração com dois processadores Intel Xeon E5520 - 2.27GHz (cada um com 4 cores e 8 *Hyper-Threads*) e 16GB de memória RAM. A rede de interconexão utilizada foi uma Gigabit Ethernet. Além disso, utilizou-se o sistema operacional Ubuntu 14.04.4 LTS (GNU/Linux 3.13.0-86-genérico x86 64) e o módulo de pré-processamento de dados foi compilado a partir do GCC-5.3.0 sem nenhuma *flag* de otimização definida.

É importante ressaltar que não foram realizados testes com carga desbalanceada para as versões de MPI-I/O, devido a implementação agrupar diferentes arquivos de entrada em um único arquivo e distribuir o processamento enviando *chunks* para os processos escravos, cada processo lê e escreve em uma porção do arquivo. Sendo assim, a versão de MPI I/O é efetiva também no cenário com diferentes tamanhos de arquivos. No entanto, o agrupamento e divisão desses arquivos em um único evita o desbalanceamento de carga, o desempenho seria o mesmo com arquivos de tamanhos idênticos.

A Figura 2 ilustra as versões paralelas do módulo de pré-processamento de dados executando com carga de trabalho grande (balanceada e desbalanceada). A execução com 0 processos paralelos representa a execução sequencial. Na Figura 2(a) é possível observar o resultado do experimento com o uso da carga de trabalho balanceada. Note que MPI-E e MPI-D apresentaram uma curva de desempenho semelhante. A exceção ficou por conta de MPI-I/O que apresentou um desempenho inferior. No outro cenário apresentado na Figura 2(b), com carga de trabalho desbalanceada, MPI-D se mostrou mais eficiente que MPI-E. Em relação a MPI-E, a diferença (pequena) de desempenho se torna mais evidente a partir do uso de 5 processos escravos.

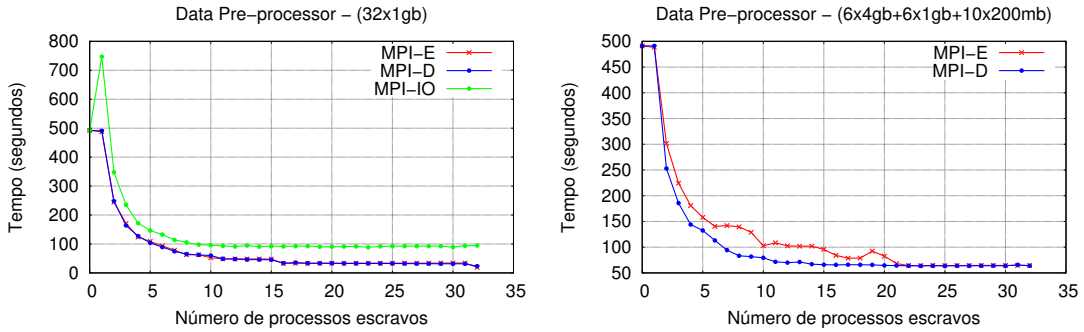
As versões paralelas do módulo de pré-processamento de dados executando com carga de trabalho média (balanceada e desbalanceada) são apresentadas na Figura 3. O experimento com carga balanceada (Figura 3(a)) apresentou novamente uma curva de desempenho praticamente igual para MPI-E e MPI-D. MPI-I/O se mostrou mais uma vez menos eficiente. No entanto, é possível perceber uma aproximação de MPI-I/O com base em sua curva de desempenho. O experimento desbalanceado (3(b)) por sua vez, sinaliza




(a) Carga grande balanceada. (b) Carga grande desbalanceada.

**Figura 2. Módulo de pré-processamento de dados paralelo com carga grande.**

uma tendência, onde as curvas de desempenho de MPI-E, MPI-D e MPI-I/O praticamente convergiram. MPI-D continuou como a versão mais eficiente, conseguindo superar em eficiência MPI-E entre 3 e 21 processos escravos. O tempo total para execução de MPI-I/O continuou sendo maior.



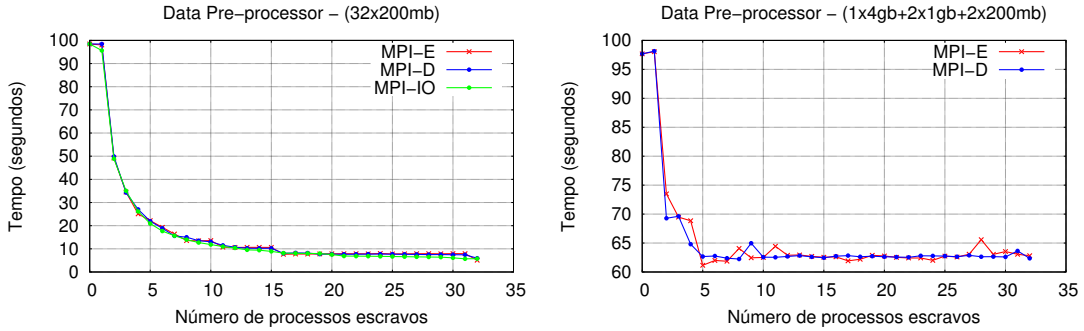

(a) Carga média balanceada. (b) Carga média desbalanceada.

**Figura 3. Módulo de pré-processamento de dados paralelo com carga média.**

Para o último cenário, mostrado na Figura 4, foram executadas versões paralelas do módulo de pré-processamento de dados com carga de trabalho pequena (balanceada e desbalanceada). A Figura 4(a) apresenta o experimento realizado com cargas balanceadas. Nesse cenário, pode-se dizer que as três versões (MPI-E, MPI-D e MPI-I/O) tiveram resultados semelhantes de desempenho. Porém, o experimento realizado com carga pequena balanceada mostra o cenário MPI-I/O com desempenho levemente superior aos outros cenários. A sua curva de desempenho foi mais eficiente em relação a utilização de recursos de MPI-E e MPI-D. Esse comportamento se torna mais evidente a partir do uso de 5 processos escravos, se mantendo assim até o final da execução.

A partir dos resultados obtidos, é possível identificar tendências e perfazer algumas considerações. A versão de MPI-D claramente se mostrou mais eficiente com o uso de cargas grandes e médias (sejam balanceadas ou desbalanceadas). Esse resultado de certa forma já era esperado, uma vez que o comportamento dinâmico na distribuição das tarefas de MPI-D possibilita uma melhor utilização dos recursos. No entanto, com o uso de cargas pequenas balanceadas o MPI-I/O apresentou crescimento de desempenho da versão de MPI-I/O. A curva de desempenho mostrou uma utilização de recursos bastante

acentuada. Isso ocorre devido ao MPI-I/O operar sobre um único arquivo compartilhado com todos os processos, que com tamanho pequeno tiveram um melhor balanceamento. Por outro lado, o MPI-I/O com um arquivo maior, tem desempenho inferior, limitado pelo *chunk* do arquivo mais lento processado nos escravos e também pelo gargalo de desempenho nas operações de *read* e *write* de disco no arquivo compartilhado.




(a) Carga pequena balanceada. (b) Carga pequena desbalanceada.

**Figura 4. Módulo de pré-processamento de dados paralelo com carga pequena.**

## 6. Conclusões

Neste artigo foi apresentada uma proposta de suporte ao processamento paralelo e distribuído, com foco em uma DSL para visualização de grandes conjuntos de dados geoespaciais. Para isso, foram avaliadas diferentes implementações do padrão de programação distribuído MPI. O pré-processamento de dados é a etapa mais demorada para visualização de dados. Assim, o uso do processamento distribuído torna possível evitar gargalos de desempenho de disco em um único nodo e potencialmente aumentar o desempenho com diversas operações simultâneas em diferentes nodos.

Os resultados obtidos demonstram um desempenho superior da versão MPI-D com o uso de cargas de tamanho grande e médio. A abordagem de distribuição dinâmica de tarefas permitiu uma melhor utilização dos recursos, com cargas balanceadas e desbalanceadas. Por outro lado, a versão de MPI-I/O obteve o melhor resultado com cargas pequenas, mostrando que a carga de trabalho não foi limitada pelo subsistema de E/S, não havendo gargalos de desempenho devido ao seu tamanho reduzido.

Futuramente, um objetivo é implementar novas técnicas em implementações do MPI-I/O. Ainda, pretende-se implementar uma versão de MPI adaptativo, ou seja, a versão deverá permitir a criação de processos em tempo de execução, permitindo assim explorar a elasticidade horizontal do ambiente. Para isso, pretende-se fazer uso do estilo de programação dinâmica do MPI-2.

## Agradecimentos

Os autores gostariam de agradecer o suporte financeiro parcial das pesquisas para as seguintes instituições: PUCRS, CAPES e CNPq.

## Referências

[Ayachit 2015] Ayachit, U. (2015). *The ParaView Guide: A Parallel Visualization Application*. Kitware, Inc., USA.

[Griebler et al. 2017] Griebler, D., Danelutto, M., Torquati, M., and Fernandes, L. G. (2017). SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):20.

[IDC 2012] IDC (2012). The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East.

[Kraak and Ormeling 2011] Kraak, M. and Ormeling, F. (2011). *Cartography, Third Edition: Visualization of Spatial Data*. Guilford Publications.

[Latham et al. 2017] Latham, R., Bautista-Gomez, L., and Balaji, P. (2017). Portable Topology-Aware MPI-I/O. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 710–719.

[Ledur et al. 2017] Ledur, C., Griebler, D., Manssour, I., and Fernandes, L. G. (2017). A High-Level DSL for Geospatial Visualizations with Multi-core Parallelism Support. In *41th IEEE Computer Society Signature Conference on Computers, Software and Applications*, COMPSAC'17, Torino, Italy. IEEE.

[Ledur et al. 2015] Ledur, C., Griebler, D., Manssuor, I., and Fernandes, L. G. (2015). Towards a Domain-Specific Language for Geospatial Data Visualization Maps with Big Data Sets. In *ACS/IEEE International Conference on Computer Systems and Applications*, AICCSA'15, page 8, Marrakech, Marrocos. IEEE.

[Mendez et al. 2017] Mendez, S., Rexachs, D., and Luque, E. (2017). Analyzing the Parallel I/O Severity of MPI Applications. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 953–962.

[Moreland 2013] Moreland, K. (2013). A Survey of Visualization Pipelines. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):367–378.

[Perrot et al. 2015] Perrot, A., Bourqui, R., Hanusse, N., Lalanne, F., and Auber, D. (2015). Large Interactive Visualization of Density Functions on Big Data Infrastructure. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 99–106.

[Rünger and Rauber 2013] Rünger, G. and Rauber, T. (2013). *Parallel Programming - for Multicore and Cluster Systems; 2nd Edition.* Springer.

[Seo et al. 2015] Seo, S., Latham, R., Zhang, J., and Balaji, P. (2015). Implementation and Evaluation of MPI Nonblocking Collective I/O. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1084–1091.

[Steed et al. 2013] Steed, C. A., Ricciuto, D. M., Shipman, G., Smith, B., Thornton, P. E., Wang, D., Shi, X., and Williams, D. N. (2013). Big Data Visual Analytics for Exploratory Earth System Simulation Analysis. *Comput. Geosci.*, 61:71–82.

[Thomee et al. 2016] Thomee, B., Shamma, D. A., Friedland, G., Elizalde, B., Ni, K., Poland, D., Borth, D., and Li, L.-J. (2016). YFCC100M: The New Data in Multimedia Research. *Commun. ACM*, 59:64–73.

[Wylie and Baumes 2009] Wylie, B. N. and Baumes, J. (2009). A Unified Toolkit for Information and Scientific Visualization. In *VDA*, page 72430.

[Zhang et al. 2015] Zhang, T., Hua, G., and Ligmann-Zielinska, A. (2015). Visually-driven Parallel Solving of Multi-objective Land-use Allocation Problems: A Case Study in Chelan, Washington. *Earth Science Informatics*, 8:809–825.

# A Fast Similarity Search kNN for Textual Datasets

**Leonardo Afonso Amorim, Mateus Ferreira e Freitas,
Paulo Henrique da Silva, Wellington Santos Martins**

[1] Instituto de Informática – Universidade Federal de Goiás (UFG)
Caixa Postal 131 – 74.001-970 – Goiânia – GO – Brazil

{leonardoafonso,mateusfreitas,paulohsilva,wellington}@inf.ufg.br

***Abstract.*** *The $k$ nearest neighbors ($k$NN) is an algorithm for finding the closest $k$ points in metric spaces. Due to its high computational costs, many parallel solutions have been proposed, including some implementations targeted at modern accelerators. However, most approaches assume relatively low dimensionality and dense data. Such conditions do not apply to textual datasets, which are known for their high dimensionality and sparsity. This work presents a fine-grained parallel algorithm that applies filtering technique based on most common important terms of the query document using an inverted index and its implementation on GPU. Our method improves the top $k$ nearest neighbors search in textual datasets by up to 37x with a single GPU.*

## 1. Introduction

Searching is one of the most fundamental problems in computer science, present in practically all applications. Most of the early searching algorithms were designed with the traditional notion of exact search, i.e, it consisted in finding an element whose identifier matched exactly to a given search key. Nowadays, a great number of applications have to work with databases containing mostly unstructured data. In this scenario, it is not always possible to define meaningful search keys for each database element. In addition, the rise of high dimensional big data from different data sources (e.g. images, sounds, text), and the lack of a natural ordering among dimensions made it hopeless to hierarchically search using classical exact search techniques. In modern information retrieval systems, the queries usually ask for relevant objects, that are similar, to a given one, whereas comparison for exact equality is very rare. Hence, the focus of searching has been changed to similarity or proximity search.

A well-known type of similarity search is the Nearest Neighbor (NN) search, which searches for data points that are close to a given query point based on a distance measure. The closeness depends on the distance measure defined that in turn depends on the topologies and characteristics of the data. One particular type of nearest neighbor search that has a wide applicability is finding the $k$ nearest neighbors, known as $k$NN. It is extensively used in information retrieval systems, especially in search engines either for text, image or audio retrieval for retrieving the data as well as ranking them. Apart from applications focused purely on search, one common application domain for $k$NN searching is instance-based learning. This so-called $k$NN classifier estimates or classifies a point based on the consensus of its neighbors. However, finding the $k$ nearest neighbors of a query point can easily become prohibitive for large datasets and high dimensional space. Thus, a variety of parallel solutions have been proposed, including some implementations targeted at modern accelerators (GPUs).

Most of the existing GPU-based $k$NN parallel implementations assume relatively low dimensionality and dense data. Such conditions do not apply to textual datasets, which are known for their high dimensionality and sparsity. In this work, we present a fine-grained parallel $k$NN algorithm and GPU-based implementation that exploits data parallelism and greatly improves the top $k$ nearest neighbors search in textual datasets. The solution efficiently implements a threshold-based filtering technique and an inverted index on the GPU. Different from many $k$NNs, our approach can achieve high performance even when dealing with just one query, which enables the processing of real-time $k$NN searches. The solution is also made scalable by exploiting task parallelism so that multi-GPU systems can be used to process a large number of queries. Experiments performed with MEDLINE dataset showed significant performance gains when compared with other $k$NN implementations described in the literature. The main contributions of the paper are: (i) A threshold-based filtering technique that improves the sorting time of $k$NN candidates; (ii) A scalable multi-GPU implementation that exploits both data parallelism and task parallelism; (iii) Extensive experimental work with a standard real-world textual dataset.

This paper is organized as follows. In Section 2 we present the $k$NN problem, its uses in document retrieval and some related work which aims at improving the $k$NN on GPU platforms. In section 3 we present the basis of our GPU-based parallel $k$NN search for textual datasets, as well as its extension to a multi-GPU environment and filtering proposals. The experiments are described and discussed in section 4. Finally, in Section 5 we present our conclusions and highlight future work.

## 2. Background

The $k$ nearest neighbors ($k$NN) is an algorithm for finding the closest $k$ points in metric spaces. Formally, the $k$NN search problem is defined as follows: given a set $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ of $n$ points in $\mathbb{R}^d$ space and a query point $q$ also in $\mathbb{R}^d$, find the $k$ closest points in $\mathcal{X}$ to $q$. The closeness depends on the distance measure used, which may be the Euclidean distance, the Manhattan distance or any other distance that is appropriate to the context.

One way to solve the $k$NN problem is to use a brute force approach to find the nearest neighbours of a point. It consists of computing all the distances between $q$ and the points in $\mathcal{X}$, sorting the computed distances and selecting the $k$ points corresponding to the $k$ smallest distances. However, the actual calculation of the distances and the selection of the nearest neighbours for large datasets require high computation cost, $\mathcal{O}(nd)$ for calculating the $n$ distances and $\mathcal{O}(n \log n)$ for sorting them.

Various $k$NN algorithms have been proposed to reduce this computation time [Friedman et al. 1977, Johnson et al. 2017, Chen et al. 2017, Gutiérrez et al. 2016, Wang et al. 2018]. In general, the idea is to reduce the number of distances calculated. For example, some algorithms hierarchically partition the data points, creating a tree structure (e.g. $kd$-tree), and only calculate distances within nearby space [Friedman et al. 1977]. This approach reduces the time complexity but does it at the cost of increased preprocessing time. Also, for sufficiently high dimension, this approach ends up having to search many extra nodes and offer little improvement over brute-force searching.

For very large dimensions, the most promising approaches sacrifice some precision to gain more efficiency and find only approximate nearest neighbors. These ap-

proaches tend to compute neighbors that are close enough to the query item instead of the exact $k$ nearest neighbors. One of the widely used approximate methods is locality-sensitive hashing (LSH), which uses a family of hash functions to cluster nearby items into bins, with a high probability [Paulevé et al. 2010]. Query items are hashed into one bin whose items are used as potential candidates for the final results. The search time is sublinear in the collection data size. However, the bin sizes are usually large enough to require an exact $k$NN search in the chosen bin. Besides, in order to scale, approximate methods sacrifice effectiveness using an approximated $k$NN solution.

Similarity searching algorithm is at the heart of document retrieval systems. It scores queries against documents and computes the highest scoring documents for presentation to the user in ranked order. The $k$NN algorithm is commonly used for this function, retrieving the most similar $k$ documents for each query document. The most traditional way of finding the nearest neighbors is to represent each document as a vector. In the vector space model [Salton et al. 1975], a document is a vector with one component for each unique term (word) in the vocabulary of the collection. As the vocabulary can be arbitrarily large, the dimensionality of this space is usually very high, with tens of thousands of terms used routinely.

The value for each coordinate of the document is the weight for that term in that document. In this work, we use TF-IDF weight. In information retrieval, TF-IDF, short for term frequency-inverse document frequency, is a numerical statistic that is designated to reflect on how important a word is to a document in a collection or corpus. While computing TF (Term Frequency), all terms are considered equally important, TF-IDF weighs down the frequent terms while it scales up the rare ones.

This is known as bag-of-words representation, since the ordering of words (terms) is ignored. Also, since most coordinates contain terms that do not occur in the document (zero weight), documents are represented by sparse vectors. Queries are also represented as vectors. Similarity search ranks documents with respect to their similarity to the query. The similarity measure most commonly used is the cosine of the angle between the query vector and the document vector. It depends only on the directions of these vectors, and is independent of their magnitudes, avoiding assigning higher scores to longer documents.

## Related Work

Although the $k$NN algorithm can be applied broadly, it has some shortcomings. For large datasets and high dimensional space, its complexity $\mathcal{O}(nd)$ can easily become prohibitive. Moreover, if $m$ succesive queries are to be performed, the complexity further increases to $\mathcal{O}(mnd)$. This has motivated a number of parallel implementations of the $k$NN method over the last years. More recently, with the powerful and affordable GPU (Graphics Processing Units) many-core accelerators, some proposals have been presented to accelerate the $k$NN algorithm via a highly mutithreaded fine-grained data parallel approach.

Johnson et al. proposed FAISS, an approximate $k$NN search algorithm for dense vectors [Johnson et al. 2017]. It uses domain compression techniques, and inverted lists to build lookup tables for fast searching. It also provides an exact $k$NN search for dense datasets. It implements a brute force approach through matrix operations by using cuBLAS, a CUDA linear algebra library, and a novel GPU $k$-selection method that is applied during matrix multiplication. Although it is not suited for very high dimensions

when the data is too large to fit in GPU memory the problem is tiled over query batches, while using pinned memory to overlap the data transfer and computation.

Gutiérrez et al. proposed GPU-SME-kNN (Scalable and Memory Efficient kNN) to accelerate $k$NN for large datasets with a memory efficient tiling scheme. Their method is able to incrementally select $k$ neighbors by tiling the distance matrix and merging the $k$ distances from the previous tile with a novel quicksort-inspired $k$-selection [Gutiérrez et al. 2016]. This work is not suited for sparse datasets.

Chen et al. proposed a GPU-based $k$NN that uses triangular inequality. It first creates clusters for the query and target sets, then applies an upper bound filter to discard distant target clusters. A second filtering is done on the remaining clusters by sorting the clusters and candidate points, in order to increase the effectiveness of filtering by triangle inequality. It also can adapt its algorithm according to input data [Chen et al. 2017].

Wang et al. proposed a similarity search for high-dimensional and sparse datasets, which implements novel LSH techniques that make it not require similarity computations and high memory. This work does not provide an exact solution, but it is the state-of-the-art GPU approximate $k$NN [Wang et al. 2018].

An approach that exploits fine granularity processing in an exact way is the G-KNN. The G-KNN is a parallel algorithm of the $k$NN and was also implemented on GPUs. This algorithm stands out because it exploits high-dimensional and sparse datasets, which is an advantage over other algorithms in the literature that parallelize $k$NN using GPU. This algorithm was developed to run in an environment with a single GPU. An indexing strategy based on a graph data structure was adopted and the algorithm was divided into two phases: the generation of the model and the classification of the documents. The first phase consists of two steps: calculating the distance between each test object and all training objects and ordering these distances. In the second phase, each test document is classified according to the nearest k training objects [Rocha et al. 2015].

Canuto's et al. work presents a new GPU-based implementation of $k$NN, called GPU-based Textual $k$NN (GT-$k$NN), specially designed for high-dimensional and sparse datasets. This algorithm allows a very fast and much more scalable meta-feature generation which allows one to apply this technique in large collections much faster. Their solution efficiently implements an inverted index in the GPU, by using a parallel counting operation followed by a parallel prefix-sum calculation. At query time, this inverted index is used to quickly find the documents sharing terms with the query document. This is made by constructing a query index which is used for a load balancing strategy to evenly distribute the distance calculations among the GPU's threads. This solution does not use any filtering technique [Canuto et al. 2015].

Matsumoto and Yiu proposed an exact solution by applying a fast sampling-based pruning method to compress distance matrices. In this work, samples are chosen at random from the full dataset. On a small sample, initial distance computation with a set of queries can be done quickly, with the $k$-th nearest neighbor extracted as the threshold value. Objects that have a distance from the query greater than the threshold are discarded. To balance parallelism and compression, queries are arranged into smaller batches. Besides that, they use a standard matrix-based approach to compute Euclidean (L2) distance between the data points and the query points [Matsumoto and Yiu 2015].

Alewiwis et al. approach proposes the application of a new filtering technique that decreases the number of comparisons between the query set and the search set to find highly similar documents. In general, this method proposes a filtering technique known as prefix filtering, in which only the most important features are used to find highly similar documents. This work [Alewiwi et al. 2016] does not provide an exact solution and its implementation is a coarse granularity solution. Another effort, not directly related to this work, use GPUs to accelerate the $k$NN search for low dimensional data with a $kd$-tree data structure [Gieseke et al. 2014].

Our proposal differs from the above mentioned work in many aspects. First, it is an exact $k$NN solution but it does not use the brute-force approach. Since we deal with textual datasets, we avoid comparing the query with all documents in the collection, by creating an inverted index and quickly finding the documents sharing terms with the query document. This also save us a lot of space since the inverted index corresponds to a sparse representation of the data. Other advantage of our approach is that we do not have to rely on multiple queries to achieve high performance. A single query is enough to completely occupy the GPU and this is important since some applications may require a fast processing of a continuous streaming of data. However, our proposal also handles well batches of queries, by simply processing one after another on a multi-GPU environment. In addition, our algorithm uses filtering techniques to discard documents that do not have important terms in common in relation to the query document. This strategy avoids unnecessary computing by further enhancing performance. Our goal is to achieve a high similarity value between the samples and the current query. At query time, we implement a threshold-based filtering by selecting samples (documents) that share terms with high TF-IDF values. The threshold is chosen among these samples, then all distances smaller than it can be pruned, while the higher distances are compacted into a smaller array[1]. Our method uses the cosine as distance calculation. Finally, the $k$ nearest neighbors are determined through the use of a radix sort algorithm on this smaller array.

## 3. A Parallel *k*NN Proposal

A fine-grained parallel algorithm takes advantage of data parallelism by processing individual items, terms of an individual document, in parallel. This greatly improves the $k$ nearest neighbors search in textual datasets and can be easily mapped to modern highly threaded accelerators like manycore GPUs. Our implementation, called Fast Similarity Search for Text (FaSST-$k$NN), efficiently implements an inverted index taking advantage of Zipf's law, which states that in a textual corpus, few terms are common, while many of them are rare. This makes the inverted index a good choice for saving space and avoiding unnecessary calculations. At query time, this inverted index is used to quickly find the documents sharing terms with the query document. This is made by constructing a query index which is used for a load balancing strategy to evenly distribute the distance calculations among the GPU's threads. The inverted index and distance calculation were based on GT-$k$NN [Canuto et al. 2015]. Also at query time, it implements a new threshold-based filtering by selecting samples (documents) that share terms with high TF-IDF values. The threshold is chosen among these samples, then all distances smaller than it can be pruned. This is done on the GPU with a parallel compaction algorithm, which copies all distances greater than the threshold into a smaller array. Finally, the $k$

[1]Since the cosine distance is a similarity metric, the nearest distances are the higher ones.

nearest neighbors are determined through the use of a radix sort algorithm on this smaller array. In addition to exploiting intra-query parallelism, the solution also deals with inter-query parallelism, which allows the use of modern multi-GPU systems. Next, we present a detailed description of these steps.

### 3.1. Data Indexing

The data indexing process consists of creating an inverted index in the GPU memory, assuming the input dataset fits in memory and is static. Let $\mathcal{V}$ be the vocabulary of the input dataset, that is the set of distinct terms of the input set of documents $\mathbb{D}_{in}$. The input data is the set $\mathcal{E}$ of distinct term-documents $(t, d)$, pairs occurring in the original dataset, with $t \in \mathcal{V}$ and $d \in \mathbb{D}_{in}$. Each pair $(t, d) \in \mathcal{E}$ is initially associated with a term frequency $tf$, which is the number of times the term $t$ occurs in the document $d$. An array of size $|\mathcal{E}|$ is used to store the inverted index. Once the set $\mathcal{E}$ has been moved to the GPU memory, each pair in it is examined in parallel, so that each time a term is visited the number of documents where it appears (document frequency - $df$) is incremented and stored in the array $df$ of size $|\mathcal{V}|$. A parallel prefix-sum is executed on the $df$ array by mapping each element to the sum of all terms before it and storing the results in the $index$ array. Thus, each element of the $index$ array points to the position of the corresponding first element in the $invertedIndex$, where all $(t, d)$ pairs will be stored ordered by the term. Finally, the pairs $(t, d)$ are processed in parallel and the *term frequency-inverse document frequency* $tf - idf(t, d)$ for each pair is computed and included together with the documents identification in the $invertedIndex$ array, using the pointers provided by the $index$ array. Also during this parallel processing, the value of the norm for each input document is computed and stored in the $norms$ array. These values are used later when calculating the cosine distances. [Canuto et al. 2015].

### 3.2. *k* Nearest Neighbors Search

Given a query, the $k$NN search consists of two steps. First, the distances of the query (document) $q$ to all input data (documents in $\mathbb{D}_{in}$) have to be computed. Then, the top $k$ documents, that is, those closer to the query, are selected. The distances computation can take advantage of the inverted index model, because only the distances between query $q$ and those documents in $\mathbb{D}_{in}$ that have terms in common with $q$ have to be computed. These documents correspond to the elements of the $invertedIndex$ pointed to by the entries of the $index$ array corresponding to the terms occurring in the query $q$.

The obvious solution to compute the distances is to distribute the terms of query $q$ evenly among the processors and let each processor $p$ access the inverted lists corresponding to terms allocated to it. However, the distribution of terms in documents of text collections is known to follow approximately the Zipf's Law. This means that few terms occur in large amount of documents and most of terms occur in only few documents. Consequently, the sizes of the inverted list also vary according to the Zipf's Law, thus distributing the work load according to the terms of $q$ could cause a great imbalance of the work among the processors.

The load balancing technique described in [Canuto et al. 2015] is used to boost the computation of the distances, by distributing the documents evenly among the processors so that each processor computes approximately the same number of distances. After the distances are computed, the $k$ closest documents to the query are selected. This is

accomplished by making use of a sorting algorithm on the array containing the distances, which is of size $|\mathbb{D}_{in}|$. For this, we used the CUDA Thrust radix sort. Next, we describe our filtering proposal that reduce both this array's size and its sorting time.

### 3.3. Threshold-based Filtering

In order to increase the efficiency of the $k$NN computation, we propose a filtering techniques to decrease the sorting time of $k$NN candidates. By choosing a number of samples (documents) that have high similarity to the query, a threshold value can be chosen to discard all candidates whose similarity value are lower than it. In order to keep the $k$NN search exact, at least $k$ samples need to be chosen, and the $k$-th smallest of those needs to be the threshold. This way, at most $|\mathbb{D}_{in}|$ - $k$ candidates are filtered, in the best case where the true $k$ are sampled. In the worst case, when the smallest of all distances is in the sample, no candidate would be discarded since there is no distance smaller than it.

For a query document $x$, we exploit the numerator from the cosine similarity function: $cos(x, y) = \frac{x \,.\, y}{\|x\|\|y\|}$, aiming to maximize it with documents $y$ that share terms with high TF-IDF values. The main idea of our proposal is to sort the query's terms by TF-IDF in a descending order[2] and to execute one of the two possible proposed sampling methods. Also, to further increase the chance of selecting samples that will yield a higher similarity, we sort each inverted list by TF-IDF as well, as shown later in our proposal's flowchart.

#### 3.3.1. Sampling Method #1

The first proposed method is based on choosing the documents' Ids from the inverted index list of the first query term, until the number of samples is completed. If it is not enough to complete the sample size, the documents of the list of the next term are chosen; if it still does not complete, the choice is performed randomly. A set data structure is used to ensure that the samples have distinct Ids. This also applies for sampling method #2.

#### 3.3.2. Sampling Method #2

The second proposed method receives a parameter $T$, which is the number of terms to be used. Then, the documents' Ids are chosen by doing a round-robin on the $T$ inverted lists of the first $T$ terms of the query, until it completes the required number of samples; if it is not enough, the documents in the list of the next term $T + i$ are chosen, like in the first method; if it still does not complete the sample size, it chooses randomly. For both methods, the random sampling only happens when the query's terms corresponding inverted lists has few documents, or the query itself has few terms.

### 3.4. Fast Similarity Search for Text (FaSST-*k*NN)

Figure 1 shows the flowchart of our proposal FaSST-$k$NN, divided in four steps. First, the input data is read, then copied to GPU memory and is built as an inverted index. Then the inverted index lists are sorted on the GPU by using two stable sorts by key, an efficient approach to sort many sub arrays (Inverted lists). One with the term id as the key, then

[2]Meaning the first terms are likely to contribute more to the final cosine similarity value.

one with the index TF-IDF as key. This sorted inverted index is copied back to the CPU, where the samplings for queries are done. The next three steps are done for each query. The query is received, then sorted. Its terms are used in one of the proposed sampling methods, which will return a list of the chosen documents Ids, and copy it to the GPU. In the third step, the query is sent to the GPU, where the cosine distance is calculated against the inverted index, and written in a distance vector of size $|\mathbb{D}_{in}|$. By using the sample lists, only the chosen distances are copied to the CPU, where the threshold is returned with the QuickSelect $k$-selection algorithm. Finally, at step four, the threshold is sent to the GPU, where a GPU parallel compaction algorithm prunes all distances smaller than the threshold, and writes the distances greater than it at a smaller vector. This vector is sorted with Thrust radix sort, then the first $k$ are copied to the CPU, and the corresponding document Ids are printed.

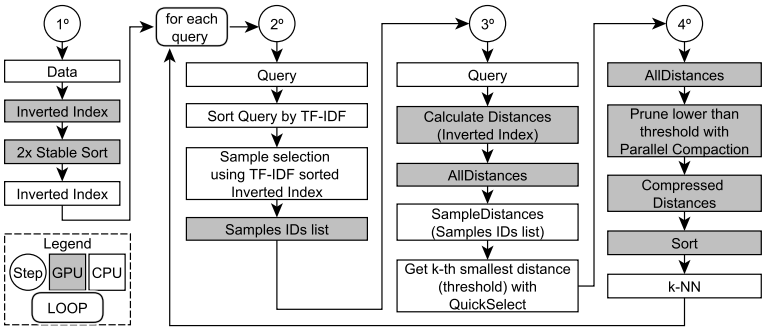



**Figure 1. FaSST-$k$NN flowchart.**

### 3.5. Multi-Query *k*NN Search

The FaSST-$k$NN algorithm was designed having in mind a manycore architecture (accelerator) with (global) shared memory. It exploits data parallelism when processing a single query, and uses a single accelerator. It does that by making use of thousands of threads to index the dataset and to find the nearest neighbors of a single query document. However, in many situations, the $k$NN search has to be invoked repeatedly, to deal with the processing of many queries. The FaSST-$k$NN algorithm can handle that by processing the queries one after another, once the input data has been indexed. This streaming operation requires that the $k$ nearest neighbors are returned before another query can be processed. In addition, a query specific memory allocation is needed for every query. Moreover, machines with more than one accelerator (GPU) can not take advantage of the extra computing power for the $k$NN search. This has motivated us to extend the FaSST-$k$NN to deal with multiple queries in a multi-GPU platform.

In the multi-GPU version, called mFaSST-$k$NN, task parallelism is exploited in addition to data parallelism. The data indexing step is performed by replicating the input data $\mathcal{E}$ in each of the $g$ available GPUs and then, in parallel, creating $g$ copies of the inverted-index. Thus, each GPU receives the same input and they all produce the same inverted-index in their memory. Next, each GPU performs the $k$NN search by dynamically requesting queries, so that there is no load imbalance between GPUs when queries sizes differ greatly. This is possible since the queries are completely independent of each other. Since the queries are of different size, we preallocate memory based on the biggest

query, i.e. the document with the largest number of terms. This saves us a lot of time since GPU memory allocation can be very costly.

## 4. Experimental Evaluation

The experimental work was conducted on a machine running Debian 9.4, with a Intel Xeon E5-2620, 16GB of ECC RAM, and four GeForce Nvidia GTX Titan Black, with 6GB of RAM and 2,880 cuda cores each. The CPU code was compiled with GCC 6.3.0 while the GPU code used the compiler provided by the CUDA 9.0. All the codes targeted the native architecture and had the O3 optimization flag set. In order to consider the costs of all data transfers in our experiments, we report the wall times on a dedicated machine so as to rule out external factors, like high load caused by other processes. The reported numbers are average of 10 independent runs.

In order to evaluate the $k$NN search, we consider a large real-world textual dataset, MEDLINE, which has the characteristics of high dimensionality and sparsity. For it, we performed a traditional preprocessing task: we removed stopwords, using the standard SMART list, and applied a simple feature selection by removing terms with low "document frequency (DF)"[3]. Regarding term weighting, we used TF-IDF. The specific details of the resulting MEDLINE are: 268,766 terms, 861,454 documents and 30.88 density[4].

We compare the computation time to perform a $k$NN search using the following algorithms: (1) **FaSST-$k$NN**, (2) **mFaSST-$k$NN**, our GPU-based implementations of $k$NN; (3) **G-KNN**, a GPU $k$NN implementation using CUDA proposed by Rocha et al. [Rocha et al. 2015]. We chose G-KNN because it is the only exact similarity search implementation we have found that deals with high dimensionality and sparsity found in textual datasets like MEDLINE. We adopted $k = \{32, 64, 128, 256, 512\}$ and sample size as $2 \times k$ in all experiments[5] . To measure our proposals' performance, we selected 20% of the dataset as search queries (Around 172,290 documents).

### 4.1. Computational Time

Table 1 shows the total time to process all queries using ours and the baseline's $k$NN implementations. From here on, we refer the variations of FaSST-$k$NN as: "NoFilter"[6] for a version that does not use our proposed filtering; "Random" for a random sampling scheme; "M1" for sampling method 1; "M2-T$X$" for sampling method 2, where $X$ indicates the value of its parameter $T$ (the number of terms). We only show $T = \{2, 3, 4\}$ due to space restrictions.

As can be seen, our filtering methods shows a higher efficiency with lower $k$, going from 441 to below 200 seconds, since less time is spent on sampling. The random sampling had no advantage, showing that our methods works. M2 ended up selecting better samples than M1, having a lower processing time. In this test, the parameter $T = 3$ achieved its best overall performance. We show the speedup regarding M2-T3.

FaSST-$k$NN shows significant speedups over the G-KNN implementation, in comparison, reaching up to $37x$ for $k = 32$. This is mainly due to its implementation that

[3]We removed all terms that occur in less than six documents (i.e., DF<6).

[4]Density is the average number of terms in a document.

[5]A higher ratio yields a higher compaction ratio, but also a greater sampling time.

[6]This is like GT-$k$NN [Canuto et al. 2015], but with Thrust library functions for sorting and prefix-sum.

**Table 1. Query times in seconds and speedups to find the K nearest neighbors with 1 GPU.**

| | Query time | | | | | | | M2-T3 Speedup | |
|---|---|---|---|---|---|---|---|---|---|
| K | G-KNN | NoFilter | Random | M1 | M2-T2 | **M2-T3** | M2-T4 | G-KNN | NoFilter |
| 32 | 6657 | 441.09 | 521.71 | 188.49 | 180.35 | 178.88 | 176.52 | 37.21 | 2.47 |
| 64 | 6674 | 459.55 | 542.19 | 216.43 | 202.66 | 204.19 | 208.97 | 32.68 | 2.25 |
| 128 | 6694 | 486.10 | 581.88 | 262.78 | 258.83 | 256.30 | 257.48 | 26.12 | 1.90 |
| 256 | 6718 | 522.40 | 656.69 | 358.84 | 343.55 | 342.72 | 343.78 | 19.60 | 1.52 |
| 512 | 6741 | 586.62 | 850.89 | 523.76 | 517.08 | 515.24 | 522.43 | 13.08 | 1.14 |

**Table 2. Query times in seconds and speedups to find the K nearest neighbors with 2 and 4 GPUs.**

| | 2 GPUs | | | | 4 GPUs | | | |
|---|---|---|---|---|---|---|---|---|
| | Query time | | Speedup M2-T3 | | Query time | | Speedup M2-T3 | |
| K | NoFilter | M2-T3 | NoFilter | 1-M2-T3 | NoFilter | M2-T3 | NoFilter | 1-M2-T3 |
| 32 | 229.34 | 87.69 | 2.62 | 2.04 | 115.50 | 45.13 | 2.56 | 3.96 |
| 64 | 233.18 | 104.28 | 2.24 | 1.96 | 117.45 | 51.29 | 2.29 | 3.98 |
| 128 | 241.76 | 126.99 | 1.90 | 2.02 | 121.54 | 63.32 | 1.92 | 4.05 |
| 256 | 258.13 | 173.07 | 1.49 | 1.98 | 127.38 | 85.19 | 1.50 | 4.02 |
| 512 | 293.43 | 257.70 | 1.14 | 2.00 | 142.45 | 127.36 | 1.12 | 4.05 |

computes the distance to all documents, while ours use an inverted index and the filter technique. Our best method achieved a speedup of $2.4x$ when comparing with NoFilter and $k = 32$, showing that the filtering could greatly compact the distance vector. For higher $k$ the speedup gets lower, since more time is spent on sampling on the CPU.

Table 2 shows the time and speedup for mFaSST-$k$NN with 2 and 4 GPUs, comparing NoFilter and M2-T3. The performance of M2-T3 over NoFilter remained similar to when 1 GPU was used, and over M2-T3 with 1 GPU (1-M2-T3) the speedup was around the ideal of $2x$ and $4x$, for 2 and 4 GPUs, respectively. This shows that our GPU $k$NN search scales well with more devices[7].

## 4.2. Runtime Profiling

We show the impact of sorting the inverted index before doing the sampling method on Table 3. For $k = 32$, the compaction ratio up to more than 3 times when using the sorted version, while for $k = 512$ it increases up to 1.6 times. For $k = \{62, 128, 256\}$ this value decreased from $3x$ to $1.6x$ as $k$ increased. This confirms that our proposals benefits further from this sorting of the data. The random sampling achieved only 1.05x of compaction.

The FaSST-$k$NN and mFaSST-$k$NN sampling and sorting times are shown in Tables 4 and 5. It represents the sum of time spent in these operations in each query. Since the sampling is done on the CPU, we can see that the time spent on it is quite high, specially for $k = 512$, reaching the hundreds of seconds for sampling method 2 (M2). This is partially due to associating a single CPU thread to a GPU, lacking any multicore parallelism or overlap of data transfer and computation. Its time decreases almost linearly with the number of GPUs. It also shows that, despite M2 taking more time in sampling, the total and sorting times decreased enough to make it the better method.

Table 5 shows that the sorting time decreases greatly with our proposed methods, going from 405 to just 20 seconds with $k = 32$ and the best method M2-T3. As $k$ goes up to 512, the gain over NoFilter decreases, since the compaction ratio also decreases when using the selected sample size of $2 \times k$.

Although not listed in any table, the CPU I/O, indexing, and inverted index sorting times, are 12, 1 and 1.2 second, respectively. For 1 GPU, the threshold selection time was

[7] An ideal speedup is also expected for G-KNN if it used more GPUs.

**Table 3. Impact of the sorted inverted index on the compaction ratio.**

| K | Data state | M1 | M2-T2 | M2-T3 | M2-T4 |
|---|---|---|---|---|---|
| 32 | unsorted | 107.19x | 106.42x | 90.73x | 77.44x |
| 32 | sorted | 214.78x | 278.27x | 277.80x | 268.98x |
| 512 | unsorted | 10.09x | 41.49x | 39.66x | 37.29x |
| 512 | sorted | 16.68x | 51.92x | 52.92x | 53.40x |

**Table 4. Sum of sampling times in seconds.**

| | 1 GPU | | | | | 2 GPUs | | | 4 GPUs | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| K | Random | M1 | M2-T2 | M2-T3 | M2-T4 | Random | M1 | M2-T3 | Random | M1 | M2-T3 |
| 32 | 6.18 | 24.42 | 24.12 | 25.07 | 22.34 | 3.28 | 11.00 | 11.36 | 1.62 | 5.33 | 5.75 |
| 64 | 10.36 | 28.49 | 27.53 | 29.51 | 31.11 | 5.30 | 13.65 | 14.44 | 2.62 | 6.48 | 6.82 |
| 128 | 18.11 | 34.67 | 38.96 | 40.01 | 39.79 | 9.41 | 17.09 | 19.34 | 4.63 | 7.97 | 9.06 |
| 256 | 31.84 | 49.29 | 56.89 | 58.29 | 57.85 | 17.80 | 24.74 | 29.90 | 9.35 | 11.71 | 14.17 |
| 512 | 58.11 | 76.01 | 96.22 | 102.19 | 106.88 | 35.42 | 40.68 | 51.99 | 19.34 | 20.04 | 24.81 |

**Table 5. Sum of sorting times in seconds.**

| | 1 GPU | | | | | 2 GPUs | | | 4 GPUs | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| K | NoFilter | M1 | M2-T2 | M2-T3 | M2-T4 | NoFilter | M1 | M2-T3 | NoFilter | M1 | M2-T3 |
| 32 | 405.73 | 26.78 | 21.66 | 20.57 | 21.82 | 200.07 | 13.87 | 10.68 | 107.32 | 7.09 | 5.70 |
| 64 | 406.39 | 35.76 | 28.47 | 27.13 | 28.44 | 201.20 | 18.50 | 14.46 | 106.62 | 9.52 | 7.30 |
| 128 | 405.57 | 49.36 | 41.78 | 39.15 | 40.21 | 201.89 | 26.32 | 20.11 | 105.85 | 14.02 | 10.63 |
| 256 | 406.13 | 74.66 | 59.34 | 57.62 | 57.92 | 202.39 | 39.34 | 30.37 | 103.40 | 21.52 | 16.04 |
| 512 | 408.87 | 112.74 | 85.96 | 84.62 | 86.71 | 202.72 | 60.98 | 44.91 | 101.20 | 33.10 | 24.57 |

1 and 6 seconds for $k = 32$ and $k = 512$, respectively. And the compaction time was 24 and 38 seconds, for these same values of $k$. With multi-GPUs these times got almost ideals speedups. The FaSST-$k$NN performs multiple queries exceptionally well but it excels at single query $k$NN search. Once the dataset has been read, moved to the GPU and indexed, subsequent queries can be processed very fast. Considering 1 GPU and $k = 512$, FaSST-$k$NN takes 3 milliseconds[8] at average to process a single query, making it suitable for on-the-fly top $k$ search using real datasets.

### 4.3. Final remarks

The FaSST-$k$NN is a very fast and scalable GPU-based tool for computing the top $k$ nearest in high dimensional and sparse data. However, it has some limitations. First, it is tailored to textual data that can be efficiently represented using an inverted index. It also requires a pre-processing of the dataset to conform to the input format. Second, the tool can only be ran in machines with NVIDIA accelerators, since it has been developed using CUDA. Third, although we were able to process large datasets, the GPU memory limits the maximum size to only a few tens of gigabytes. Fourth, the inverted index is not distributed over the GPU's memory but replicated in each memory, which further increases the memory problem. Finally, it can process similarity search real fast due to its filtering scheme, specially for lower $k$ values, although a good portion of the time is spent on the CPU when $k$ and the sample size are higher.

## 5. Conclusion

Intensive use of $k$NN, combined with the high dimensionality and sparsity of textual data, makes it a challenging computational task. We have presented a fine-grained parallel algorithm and a very fast and scalable GPU-based approach for computing the top $k$ nearest neighbors search in textual datasets. Different from other GPU-based $k$NN implementations, we avoid comparing the query document with all training documents. Instead, we

[8]We calculated with the time of method 2 with $T = 3$ and the size of the query set (172,290).

use an inverted index in the GPU that is used to quickly find the documents sharing terms with the query document. Although the index does not allow a regular and predictable access to the data, a load balancing strategy is used to evenly distributed the computation among thousand threads in the GPU. Furthermore, we proposed a filtering technique that decreases the sorting time of $k$NN candidates. Our filtering method allows the removal of documents with similarity less than a threshold so that our algorithm spends less time in the sorting phase of the nearest $k$ documents. Our proposal was extended to exploit multi-GPU platforms thus permitting the processing of multiple queries in parallel. We tested our approach in a very memory-demanding and time-consuming task which requires intensive and recurrent execution of $k$NN. Our results show very significant gains in speedup when compared to our baselines. In the future, we plan to apply our solution to different information retrieval tasks such as document categorization, document clustering, and recommender systems.

## References

Alewiwi, M., Örencik, C., and Savas, E. (2016). Efficient top-k similarity document search utilizing distributed file systems and cosine similarity. *Cluster Computing*.

Canuto, S., Gonçalves, M., Santos, W., Rosa, T., and Martins, W. (2015). An efficient and scalable metafeature-based document classification approach based on massively parallel computing. In *SIGIR*. ACM.

Chen, G., Ding, Y., and Shen, X. (2017). Sweet knn: An efficient knn on gpu through reconciliation between redundancy removal and regularity. In *ICDE*. IEEE.

Friedman, J. H., Bentley, J. L., and Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *In TOMS. ACM*.

Gieseke, F., Heinermann, J., Oancea, C., and Igel, C. (2014). Buffer kd trees: processing massive nearest neighbor queries on gpus. In *ICML*.

Gutiérrez, P. D., Lastra, M., Bacardit, J., Benítez, J. M., and Herrera, F. (2016). Gpu-sme-knn: Scalable and memory efficient knn and lazy learning using gpus. *Information Sciences*, 373:165–182.

Johnson, J., Douze, M., and Jégou, H. (2017). Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*.

Matsumoto, T. and Yiu, M. L. (2015). Accelerating exact similarity search on cpu-gpu systems. In *2015 IEEE International Conference on Data Mining*.

Paulevé, L., Jégou, H., and Amsaleg, L. (2010). Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*.

Rocha, L., Ramos, G., Chaves, R., Sachetto, R., Madeira, D., Viegas, F., Andrade, G., Daniel, S., Gonçalves, M., and Ferreira, R. (2015). G-knn: an efficient document classification algorithm for sparse datasets on gpus using knn. In *SIGAPP*. ACM.

Salton, G., Wong, A., and Yang, C.-S. (1975). A vector space model for automatic indexing. *Communications of the ACM*.

Wang, Y., Shrivastava, A., Wang, J., and Ryu, J. (2018). Randomized algorithms accelerated over cpu-gpu for ultra-high dimensional similarity search. SIGMOD '18. ACM.

# Análise das oportunidades de Otimização para Ambientes Intel® Xeon Phi ™ e Intel® Xeon Scalable Processors™ de um Método Numérico para o Escoamento Bifásico de Fluidos em Meios Porosos

**Thiago Teixeira**[1] **, Frederico L. Cabral**[1]**, Carla Osthoff**[1]**, Roberto P. Souto**[1]**, Márcio R. Borges**[1]

[1]Laboratorio Nacional de Computacao Cientifica (LNCC)
Av. Getúlio Vargas, 333. Quitandinha – 25651-075 – Petrópolis – RJ – Brasil

{tteixeira,fcabral,osthoff,rpsouto,mrborges}@lncc.br

***Abstract.*** *This paper presents optimizations of a numerical method for the biphasic flow of fluids in porous media, for Intel® Xeon Phi ™ and Intel® Xeon Scalable Processors™. Intel® Parallel Studios XE suite tools were used in the study of three method implementations (naive, EWS-Sync and RC). The EWS-Sync implementation consists of replacing the OpenMP barriers by an explicit thread synchronization mechanism. By avoiding barriers, by explicitly synchronizing threads waiting only for their adjacent ones in execution, the spin time was reduced. These implementations obtained a Speedup of 27x, compared to a serial execution, in multi-core and manycore architectures.*

***Resumo.*** *Este artigo apresenta otimizações de um método numérico para o escoamento bifásico de fluidos em meios porosos, voltado à execução paralela em ambientes Intel® Xeon Phi ™ e Intel® Xeon Scalable Processors™. As ferramentas do suíte Intel® Parallel Studios XE, foram utilizadas no estudo de três implementações do método (naive, EWS-Sync e RC). A implementação EWS-Sync consiste em substituir as barreiras do OpenMP por um mecanismo explícito de sincronismo entre threads. Ao evitar barreiras, sincronizando explicitamente threads que aguardam apenas suas adjacentes na execução, o tempo de spin foi reduzido. Estas implementações obtiveram Speedup de 27x, comparado a execução serial, tanto em arquiteturas multi-core quanto em manycore.*

## 1. Introdução

Dentro de diversas áreas da engenharia e ciências aplicadas, existe um grande interesse no desenvolvimento de modelos matemáticos e métodos computacionais para a simulação de escoamentos em meios porosos. Na engenharia de Petróleo, a otimização dos processos de recuperação de hidrocarbonetos está intimamente relacionada com a simulação precisa destes processos. Para isto, diversos fatores devem ser adequados e considerados nos modelos físico-matemático e numéricos, como por exemplo a troca de massa e momento linear das fases que escoam, suas relações de capilaridade e mobilidade, a estabilidade dos poços de produção e injeção, dentre inúmeros outros [Correa 2013].

O modelo computacional deste estudo apresenta uma metodologia numérica, proposta em [Correa 2013], para a simulação do escoamento bifásico (água e óleo) em um

reservatório rígido altamente heterogêneo. Este problema é modelado por um sistema de equações diferenciais parciais, basicamente composto por um subsistema elíptico para a determinação do campo de velocidades e uma equação hiperbólica não linear para o transporte das fases que escoam (equação da saturação). Do ponto de vista numérico, o modelo propõe a aplicação de um método de elementos finitos localmente conservativo para a velocidade da mistura e um método de volumes finitos não-oscilatório de alta ordem, baseado em esquemas centrais, para a equação hiperbólica não-linear que governa a saturação das fases.

Este artigo apresenta a análise das oportunidades de otimização de código do método numérico em questão para as plataformas Intel® Xeon Phi ™ e Intel® Xeon Scalable Processors™, seguindo a metodologia de otimização guiada por perfilagem (*profile guided optimization*), com o uso das ferramentas do suíte Intel ® Parallel Studio ™, como o VTune Amplifier™, Thread Advisor™ e Parallel Advisor™.

Este artigo é organizado da seguinte forma: a Seção 2 descreve os trabalhos relacionados; a Seção 3 descreve o modelo matemático; a Seção 4 mostra o resultado da análise do perfil do código com o auxílio das ferramentas de perfilagem; a Seção 5 apresenta as propostas de otimização de acordo com as oportunidades identificadas; a Seção 6 mostra o resultados dos experimentos realizados e por fim, na Seção 7 faz a conclusões e propõe os trabalhos futuros.

## 2. Trabalhos Relacionados

Dois paradigmas de sincronização estão sendo utilizados no problema de sincronismo na computação *multi-thread*: São as estratégias *work sharing* e *work stealing*. Na estratégia *Work-stealing*, processadores subutilizados tentam ”roubar”*treads* de outros processadores. A ideia de *work-stealing* é datada desde o trabalho proposto por Burton e Sleep [Burton 1981]. Esses autores apresentam um modelo para a execução simultânea de árvores de processos, o qual provê a base para combinar a criação de novas tarefas aos recursos disponíveis [Burton 1981]. Eles também apresentam a interpretação de uma topologia para o suporte de árvores de processos virtuais em uma rede física. Esses autores apontam os benefícios do paradigma do *work-stealing* para a redução do espaço e da comunicação em um contexto paralelo. Além disso, muitos pesquisadores já implementaram variantes desta estratégia. Blumofe e Leiserson [Blumofe 1999] analisaram um algoritmo para sincronismo na computação *multi-threaded fully strict* (bem estruturado).

No paradigma de *work-sharing*, sempre que um processador gerar uma nova *thread*, o sincronizador efetuará uma migração de um número de *threads* para outro processador compartilhando o trabalho entre processadores subutilizados no intuito de reduzir o desbalanceamento. De forma intuitiva, podemos observar que, ao comparar com a estratégia *work-sharing*, as *threads* migram com menos frequência quando a estratégia de *work-stealing* é empregada. Quando muitos processadores possuem tarefas para concluir, o sincronizador da *work-stealing* não migra as *threads* entre os processadores, não obstante, o sincronizador da *work-sharing* sempre migra as *threads* entre os processadores.

O pesquisador Penna et al. [Penna et al. 2017] propôs uma estratégia de sincronismo de *loop workload-aware* para *loops* paralelos irregulares em que interações são independentes. Estes autores aplicaram sua estratégia em uma máquina NUMA de larga escala usando um *kernel* sintético.

Várias estratégias para aperfeiçoar a performance, em aceleradores Intel® Xeon Phi™, vem sendo propostas por pesquisadores. Essas técnicas de resolução do problema buscam superar o desafio de um *speedup* linear utilizando a arquitetura, principalmente em implementações OpenMP. Um exemplo está em, Ma et al. [H. Ma et al. 2009] onde foram propostas estratégias para otimizar as construções de barreiras implícitas do OpenMP. Estes autores revelam como remover as construções de barreiras implícitas do OpenMP quando não há nenhuma dependência de dados. A segunda estratégia desses autores usa uma sincronização de espera ocupada. Suas implementações em OpenMP obtiveram resultados melhores do que as estratégicas básicas do OpenMP.

Caballero et al. [Caballero et al. 2013] introduz uma barreira baseada em árvore que usa localidade de *cache* em conjunto com as instruções SIMD. Com a estratégia apresentada pelos autores, foi possível obter um *speedup* de 2,84x comparada a barreira básica do OpenMP no EPCC *barrier micro-benchmark*. [Cabral et al. 2014] avaliou o método original Hopmoc em diferentes paradigmas de programação paralela, contudo, esta avaliação não foi realizada em aceleradores Intel® Xeon Phi™.

## 3. O Modelo Matemático

Seja $\Omega \subset \mathbb{R}^3$ um domínio conexo, aberto e limitado e $I$ um tempo de intervalo. Consideramos a lei de conservação escalar da seguinte forma:

$$\phi \frac{\partial s}{\partial t} + \nabla \cdot \mathbf{f} = 0 \text{ in } \Omega \times I. \tag{1}$$

Onde $\phi : \Omega \Rightarrow (0, \phi^{max}]$ é o coeficiente de armazenamento, $s : \Omega \times I \rightarrow Im\{s\} = [s^{min}, s^{max}]$ é a função escalar, e a função vetorial $\mathbf{f} : Im\{s\} \rightarrow \mathbb{R}^3$ é o fluxo da quantidade conservada $s$. Particularmente, para os testes considerados neste trabalho

$$\mathbf{f} = s\mathbf{v} = \left\{ \begin{array}{c} s \\ 0 \\ 0 \end{array} \right\}. \tag{2}$$

O método numérico utilizado para aproximar a solução da equação (1) é descrito, em detalhes, em [Correa 2013].

## 4. Análise do Perfil do Código

Para que se possa compreender o perfil do código, é necessário antes conhecer sua estrutura básica, com cada um de seus módulos principais, suas funções e a relação chamado-chamador entre eles, bem como a complexidade computacional de cada um. O Algoritmo 1 mostra um trecho de pseudocódigo do módulo principal, o módulo de transporte.

O módulo de transporte é responsável por preparar as variáveis para a função **KTDD_RT**, chamada dentro do Algoritmo 1 e representada no Algoritmo 2, que executa o método *Kurganov-Tadmor* DxD. Para isso, é necessário que seja definido o número de *Courant* (cr), as velocidades locais, que são resolvidas pela função **MaxDer**, e o $\Delta t$ calculado dinâmicamente de forma a respeitar a restrição de CFL.

Neste trecho de pseudocódigo pode-se observar que aparece uma região paralela OpenMP de modo que a função **dividirTrabalho**, chamada dentro do Algoritmo 2, se

```
1  begin
2      cr ← 0.125;
3      #pragma omp parallel;
4      {
5        call maxDer();
6      }
7      Δt ← nDias / numPassosTransp;
8      call KTDD_RT();
9  end
```

**Algoritmo 1:** Um trecho do código do módulo transporte

```
1  begin
2      #pragma omp parallel;
3      {
4       call dividirTrabalho (iniTrabalho, fimtrabalho);
5       for (passo ← 1; passo ≤ numPassosTransp; passo ← passo + +) do
6           call derivaU (iniTrabalho, fimTrabalho) ;
7           #pragma omp barrier;
8           call fkt (iniTrabalho, fimTrabalho);
9           #pragma omp single;
10            tTransporte = tTransporte + Δt
11     end
12     call derivaU (iniTrabalho, fimTrabalho) ;
13     }
14 end
```

**Algoritmo 2:** Um trecho do código da função KTDD_RT que executa o método Kurganov-Tadmor DxD

torna responsável por calcular, para cada *thread* criada, a região dentro da malha que cada uma irá trabalhar. Este é um tipo de divisão explícita de trabalho, onde ao invés de deixar a cargo do *framework* do OpenMP distribuir as iterações de um *loop* entre as *threads*, isto é feito explicitamente através de variáveis de controle **iniTrabalho** e **fimTrabalho**.

A função **derivaU**, chamada dentro do Algoritmo 2, calcula as derivadas em cada elemento para que a função **fkt**, chamada dentro do Algoritmo 2, possa calcular o fluxo do escoamento para o próximo passo de tempo. Como o cálculo do fluxo para um elemento depende das derivadas de alguns elementos vizinhos, existe uma barreira do OpenMP para impedir que uma thread continue seu trabalho e tente calcular o fluxo de um elemento sem que as *threads* das quais ela depende tenham terminado o cálculo das derivadas.

A função **vizinhanca**, chamada dentro do Algoritmo 3 e representada no Algoritmo 4, calcula diversos fatores como porosidade, permeabilidade e velocidade do escoamento nos elementos adjacentes, para que em seguida, a saturação de cada elemento possa ser obtida pela função **udd_rt**, chamada dentro do Algoritmo 3, que por sua vez

```
1 begin
2     for (numElem ← iniTrabalho; numElem ≤ fimTrabalho;
       numElem + +) do
3         call vizinhanca (...) ;
4         [...] ;
5         satElemAnt (numElem) ← udd_rt;
6     end
7 end
```

**Algoritmo 3:** Um trecho do código da função fkt que calcula os fluxos do transporte para a próxima iteração

executa uma iteração do método *Runge-Kutta* de terceira ordem.

```
1  begin
       // Calcula a lista dos elementos
2      if Caso 3D then
           // Identificação dos elementos 3D
3      end
       // Calcula a porosidade nos elementos vizinhos
4      if Caso 3D then
           // Porosidade 3D
5      end
       // Calcula as permeabilidades
6      if Caso 3D then
           // Permeabilidade 3D
7      end
8      uc = satElem(elemento)
       // Solução nos lados:  reconstrução linear
9      if Caso 3D then
           // reconstrução linear 3D
10     end
       // Velocidades nos lados dos elementos
11     if Caso 3D then
           // Velocidades 3D
12     end
       // Soluções nos elementos vizinhos
13     for (k ← 1; k ≤ numeroDeLadosDoElemento; k + +) do
           // Calcula a saturação
14     end
15 end
```

**Algoritmo 4:** Um trecho do código da função Vizinhança

### 4.1. Analise de desempenho do código

O código, escrito em *FORTRAN* e desenvolvido para este modelo computacional, é dividido em 14 módulos que contém uma variedade de funcionalidades matemáticas e físicas. O mesmo já foi apresentado com otimizações de paralelização através de diretivas OpenMP. Com o uso das ferramentas do suíte Intel ® Parallel Studio ™, como o VTune Amplifier™, Thread Advisor™ e Parallel Advisor™, pôde ser identificado um dos *hotspots* do código e, neste caso, foi realizada uma análise no módulo de maior gasto computacional, chamado de **transporte.F90**, onde se encontra a solução do problema numérico.

Para analisar o *loop* principal dentro da função **KTDD_RT** representada pelo Algoritmo 2, via ferramenta Thread Advisor™, foram criadas anotações à serem inseridas diretamente no código fonte. Elas foram posicionadas para marcar os locais do programa de execução serial onde a ferramenta poderá assumir se ocorrerá alguma execução paralela ou sincronização. As anotações são colocadas propositalmente em regiões paralelas do código para que a ferramenta possa examinar em detalhes essa execução. Essa análise tornou possível a visibilidade dessas funções que possuem o maior custo computacional e, por esse motivo, são os alvos para a paralelização nas plataformas Intel® Xeon Phi ™ e Intel® Xeon Scalable Processors™.

## 5. Otimização

Os experimentos conduzidos com a implementação do código original em OpenMP, que iremos chamar de OpenMp *naive*, foram realizados em uma máquina que contém Intel Xeon CPU E5-2698 v3 @ 2.30GHz com 32 cores físicos, e em uma máquina que contém Intel Xeon Phi (KNL) Intel ® Xeon Phi™ CPU 7250F @ 1.40GHz com 72 cores físicos e 4 *threads*/*core*.

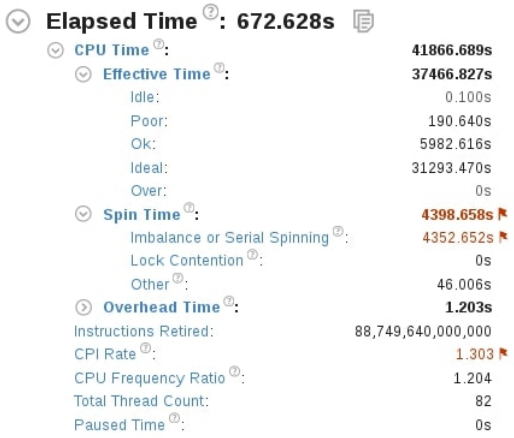



**Figura 1. Análise *Advanced Hotspots* via VTune Amplifier™ com execução do código *naive* já com diretivas OpenMP, utilizando 64 *threads* no processador Xeon.**

Pudemos observar através do Intel ® VTune Amplifier™, que grande parte do tempo de execução da função é gasto em operações de sincronização, onde em múltiplos momentos as *threads*, que terminam a execução de um *loop* dentro de uma função, aguardam o término de execução de cada uma das outras *threads* para dar continuidade a

execução do código. A analise realizada na Figura 1 foi a *Advanced Hotspots* e apresentou um alto índice de *Spin Time* (desbalanceamento de carga ou tempo de execução serial) causado pelo mecanismo de sincronização do OpenMP e uma alto índice de *clock ticks* por instrução *retired* (**CPI** *rate*).

Em geral, o **CPI** é a primeira métrica a ser verificada na performance da aplicação durante o aperfeiçoamento do código. Essa métrica é determinada ao dividir o número de ciclos do processador (*clock ticks*) pelo número de instruções *retired* (que já terminaram). O valor de **CPI** de uma aplicação é o indicador de quanta latência sua execução possui. Um alto índice de **CPI** significa mais latência, geralmente, durante a execução. Isso significa que a aplicação demorou mais *clock tick* por instrução *retired*. Geralmente o código, o processador e a configuração do sistema operacional influenciam no **CPI** de uma carga de trabalho, e o valor 0,75 é uma valor razoável (máximo) para essa métrica.

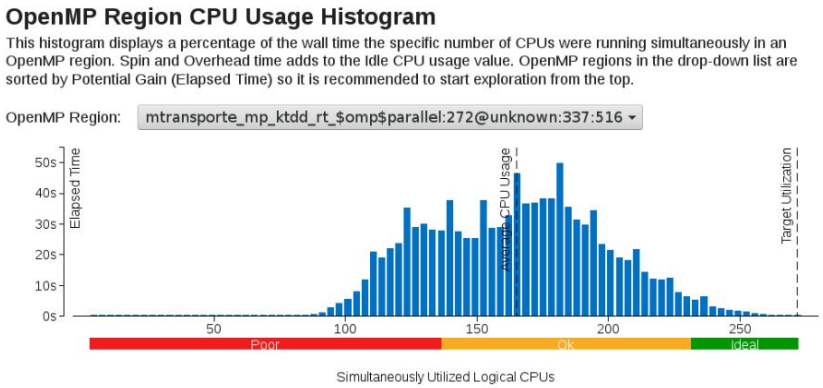



**Figura 2. Histograma da carga de trabalho dividida entre threads com execução de 272 *threads* em coprocessadores Intel Xeon Phi (KNL) com execução do código *naive*)**

## 5.1. Sincronização entre Threads Adjacentes: EWS-Sync

Um tempo baixo de *spin* pode ser desejado ao invés do aumento de trocas de contexto de *threads*. Um tempo alto de *spin*, entretanto, pode diminuir o tempo produtivo de trabalho. Neste intuito foi criada uma estratégia para alterar o padrão de escalonamento das *threads* geradas pela **API** do compilador através do modelo de programação de memória compartilhada padrão OpenMP de forma a diminuir o tempo de sincronização entre elas e o aumento no desempenho.

A estratégia, que foi apresentada em [Cabral et al. 2018] consiste em substituir as barreiras padrões do OpenMP por uma abordagem de *Locks* explicitamente programados, onde dentro da divisão da carga de trabalho em *threads*, ela apenas aguarda suas *threads* vizinhas para continuar sua execução, ao invés de aguardar o término da execução de todas as *threads* como nas barreiras padrões do OpenMP, essa estratégia se chama *Explicit work sharing with explicit synchronization* (EWS-Sync).

Os experimentos conduzidos com a implementação EWS-Sync do código foram realizados em uma máquina que contém Intel ® Xeon Phi (KNL) Intel ® Xeon Phi @ 1.40GHz com 68 *cores* físicos com 4 *threads*/*core* e uma máquina que contém Intel Xeon CPU E5-2698 v3 @ 2.30GHz com 32 cores físicos. Pode-se observar na Figura 2 que

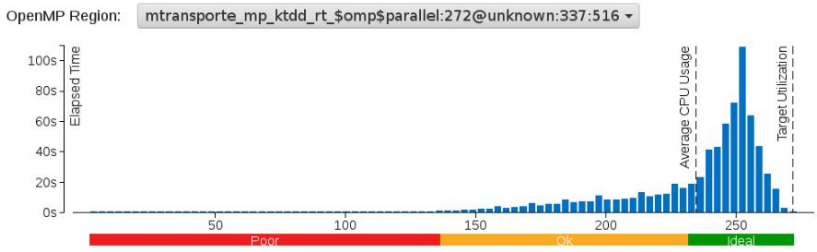



**Figura 3. Histograma da carga de trabalho dividida entre threads com execução de 272 *threads* em coprocessadores Intel Xeon Phi (KNL) com execução do código já com estratégia EWS-Sync)**

a execução apresenta um desbalanceamento significativo quanto ao número de *threads* trabalhando simultaneamente para a versão *naive*. O histograma mostra que a distribuição do número de *threads* simultâneas fica ao redor de 154, onde se encontra a média de uso da CPU. Já na Figura 3 é possível perceber que a média de uso da CPU é aumentada para 245 o que significa que mais *threads* são executadas simultaneamente. O impacto desta estratégia também pode ser observado ao se comparar as métricas geradas pelo Intel ® VTune ™. Na Figura 1 percebe-se que o *spin time* é alto, em torno de 4398 segundos, o que significa bastante tempo sendo gasto na barreira do OpenMP, enquanto na Figura 4 o *spin time* fica reduzido a 1480 segundos, o que significa que é menos tempo gasto nas barreiras do OpenMP. Isto acontece pelo fato da estratégia EWS-Sync consistir em se substituir a barreira que aparece na linha 7 do Algoritmo 2 por um mecanismo de *lock* que permite que uma determinada *thread* aguarde apenas as threads adjacentes para continuar adiante, conforme proposto em [Cabral et al. 2018].

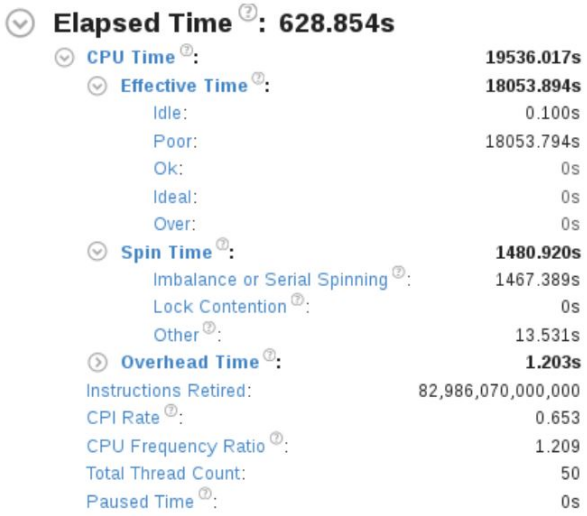



**Figura 4. Análise *Advanced Hotspots* via VTune Amplifier™ com execução do código com a estratégia EWS-sync, utilizando 64 *threads* no processador Xeon.**

Percebe-se que a carga de trabalho foi melhor balanceada com a nova estratégia, o que é importante para o ganho de desempenho final no código. Houve uma boa redução de **CPI** *rate* e no tempo de *spin* obtidos pela estratégia EWS-Sync.

### 5.2. Reestruturação do Algoritmo

Apesar de realizar cálculos de dados 3D, o código foi criado para realizar cálculos de dados em 2D, inicialmente. Desta forma, na visão de boas práticas para paralelização de códigos, o trabalho realizado não é bem otimizado. Em inúmeras áreas do código podemos ver condicionais aninhadas, tanto em outras condicionais, quanto em *loops* durante a execução do código, como pode ser visto no Algoritmo 4 que é referente a função **vizinhanca**, chamada dentro do Algoritmo 3. Isso causa um grande impacto no tempo de execução da função em que essas condicionais se encontram, tal qual no tempo de execução final do código. No caso desta modificação, que foi nomeada de Reestruturação de condicionais (RC), apesar de grande parte das funções do código possuírem condicionais similares, a função **vizinhaca** foi alvo para esta otimização. Esta função realiza o cálculo da porosidade dos elementos vizinhos ao que está sendo calculado, tanto como a permeabilidade entre outros cálculos e é utilizada diversas vezes durante a execução do código. Como pode ser observado no Algoritmo 4 , dentro da função **vizinhanca**, existem 5 condicionais vinculadas à dimensão do problema, que informa se a execução do código é no caso 2D ou 3D. A modificação consistiu em duplicar essa função, mantendo o mesmo nome na função original, e nomeando a função duplicada como **vizinhanca2D**. Foram removidas as condicionais de ambas as funções mantendo o que estava contido nas condicionais da função original, e removendo, da função duplicada, o que estava contido nessas condicionais, e foi criada uma condicional dentro da função **fkt**, chamada dentro do Algoritmo 2, antes da função **vizinhanca** ser chamada, como pode ser visto no Algoritmo 5.

```
1  begin
2      for (numElem ← iniTrabalho; numElem ≤ fimTrabalho;
       numElem + +) do
3          if Caso 3D then
4              call vizinhanca (...) ;
5          else
6              call vizinhanca2d (...) ;
7          end
8          [...] ;
9          satElemAnt (numElem) ← udd_rt;
10     end
11 end
```

**Algoritmo 5:** Um trecho do código da função fkt onde a estratégia de reestruturação foi empregada

## 6. Resultados Experimentais

As duas estratégias, EWS-Sync e RC, foram utilizadas em conjunto para a execução dos experimentos dentro da função **KTDD_RT**(Algoritmo 1). Eles foram executados em uma máquina que contém dois processadores Intel Xeon Platinum 8160 CPU @ 2.10GHz, com dois nós de 24 cores físicos, 2 *threads* por *core*. Assim, são dois *sockets* que se comunicam por um barramento UPI (Ultra Path Interconnect). Como pode ser observado

nas Figuras 5 e 7, os gráficos apresentam o tempo de execução e *speedup* do código, e nas Figuras 6 e 8, o tempo de execução e *speedup* da função **KTDD_RT**. Além da execução do código *naive*, três execuções foram realizadas utilizando afinidades de *threads* diferentes: *Balanced, Compact e Scatter*.

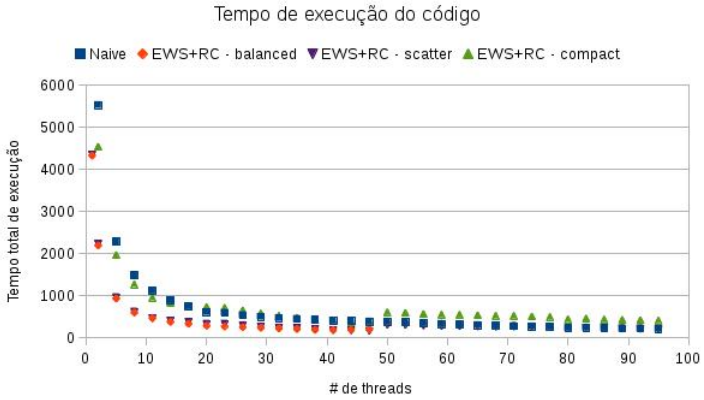



**Figura 5. Tabela com tempo de execução (em segundos) das estratégias apresentadas com a utilização de 95 *threads***

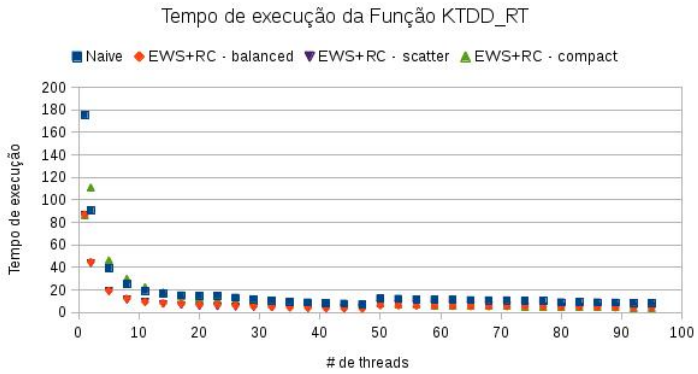



**Figura 6. Tabela com tempo de execução (em segundos) da função KTDD das estratégias apresentadas com a utilização de 95 *threads***

É possível observar que houve um ganho de desempenho de 27x levando em consideração o tempo de execução do código, com a afinidade de *threads Balanced* comparado ao código *naive*, executado de forma serial, que pode ser visualizado nas Figuras 5 e 7 que evidenciam as métricas referente ao tempo de execução e *speedup* do código. E houve um ganho de desempenho de 54x levando em consideração o tempo de execução da função **ktdd**, também comparado ao código *naive* com execução serial, visualizado nas Figuras 6 e 8 que evidenciam as métricas de tempo de execução e *speedup* da função. Todas as execuções foram realizadas utilizando de 1 a 95 *threads*.

Como a arquitetura *Skylake* contém dois *sockets* conectados por um barramento, a afinidade de *threads scatter*, que possui sua distribuição de *threads* feita entre todos os cores do sistema assim que qualquer *core* fique disponível, apresenta o pior desempenho das afinidades de *threads*. Como pode ser visto nas Figuras 7 e 8, o tráfego intenso no

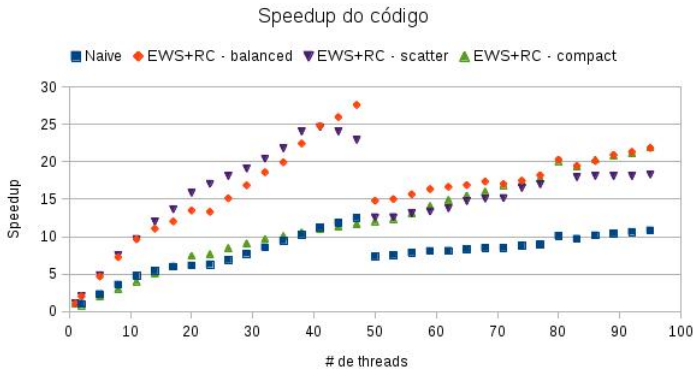



**Figura 7. Tabela com o *speed up* das estratégias apresentadas com a utilização de 95 *threads***

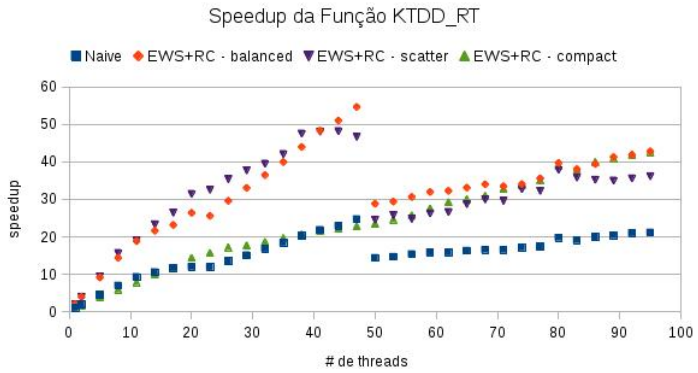



**Figura 8. Tabela com o *speed up* da função KTDD das estratégias apresentadas com a utilização de 95 *threads***

barramento gera uma latência muito alta nas execuções à partir de 47 *threads* causando uma queda no *speedup*. A afinidade *Compact* divide as *threads* de *software* para as *threads* de *hardware* da forma em que cada duas *threads* ocupam apenas um único *core*. Essa afinidade sobrecarrega alguns *cores* até mesmo quando outras *threads* estão disponíveis. Por esse motivo, o *speed up* é melhor quando usado um número pequeno de *threads*. A afinidade *Balanced* distribui as *threads* dentro de apenas um *socket* antes de designá-las para o segundo *socket*, com a diferença em que ele designa as *threads* de software para os *cores* físicos assim que eles estejam disponíveis em qualquer *socket*. A afinidade *Balanced* obteve o melhor resultado dentro das afinidades utilizadas. O motivo se deve, a necessidade de comunicação entre os *sockets* trazendo assim um *overhead* no processo.

## 7. Conclusão e Trabalhos Futuros

A implementação empregou a estratégia *explicit work-sharing* (EWS-Sync), com um mecanismo de sincronização específico, e a estratégia Reestruturação condicional (RC), realizando a remoção de condicionais aninhadas dentro de *loops* e de próprias condicionais da função **vizinhanca**. A execução das estratégias apresentadas para a implementação no código *naive* com diretivas de OpenMP, principalmente utilizando a afinidade de *threads balanced*, conquistaram *speedups* razoáveis tanto em arquiteturas *multi-core quanto* em

*manycore*.

Como trabalhos futuros, a reestruturação do código será continuada até que o módulo **Transporte.F90** esteja dentro dos padrões das boas práticas de paralelização, removendo assim, todas as condicionais aninhadas possíveis com o intuito de maior ganho de desempenho. Implementar o código em MPI/OpemMP para ambientes híbridos, como o supercomputador Santos Dumont (SDumont).

**Agradecimentos**

Este projeto teve apoio do CNPq. Gostaríamos de agradecer ao Núcleo de Computação Científica da Universidade Estadual Paulista (NCC/UNESP) por nos disponibilizar o uso do cluster multi-core heterogêneo para a execução dos nossos experimentos. Esses recursos foram parcialmente financiados pela Intel® via projetos intitulados Intel Parallel Computing Center, Modern Code Partner, e Intel/Unesp Center of Excellence in Machine Learning.

## Referências

Blumofe, R.D.; Leiserson, C. (1999). Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748.

Burton, F. W.; Sleep, M. R. (1981). Executing functional programs on a virtual tree of processors. *In Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture (Portsmouth, N.H., Oct.). ACM, New York, N.Y.*, pages 187–194.

Caballero, D., Duran, A., and Martorell., X. (2013). An OpenMP barrier using SIMD instructions for Intel Xeon Phi™ coprocessor. *OpenMP in the Era of Low Power Devices and Accelerators. IWOMP 2013. A.P. Rendell and B.M. Chapman M.S. Muller(Editors). Lecture Notes in Computer Science*, 8122:99–113.

Cabral, F., Osthoff, C., Kischinhevsky, M., and ao., D. B. (2014). Hybrid MPI/OpenMP/OpenACC implementations for the solution of convection diffusion equations with Hopmoc method. *Proceedings of 14th International Conference on Computational Science and Its Applications (ICCSA)*, pages 196–199.

Cabral, F. L., Osthoff, C., Costa, G., Gonzaga de Oliveira, S., Brandão, D., and Kischinhevsky, M. (2018). An openmp implementation of the tvd hopmoc method based on a synchronization mechanism using locks between adjacent threads on xeon phi (tm) accelerators. *Lecture Notes in Computer Science. Springer International Publishing*, 3:701–707.

Correa, M. R.; Borges, M. R. (2013). A semi-discrete central scheme for scalar hyperbolic conservation laws with heterogeneous storage coefficient and its applications to porous media flow. *International Journal for Numerical Methods in Fluids*, 73(3):205–224.

H. Ma, R. Z., Gao, X., and Zhang., Y. (2009). Barrier optimization for OpenMP program. *Proceedings of 10th ACIS Int. Conf. on Software Engineering, Artificial Intelligences, Networking, Parallel and Distributed Computing*, pages 495–500.

Penna, P., Castro, M., Plentz, P., Freitas, H., Broquedis, F., and Mehaut, J. (2017). A novel worload-aware loop scheduler for irregular parallel loops. *Braziliam Simposium of High Perfomance Computing.*, 11:527–536.

# MParCO: a Minimalist Parallel Framework for Combinatorial Optimization Applications

**Allberson B. de O. Dantas**[1] **, Ricardo C. Corrêa**[2] **,**
**Lucas B. de Vasconcelos**[3]

[1]IEAD, Universidade da Integração Internacional da Lusofonia Afro-Brasileira
Campus da Liberdade, Redenção, Brazil
allberson@unilab.edu.br

[2]Universidade Federal Rural do Rio de Janeiro
Campus Nova Iguaçu, Nova Iguaçu, Brazil
correa@ufrrj.br

[3]Centro Universitário Christus
Campus Dionisio Torres, Fortaleza, Brazil
lucas.batista@unichristus.edu.br

***Abstract.*** *This paper addresses the exploration of the potential of distributed parallelism in combinatorial optimization applications through a minimalist approach. In contrast to several parallel frameworks proposed in the last decades, specialized in parallelizing classical methods such as Branch-and-Bound and well-known metaheuristics, we propose a framework, so-called MParCO, that focuses essentially on assuring asynchronism, an essential ingredient for the performance of enumerative methods for combinatorial optimization problems.*

## 1. Introduction

A combinatorial optimization problem consists of finding certain arrangements (solutions) of some combinatorial objects among a large set of possible arrangements. A multitude of real-life problems can be seen as combinatorial optimization problems. These include, but are not limited to, cutting, packaging, scheduling, and graph problems. One way to solve such problems would simply be to list all possible solutions and save the least cost solution. However, for any problem of a minimally interesting (and useful) size, this method becomes impractical, since the number of possible solutions is very, very large. Although there are several well known techniques to deal with the complexity of such problems, such as linear and integer programming, and heuristic and approximate algorithms, such techniques are conceptually sequential, and run into the limited processing power of current processor architectures.

The difficulty of resolution exhibited by hard combinatorial optimization problems is intrinsically related to the large number of possible solutions. However, there is an intrinsic source of parallelism in such problems, which is linked to the possibility that several solutions can be enumerated in parallel. If several processing nodes are available, the enumeration can be accelerated by visiting several solutions in parallel. Following this principle, the natural strategy of parallelization is to distribute the solutions to be enumerated between the processing nodes. If this distribution could be done equitably *a priori*, the nodes could work independently (and totally asynchronously) in

their respective enumerations. In fact, asynchronism is an essential element for obtaining efficient parallel algorithms for enumerative methods for combinatorial optimization problems [Corrêa 2002, Crainic et al. 2006, Bader et al. 2005].

Based on the above, this work proposes a minimalist parallel framework for solving combinatorial optimization problems, so-called MParCO. Unlike most frameworks with the same purpose, which implement specific methods for enumerating solutions very difficult to extend, ours only worries about making the processing nodes work asynchronously, since, as mentioned before, asynchronism is a crucial element for the good performance expected during paralelization. Based on this, it currently implements two asynchronous distributed computing models, the event-driven model and the pulse-driven model [Corrêa and Barbosa 2009]. Although *ad hoc* parallelization strategies are commonly employed to solve certain combinatorial optimization problems in order to obtain the best possible performance, we expect to achieve competitive performance, taking into account the abstraction power and ease of programming obtained from a generic framework.

Summing up, the main contributions of this paper are the design and implementation of: (1) a minimalist parallel framework for combinatorial optimization applications; (2) a parallel Branch-and-Bound [Land and Doig 1960] algorithm implemented through the proposed framework; (3) a case-study to evaluate the performance of the proposed framework through an implementation of the the Maximum Independent Set (MIS) problem using the proposed parallel Branch-and-Bound algorithm.

## 2. Assynchronous Distributed Computing Models

The asynchronous distributed computing models employed currently in our framework work on an undirected graph $\Gamma = (N, L)$, where $N$ and $L$ correspond to the set of nodes of a *computational network* and the set of bidirectional channels between these nodes, respectively. Here, the notation $ij$ represents a channel between $i$ and $j$. We also define the neighborhood of a node $i$ as the set $N(i)$ of all nodes such that have a common channel with $i$. In notation, $N(i) = \{j \mid ij \in L\}$.

### 2.1. The Event-driven Model

The primordial notion of this model is the *event*, name given to the act of receiving a message and, in response to it, performing a computation in an atomic way, possibly by changing the local state of the node. The asynchronism in this case comes from the fact that computations evolve purely through the processing of events and sending of messages.

**Algorithm 1** below describes distributed event-driven computations. In such an algorithm, receiving a message $msg_i$ from a neighbor of the node $i$ changes the local state of $i$ through the execution of the function $EVENT_i$, that corresponds to a particular local computation performed by $i$. In addition to that, $EVENT_i$ generates a possibly empty set of messages $MSG_i$, where each message is destined to a neighbor of $i$. The first call of the function $EVENT_i$ consists of the spontaneous event, which is generated without message reception. The algorithm ends when every node has knowledge of the end of the distributed computation (line 4). In this work, we employ the strategy presented in [Misra and Chandy 1982] for detecting global termination.

**Algorithm 1**: Computation performed by node $i$ in the event-driven model

Input: Set $N(i)$ of neighbors of $i$ in $\Gamma$, initial local state of $i$
1. Record the initial local state of $i$
2. $EVENT_i, (-, MSG_i)$
3. Send each message in $MSG_i$ to its respective neighbor
4. **while** the global termination is not known by $i$ **do**
5. **if** a message $msg_i$ from a neighbor of $i$ has been received **then**
6. $EVENT_i(msg_i, MSG_i)$
7. Send each message in $MSG_i$ to its respective neighbor

## 2.2. The Pulse-driven Model

While the event-driven model is adequate to computations in which changes in local states happen exclusively by reaction to receptions of messages, in the pulse-driven model these state changes can even occur when no messages are received.

**Algorithm 2** below depicts pulse-driven computations. Through it, the evolution of a distributed computation, from the initial state to the final state, is guided by a mechanism that generates a sequence of pulses (*pulse generation mechanism*), which governs the evolution of the state of each node $i$. Such a mechanism is implemented through the abstract functions $getCurrent$ and $hasAdvanced$. $getCurrent$ is used to report the pulse generation mechanism that node $i$ has started its local computation associated with the most recently generate pulse. $hasAdvanced$, in turn, is a boolean function used by the pulse generation mechanism for signaling the generation of a new pulse, and if $hasAdvanced$ returns **true**, then it will not return **true** again before $getCurrent$ is called. $EVENT_i$ has the function of only obtaining relevant information from the received message, not generating messages. The node state transition is performed by the function $PULSE_i$, which receives as part of its input the current pulse number.

**Algorithm 2**: Computation performed by node $i$ in the pulse-driven model

Input: Set $N(i)$ of neighbors of $i$ in $\Gamma$, initial local state of $i$
1. $\ell_i \leftarrow getCurrent()$
2. $PULSE_i(\ell_i, MSG_i)$
3. Send each message in $MSG_i$ to its respective neighbor
4. **while** the global termination is not known by $i$ **do**
5. **if** $hasAdvanced()$ **then**
6. $\ell_i \leftarrow getCurrent()$
7. $PULSE_i(\ell_i, MSG_i)$
8. Send each message in $MSG_i$ to its respective neighbor
9. **if** a message $msg_i$ from a neighbor of $i$ has been received **then**
10. $EVENT_i(msg_i)$

As can be seen, the asynchronism in this model is clearly determined by the pulse generation strategy, and it has, in principle, more synchronization than the event-driven model. In this work, we have proposed and implemented a pulse generation mechanism (functions $getCurrent$ and $hasAdvanced$) for generating pulses using exclusively message exchanges. Our strategy is to advance the computation of a node $i$ (making it to generate a sequence of pulses) so that, for each message received by $i$, $i$ has executed at least as many pulses as the node that sent the message.

# 3. The Parallel Framework

The MPI (Message Passing Interface) library [Dongarra et al. 1995] is a *de facto* standard in HPC applications, being admittedly efficient for implementing parallelism through message exchanges. In this sense, we propose a parallel framework focused on combinatorial optimization applications implemented in C language, as an extension of this library. The implementation and documentation of the MParCO framework and the case study discussed here are available from `https://github.com/UFC-MDCC-HPC/MParCO`.

### 3.1. Event-driven Primitives

These primitives are divided into parameter setting primitives, primitives callable within events, and diffusion and execution primitives. To make use of these primitives, the header file `mparco_ed_mpi.h` should be included.

Parameter setting primitives make it possible to configure parameters for the event-driven model. They are:

**`MPI_Event_set_model`** Informs the framework what are the event and spontaneous event functions to be employed in the next computation.
**`MPI_Event_set_config_param`** Configures model parameters of event-driven computations. The only parameter currently available is `ED_TERMINATION`, which has two possible valuations, `ED_TERMINATION_MODEL`, indicating that the framework will detect termination (default), and `ED_TERMINATION_OFF`, which indicates that the application will detected termination on its own and all nodes will quit their computations when this happens.

Primitives callable within events implement direct communication between the nodes. They are:

**`MPI_Event_neighbors_test`** Tests if a node is neighbor of the current node in the virtual topology.
**`MPI_Event_Send`** Performs a blocking sending of a message during an event execution.
**`MPI_Event_Isend`** Performs a non-blocking (asynchronous) sending of a message during an event execution.
**`MPI_Event_Recv`** Performs a blocking reception of a message during an event execution.

Diffusion and execution primitives allow, respectively, the diffusion of information for all nodes reachable from a node and the start of an application based on events through the framework. In this work, we have designed and implemented an information diffusion mechanism inspired in [Barbosa 1996]. These primitives are:

**`MPI_Event_Bcast`** Performs a blocking mechanism of information diffusion using a width-based strategy.
**`MPI_Event_run`** Runs an application based on the event-driven model.

### 3.2. Pulse-driven Primitives

These primitives are divided into parameter setting primitives, primitives callable within pulses and events, and diffusion and execution primitives. In order to use then, one must include the header file `mparco_pd_mpi.h`.

Parameter setting primitives allow the configuration of parameters for a pulse-driven computation. They are:

**`MPI_Pulse_set_model`** Informs the framework what are the pulse and event functions to be employed in the next computation.
**`MPI_Pulse_set_config_param`** Similar to `MPI_Event_set_config_param`. In addition, it has the parameter `PD_PULSE_GENERATION`, which defines whether the pulse generation mechanism will be controlled by the framework or by the application. In the first case, we have the valuation `PD_PULSE_GENERATION_AUTOMATIC` (default).

In the second, we have the valuation `PD_PULSE_GENERATION_OFF`, and the application will have to control the pulse generation mechanism through the functions `MPI_Pulse_advance_pulse_counter` and `MPI_Pulse_postpone_msg` (described later).

Primitives callable within pulses and events are here similar to `MPI_Event_neighbors_test`, `MPI_Event_Send`, `MPI_Event_Isend`, and `MPI_Event_Recv`. There are also the following primitives, which allow to govern the pulse generation mechanism. They are:

**`MPI_Pulse_advance_pulse_counter`** This primitive forces the generation of a sequence of pulses in the node until the pulse represented by `newpulse` is reached.
**`MPI_Pulse_postpone_msg`** This primitive should only be used within events. It makes the message received in the event to be stored to be processed by a new event, that must occur immediately after the pulse identified by `targetpulse`.

Diffusion and execution primitives are analogous to `MPI_Event_Bcast` and `MPI_Event_run`, respectively.

## 4. Case study: Parallel Branch-and-Bound

Branch-and-bound is a generic method for finding optimal solutions of combinatorial optimization problems [Land and Doig 1960]. In [Karp and Zhang 1993], the authors have proposed a random parallel version of the branch-and-bound algorithm whose idea, in general, consists of employing a set of nodes working in parallel to expand subproblems and randomly distribute the generated subproblems between the other nodes. The randomness in this case aims to avoid a communication overhead in relation to the donor node in the search of a possible idle node to receive subproblems.

In this work, we propose, through the framework introduced here, a random parallel version of the branch-and-bound method, which is inspired in [Karp and Zhang 1993]. Our approach, however, differs from the original one by the fact that in our framework nodes work asynchronously, the graph describing the network is arbitrary and our strategy of random sharing of new subproblems between nodes is different. With regard to the third item, we believe that because our framework is asynchronous, nodes tend to become idle faster and that, instead of receiving subproblems donations inadvertently, they may ask donations to randomly chosen neighbors when they become idle.

The following three types of steps summarize the main ideas of our algorithm, and must be implemented within event and pulse functions, according to the respective model adopted:

- Branching: in a step of this type, a node $i \in N$ unstacks a subproblem $R_i$ from its stack of subproblems $Q_i$ and solves it directly, possibly updating its estimate of lower cost solution, or derives subproblems from it, becoming able to donate subproblems;
- Matching: in such a step, when the stack of subproblems of $i \in N$ is empty ($i$ is idle), it sends a pairing message to request subproblems to a possible donor node. Such a donor node is a neighbor of $i$ chosen at random;
- Donation: consists of a node $i \in N$ donate work to a requesting node. Through it, $i$ donates a set of subproblems of $Q_i$ to the requesting node, and the quantity of subproblems donated is determined by the granularity configured by the application.

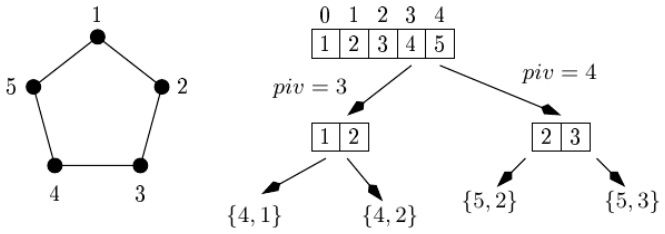



**Figure 1. Recursive sequences of subproblems generation for a graph. In these sequences, four independent sets are enumerated**

### 4.1. Parallel Maximum Independent Set

Consider $G = (V, E)$ an undirected, simple and non-empty graph, in which $V$ is the set of vertices of $G$ and $E$ is the set of edges of $G$, and $|V| = n$ and $|E| = m$. An *independent (or stable) set* $S$ is a subset of $V$ whose elements are not mutually adjacent in $G$. The Maximum Independent Set (MIS) problem consists of finding the largest independent set of $G$. The MIS problem has important applications in many areas such as social network analysis, graphical information systems and coding theory [Liu et al. 2015]. A clique on a graph is a set of vertices fully connected in the graph. Thus, finding the maximum independent set in a graph is equivalent to finding the maximum clique in the complementary graph of the original graph. A standard reference for solving the Maximum Clique problem is the sequential branch-and-bound algorithm proposed in [Tomita and Kameda 2007], called MCR. In this work, we have been inspired on the ideas of the MCR algorithm to propose a parallel strategy for solving the MIS problem, which employs the parallel branch-and-bound algorithm built on top of the MParCO framework (previous section).

In this case study, a subproblem is represented by a list of *candidate vertices*, so called by defining a subgraph of $G$ for which it is desired to find a maximum independent set. Naturally, the initial problem is represented by the list containing all the vertices of the graph. Take a subproblem defined by a list of vertices $R$, of size $r$. The generation of the $i$-th subproblem from $R$ is given by the selection of vertex $R[i]$, so-called *pivot*, and the creation of a new list of vertices (representing a new subproblem), formed by all vertices $R[j]$ such that $0 \leq j < i$ and $R[i]R[j] \notin E$ (there is no edge between $i$ and $j$). Note that the enumeration process resulting from the above operation possibly generates "empty" subproblems, either because $i = 0$ or because there are no candidate vertices in $R$ not adjacent to $R[i]$. As indicated in Figure 1, the occurrence of this fact identifies an independent set of $G$ formed by the pivots of the sequence of recursive generations of subproblems from the original problem. In the figure, $piv$ refers to the index of the current pivot. In the algorithm, pivots are stored in a stack.

Each subproblem $R$ generated during the enumeration is associated with an estimate, in the form of upper limit for the size of the independent sets which can be enumerated from $R$. The estimate employed is derived from a clique cover of the subgraph induced by $R$. A clique cover $\mathcal{C}$ is a set (or family) of cliques that cover all vertices of the (sub)graph. A direct observation is that every independent set has, at most, one vertex of each clique of a clique cover, because if there are two vertices of an independent set belonging to the same clique, there would be an edge between them, which would violate

the condition of independent set. Thus, if an independent set of size at least $|\mathcal{C}|$ is enumerated before $R$, this subproblem can be discarded from the enumeration. A click cover for the root subproblem shown in Figure 1 is the set formed by the cliques $\{1, 2\}$, $\{3, 4\}$ and $\{5\}$, indicating that it is not possible to have an independent set in the subgraph associated to that subproblem whose size is greater than 3. When the root subproblem is branched with $piv = 3$, a subproblem defined by vertices 1, 2, 3 and 4 is obtained, which allows the derivation of a clique cover consisting of $\{1, 2\}$ and $\{3, 4\}$, making it impossible to obtain an independent set with size greater than 2.

In the algorithms described in this section, the generation of a subproblem $R$ is followed by the calculation of a clique cover $\mathcal{C}$ of the subgraph associated to $R$. In this clique cover, cliques are numbered from 1 to $|\mathcal{C}|$. In [Tomita and Kameda 2007], it is shown how to calculate a clique cover of a graph. The vertices of $R$ are then sorted in ascending order according to the clique to which they belong.

Such algorithms basically perform in parallel the following recursive sequential procedure, receiving as input a list of candidate vertices $R$ and a pivot $piv$. In this procedure, variables $R$ corresponds to current list of vertices (subproblem), $Q$ corresponds to an independent set found before $R$ and $Qmax$ consists of the size of the largest independent set found so far.

At the first call of the recursive procedure, $R$ is a list formed by the vertices of $G$, that is, $R = V$. Initially, $piv = |R| - 1$. $\mathcal{C}$ is a clique cover calculated for $R[0, ..., piv]$, which corresponds to the sublist of $R$ starting in $R[0]$ and ending in $R[piv]$. We keep in $Q$ the independent set formed by the pivots of the lists in the path between the root subproblem and $R[0, ..., piv]$. Note that $|\mathcal{C}| + |Q|$ corresponds to the size of the largest independent set that can be found between the root and a leaf, passing through $R[0, ..., piv]$. So, if $|C| + |Q| \leq Qmax$, then the independent set that goes from the root to a leaf, passing through $R[0, ..., piv]$, can not improve the estimate $Qmax$. When this occurs, we discard $R[0, ..., piv]$ and update $Qmax$ to $|Q| + 1$ if $|Q| + 1 > Qmax$. When $|C| + |Q| > Qmax$, we create a new list $R'$ from $R[piv]$. If $R' \neq \emptyset$, we begin to analyze the list $R'$ and we keep $R[0, ..., piv - 1]$ for later. The algorithm ends when $R = \emptyset$. At the end of the run, the size of the maximum independent set of $G$ is $Qmax$.

### 4.2. Event-driven Algorithm

**Algorithm 3** presents a simplified event-driven approach to the MIS problem using the parallel branch-and-bound described in Section 4. A brief description of its main steps are in order. If the node receives a pairing request and can donate, it donates the right half of the base list of its stack (lines 22-34). If the node has no problems to donate, it send a donation denial message (line 36). Upon receipt of a donation message, the donated subproblem is incorporated (lines 38-39). If the current independent set (corresponding to a set formed by the pivots of the path between the base and the top of $Q_i$) can improve the estimate, the node calculates a new list $R'$ and stores it (lines 51-56). If not, it discards the current list, possibly updating the size of the largest independent set found so far, $Qmax_i$ (58-60). When the stack of $i$ is empty and there is no pending donation, $i$ sends a pairing request to a randomly chosen node (lines 43-46). The algorithm ends when the stacks of all nodes become empty. When this occurs, the maximum independent set size is the largest $Qmax_i$ between all nodes. The pulse-driven algorithm follows similar ideas and will be ommited for space limitations.

**Algorithm 3**: Branch-and-bound algorithm for the MIS problem through the event-driven model performed by node $i$

Input: $G = (V, E)$ (when $i = 0$)
// Initialization
1. **include** <mparco_ed_mpi.h>
2. **if** $i = 0$ **then**
3. $R_i \leftarrow V$
4. $beg_i \leftarrow 0$
5. $piv_i \leftarrow |R_i| - 1$
6. Calculate a clique cover $\mathcal{C}_i$ for $R_i[beg_i, ..., piv_i]$
7. Sort $R_i[beg_i, ..., piv_i]$
8. Stack $\langle R_i, beg_i, piv_i, \mathcal{C}_i \rangle$ in $Q_i$
9. **else**
10. $Q_i \leftarrow \emptyset$
11. $Qmax_i \leftarrow 0$
12. $has\_pending\_donation_i \leftarrow$ **false**
13. $can\_donate_i \leftarrow$ **false**
// Configuration
14. MPI_Event_set_config_param(ED_TERMINATION, ED_TERMINATION_MODEL)
15. MPI_Event_set_model(event$_i$, spontaneous_event$_i$)
// Execution
16. MPI_Event_run(...)

**Function 3.1**: spontaneous_event$_i$(...)
17. **include** code snippet branch$_i$
18. **return** 1 //node has workload

**Function 3.2**: event$_i$(...)
19. MPI_Event_Recv(msg$_i$, ...)
20. **switch** msg$_i$
21. **case** "$j$ wants to pair with $i$"
22. **if** $Q_i \neq \emptyset$ **and** $can\_donate_i$ **then**
23. Let $\langle R_i, beg_i, piv_i, \mathcal{C}_i \rangle$ be the base of stack $Q_i$
24. $beg'_i \leftarrow \left\lfloor \frac{piv_i}{2} \right\rfloor$
25. $piv'_i \leftarrow piv_i$
26. $R'_i \leftarrow R_i[beg'_i, ..., piv'_i]$
27. Calculate a clique cover $\mathcal{C}'_i$ for $R'_i[beg'_i, ..., piv'_i]$
28. Sort $R'_i[beg'_i, ..., piv'_i]$
29. Let $D_i$ be the tuple $\langle R'_i, beg'_i, piv'_i, \mathcal{C}'_i \rangle$
30. $piv_i \leftarrow \left\lfloor \frac{piv_i}{2} \right\rfloor - 1$
31. Recalculate a clique cover $\mathcal{C}_i$ for $R_i[beg_i, ..., piv_i]$
32. Sort $R_i[beg_i, ..., piv_i]$
33. $can\_donate_i \leftarrow$ **false**
34. MPI_Event_ISend("$i$ donates $D_i$ to $j$", $j$, ...)
35. **else**
36. MPI_Event_ISend("$i$ can not donate subproblems to $j$", $j$, ...)
37. **case** "$j$ donates $D_j$ to $i$"
38. Stack $D_j$ in $Q_i$
39. $has\_pending\_donation_i \leftarrow$ **false**
40. **case** "$j$ can not donate subproblems to $i$"
41. $has\_pending\_donation_i \leftarrow$ **false**
42. **include** code snippet branch$_i$
43. **if** $\neg has\_pending\_donation_i$ **then**
44. $has\_pending\_donation_i \leftarrow$ **true**
45. Let $j \in N(i)$ a node chosen at random
46. MPI_Event_ISend("$i$ wants to pair with $j$", $j$, ...)
47. **return** 1 //node has workload
48. **return** 0 //node is idle

**Code Snippet 3.1**: branch$_i$

49. **if** $Q_i \neq \emptyset$ **then**
50. Unstack $\langle R_i, beg_i, piv_i, \mathcal{C}_i \rangle$ from $Q_i$
51. **if** $|\mathcal{C}_i| + |Q_i| > Qmax_i$ **then**
52. Calculate a new list $R'_i$ from pivot $R_i[piv_i]$
53. **if** $R'_i \neq \emptyset$ **then**
54. Calculate a clique cover $\mathcal{C}'_i$ for $R'_i[beg_i, ..., piv_i - 1]$
55. Sort $R'_i[beg_i, ..., piv_i - 1]$
56. Stack $\langle R'_i, beg_i, piv_i - 1, \mathcal{C}_i \rangle$ in $Q_i$
57. $can\_donate_i \leftarrow$ **true**
58. **else**
59. **if** $|Q_i| + 1 > Qmax_i$ **then**
60. $Qmax_i \leftarrow |Q_i| + 1$
61. **return** 0 //node is idle

## 4.3. Experiments

In order to evaluate the performance of our framework and parallel versions of the MIS problem, we have implemented a sequential version to solve this problem based on [Tomita and Kameda 2007]. The experiments were carried out in a CentOS Linux computing cluster consisting of 16 processing nodes, each one having 2 processing cores and connected through a gigabit fast-ethernet network. The processing nodes are Intel Xeon processors clocked at 1.8GHz, each RAM memory has capacity of 2GB and the chosen MPI implementation was MPICH[1]. To perform the tests, we have used complement graphs of the 10 most difficult graphs (in terms of finding their maximum cliques) reported in [Tomita and Kameda 2007], which belong to the DIMACS package [Johnson and Trick 1996]. We have also chosen 8 dense random graphs (their complements are, of course, sparse). In the sequential version, we have measured the execution time and the number of branchings performed by the algorithm. In the parallel versions, in addition to these values, we have also calculated the number of donation requests made by all nodes (which corresponds to the number of matching steps) and the number of donations denied by all nodes.

Figure 2 shows the times calculated for the MIS problem through the MParCO framework. We have employed an experimental limit of 1800s for the execution of all experiments. Thus, for cases where this threshold is reached, the number of branchings and requests and denials of donations can serve as a basis for assessing the scenario. Below the name of each graph is shown its number of nodes ($n$), the size of its maximum independent set ($\alpha(G)$) and its original density ($D(G)$), given by $\frac{m}{(\frac{n \times (n-1)}{2})}$.

[1] http://www.mpich.org/

| Graph (n,α(G),D(G)) | Time(s) | Branch. (÷10$^5$) | Nodes | Time(s) | Speed. | Efic. | Branch. (÷10$^5$) | Don. Req. | Den. Don. (%) | Time(s) | Speed. | Efic. | Branch. (÷10$^5$) | Don. Req. | Den. Don. (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **p_hat300-3** | | | | | | | | | | | | | | | |
| (300, 36, 0.744) | 30,65 | 20,69 | 4 | 9,03 | 3,39 | 0,85 | 21,1 | 189,5 | 10 (5,28) | 10,95 | 2,80 | 0,70 | 21,01 | 193,3 | 11,8 (6,1) |
| | | | 8 | 5,84 | 5,25 | 0,66 | 20,41 | 517,6 | 36,6 (7,07) | 8,33 | 3,68 | 0,46 | 20,91 | 280 | 27,4 (9,79) |
| | | | 16 | 4,58 | 6,69 | 0,42 | 20,64 | 478 | 59,2 (12,38) | 6,96 | 4,40 | 0,28 | 21,55 | 249,8 | 53,2 (21,3) |
| **p_hat500-3** | | | | | | | | | | | | | | | |
| (500, 50, 0.75) | 1800,07 | 667,59 | 4 | 1472,18 | 1,22 | 0,31 | 1902 | 425 | 11 (2,59) | 1357,16 | 1,33 | 0,33 | 1908,94 | 453 | 11,33 (2,5) |
| | | | 8 | 755,73 | 2,38 | 0,30 | 1956,41 | 147 | 13 (8,84) | 691,91 | 2,60 | 0,33 | 1992,47 | 1249,67 | 29 (2,32) |
| | | | 16 | 382,05 | 4,71 | 0,29 | 1910,22 | 2821,5 | 61 (2,16) | 361,98 | 4,97 | 0,31 | 1908,15 | 2558,33 | 54,67 (2,14) |
| **brock400_1** | | | | | | | | | | | | | | | |
| (400, 27, 0.75) | 1800,02 | 1876,52 | 4 | 957,96 | 1,88 | 0,47 | 3275,69 | 346,67 | 11 (3,17) | 957,96 | 1,88 | 0,47 | 3378,16 | 346,67 | 11 (3,17) |
| | | | 8 | 461,03 | 3,90 | 0,49 | 3040,49 | 859 | 24,67 (2,87) | 461,03 | 3,90 | 0,49 | 3039,51 | 859 | 24,67 (2,87) |
| | | | 16 | 263,77 | 6,82 | 0,43 | 3425,13 | 1630 | 67,67 (4,15) | 263,77 | 6,82 | 0,43 | 3630,9 | 1630 | 67,67 (4,15) |
| **brock400_2** | | | | | | | | | | | | | | | |
| (400, 29, 0.75) | 1540,94 | 1198,89 | 4 | 375,16 | 4,11 | 1,03 | 969,36 | 301,1 | 11,2 (3,72) | 387,1 | 3,98 | 1,00 | 967,98 | 344,3 | 14,1 (4,1) |
| | | | 8 | 206,91 | 7,45 | 0,93 | 1110,67 | 731 | 28 (3,83) | 193,78 | 7,95 | 0,99 | 1117,44 | 730,2 | 32,8 (4,49) |
| | | | 16 | 140,75 | 10,95 | 0,68 | 1625,32 | 1398,8 | 66,8 (4,78) | 132,03 | 11,67 | 0,73 | 1665,37 | 1425,8 | 63,6 (4,46) |
| **brock400_4** | | | | | | | | | | | | | | | |
| (400, 33, 0.75) | 1298,28 | 1159,99 | 4 | 335,45 | 3,87 | 0,97 | 978,22 | 271,9 | 11,5 (4,23) | 266,34 | 4,87 | 1,22 | 1014,26 | 301 | 13,7 (4,55) |
| | | | 8 | 151,82 | 8,55 | 1,07 | 1055,2 | 603 | 26,2 (4,34) | 129,16 | 10,05 | 1,26 | 1202,62 | 536,8 | 31,4 (5,85) |
| | | | 16 | 24,47 | 53,06 | 3,32 | 189,87 | 1082,8 | 59 (5,45) | 24,86 | 52,22 | 3,26 | 229,94 | 1025 | 63,7 (6,21) |
| **DSJC500.5** | | | | | | | | | | | | | | | |
| (500, 13, 0.502) | 7,84 | 13,12 | 4 | 6,16 | 1,27 | 0,32 | 12,82 | 176,2 | 10,3 (5,85) | 6,58 | 1,19 | 0,30 | 13,13 | 221,1 | 17,6 (7,96) |
| | | | 8 | 2,73 | 2,87 | 0,36 | 12,91 | 213 | 28,4 (13,33) | 3,67 | 2,14 | 0,27 | 12,92 | 142,2 | 28,2 (19,83) |
| | | | 16 | 2,29 | 3,42 | 0,21 | 12,39 | 127,6 | 48,4 (37,93) | 2,4 | 3,27 | 0,20 | 12,5 | 107,6 | 44,4 (41,26) |
| **DSJC1000.5** | | | | | | | | | | | | | | | |
| (1000, 15, 0.50) | 706,39 | 911,61 | 4 | 230,26 | 3,07 | 0,77 | 907,12 | 166,33 | 11,33 (6,81) | 204,84 | 3,45 | 0,86 | 907,14 | 166,33 | 6 (3,61) |
| | | | 8 | 102,98 | 6,86 | 0,86 | 895,65 | 1903,67 | 30 (1,58) | 102,98 | 6,86 | 0,86 | 896,66 | 1903,67 | 41,67 (2,19) |
| | | | 16 | 59,49 | 11,87 | 0,74 | 873,94 | 2075 | 101,67 (4,9) | 64,78 | 10,90 | 0,68 | 884,87 | 2075 | 69 (3,33) |
| **MANN_a27** | | | | | | | | | | | | | | | |
| (378, 126, 0.9901) | 8,17 | 0,38 | 4 | 2,13 | 3,84 | 0,96 | 0,38 | 22 | 9 (40,91) | 2,07 | 3,95 | 0,99 | 0,38 | 26,8 | 9,5 (35,45) |
| | | | 8 | 1,12 | 7,29 | 0,91 | 0,39 | 49,8 | 19,8 (39,76) | 1,05 | 7,78 | 0,97 | 0,39 | 67,8 | 22,8 (33,63) |
| | | | 16 | 0,71 | 11,51 | 0,72 | 0,39 | 102,6 | 40,2 (39,18) | 0,56 | 14,59 | 0,91 | 0,4 | 130,6 | 44,4 (34) |
| **MANN_a45** | | | | | | | | | | | | | | | |
| (1035, 345, 0.996) | 1802,77 | 10,57 | 4 | 1328,85 | 1,36 | 0,34 | 29,63 | 49,8 | 11,1 (22,29) | 1800,24 | 1,00 | 0,25 | 35,62 | 51,5 | 10,4 (20,19) |
| | | | 8 | 660,56 | 2,73 | 0,34 | 29,52 | 138,6 | 24,4 (17,6) | 1277,86 | 1,41 | 0,18 | 29,52 | 127,2 | 25,2 (19,81) |
| | | | 16 | 341,56 | 5,28 | 0,33 | 29,56 | 367 | 45,5 (12,4) | 633,92 | 2,84 | 0,18 | 29,56 | 186,8 | 52,8 (28,27) |
| **san400_0.9_1** | | | | | | | | | | | | | | | |
| (400, 100, 0.9) | 20,22 | 2,62 | 4 | 0,21 | 96,29 | 24,07 | 0,06 | 10,9 | 8,6 (78,9) | 1,46 | 13,85 | 3,46 | 0,05 | 19,9 | 11,6 (58,29) |
| | | | 8 | 0,18 | 112,33 | 14,04 | 0,12 | 19,4 | 16,6 (85,57) | 0,1 | 202,20 | 25,28 | 0,18 | 29,2 | 18,6 (63,7) |
| | | | 16 | 0,16 | 126,38 | 7,90 | 0,36 | 42 | 34,6 (82,38) | 0,35 | 57,77 | 3,61 | 1,03 | 54,8 | 37,2 (67,88) |
| **g200_90** | | | | | | | | | | | | | | | |
| (200, 40, 0.90) | 1504,28 | 984,92 | 4 | 413,99 | 3,63 | 0,91 | 952,42 | 104,33 | 10,67 (10,22) | 393,16 | 3,83 | 0,96 | 954,38 | 111 | 11,33 (10,21) |
| | | | 8 | 169,01 | 8,90 | 1,11 | 731,53 | 249 | 23 (9,24) | 156,45 | 9,62 | 1,20 | 738,44 | 326,33 | 24 (7,35) |
| | | | 16 | 88,54 | 16,99 | 1,06 | 769,28 | 632 | 52,33 (8,28) | 553,42 | 2,72 | 0,17 | 784,38 | 421,67 | 56,33 (13,36) |
| **g300_70** | | | | | | | | | | | | | | | |
| (300, 20, 0.70) | 45,67 | 58,85 | 4 | 14,43 | 3,16 | 0,79 | 59,46 | 195,67 | 9 (4,6) | 13,05 | 3,50 | 0,87 | 60,6 | 247,33 | 12 (4,85) |
| | | | 8 | 7,6 | 6,01 | 0,75 | 59,95 | 370,5 | 34 (9,18) | 7,31 | 6,25 | 0,78 | 59,83 | 327 | 25,33 (7,75) |
| | | | 16 | 4,12 | 11,08 | 0,69 | 57,42 | 863,33 | 64,33 (7,45) | 4,12 | 11,08 | 0,69 | 58,48 | 863,33 | 64,33 (7,45) |
| **g300_90** | | | | | | | | | | | | | | | |
| (300, 44, 0.9) | 1800,02 | 1147,72 | 4 | 1800,05 | 1,00 | 0,25 | 3920,47 | 27,4 | 11,3 (41,24) | 1800,76 | 1,00 | 0,25 | 4161,37 | 37,8 | 12,6 (33,33) |
| | | | 8 | 1800,03 | 1,00 | 0,12 | 7288,65 | 62 | 23,8 (38,39) | 1800,68 | 1,00 | 0,12 | 7902,27 | 69,6 | 24,4 (35,06) |
| | | | 16 | 1800,08 | 1,00 | 0,06 | 10974,86 | 113,2 | 47 (41,52) | 1800,72 | 1,00 | 0,06 | 15520,03 | 88,4 | 43,6 (49,32) |
| **g300_95** | | | | | | | | | | | | | | | |
| (300, 66, 0.95) | 1800,01 | 739,76 | 4 | 1800,06 | 1,00 | 0,25 | 2350,18 | 54,7 | 14,5 (26,51) | 1800,47 | 1,00 | 0,25 | 2538,23 | 80,7 | 15,5 (19,21) |
| | | | 8 | 1800,09 | 1,00 | 0,12 | 4705,24 | 78,8 | 26 (32,99) | 1800,47 | 1,00 | 0,12 | 4958,74 | 82,4 | 27,2 (33,01) |
| | | | 16 | 1800,11 | 1,00 | 0,06 | 6989 | 137,4 | 48,8 (35,52) | 1800,52 | 1,00 | 0,06 | 9903,29 | 131,4 | 44,2 (33,64) |
| **g400_70** | | | | | | | | | | | | | | | |
| (400, 21, 0.70) | 649,94 | 754,65 | 4 | 200,99 | 3,23 | 0,81 | 749,95 | 44,5 | 11,2 (25,17) | 180,74 | 3,60 | 0,90 | 760,27 | 290,33 | 10,33 (3,56) |
| | | | 8 | 100,43 | 6,47 | 0,81 | 761,67 | 79 | 25 (31,65) | 91,82 | 7,08 | 0,88 | 763,2 | 733 | 23,33 (3,18) |
| | | | 16 | 53,33 | 12,19 | 0,76 | 784,34 | 96 | 40,5 (42,19) | 48,98 | 13,27 | 0,83 | 796,14 | 1533,67 | 66 (4,3) |
| **g400_90** | | | | | | | | | | | | | | | |
| (400, 47, 0.9) | 1800 | 1149,71 | 4 | 1800,02 | 1,00 | 0,25 | 3862,14 | 42,1 | 11,4 (27,08) | 1800,8 | 1,00 | 0,25 | 4225,17 | 35,6 | 11,3 (31,74) |
| | | | 8 | 1800,04 | 1,00 | 0,12 | 7172,51 | 52,2 | 20,2 (38,7) | 1800,69 | 1,00 | 0,12 | 7755,02 | 72 | 22,4 (31,11) |
| | | | 16 | 1800,07 | 1,00 | 0,06 | 13637,75 | 117 | 48 (41,03) | 1800,68 | 1,00 | 0,06 | 14977,9 | 137 | 49,4 (36,06) |
| **g400_95** | | | | | | | | | | | | | | | |
| (400, 71, 0.95) | 1800,16 | 833,64 | 4 | 1800,02 | 1,00 | 0,25 | 2296,16 | 24,9 | 10,7 (42,97) | 1800,51 | 1,00 | 0,25 | 2581,24 | 39 | 10,9 (27,95) |
| | | | 8 | 1800,08 | 1,00 | 0,13 | 4793,47 | 50 | 21 (42) | 1800,51 | 1,00 | 0,12 | 5274,85 | 74 | 25,6 (34,59) |
| | | | 16 | 1800,06 | 1,00 | 0,06 | 9156,32 | 108,5 | 47 (43,32) | 1800,3 | 1,00 | 0,06 | 1802,55 | 53,2 | 36,4 (68,42) |
| **g1000_50** | | | | | | | | | | | | | | | |
| (1000, 15, 0.5) | 662,96 | 852,2 | 4 | 197,51 | 3,36 | 0,84 | 847,85 | 1501,4 | 24,2 (1,61) | 194,51 | 3,41 | 0,85 | 768,95 | 733,8 | 16,2 (2,21) |
| | | | 8 | 106,07 | 6,25 | 0,78 | 857,36 | 2053,8 | 33,8 (1,65) | 105,2 | 6,30 | 0,79 | 859,56 | 1348,2 | 33,2 (2,46) |
| | | | 16 | 73,51 | 9,02 | 0,56 | 864,97 | 3106 | 59 (1,9) | 82,41 | 8,04 | 0,50 | 871,33 | 2322 | 62,4 (2,69) |

- DIMACS Graph
- Random Graph
- Sequential Computation
- Event-driven Computation
- Pulse-driven Computation

**Figure 2. Calculated times for the Maximum Independent Set Problem through the MParCO Framework**

An initial observation is that in hard graphs whose sequential runs have been completed within the time limit of 1800s (e.g. brock400_2, brock400_4 and g200_90), we have a small percentage of denials of donations. This reveals that a large number of branchings tends to reduce the effects of unbalancing. For easy graphs (p_hat300-3, DSJC500.5, g300_70, etc.), if we look at the number of denials of donations, we generally have a greater unbalancing. However, for these executions, this unbalancing does not significantly interfere in performance, since the gain with parallelization of the enumeration is greater, which in some cases results in a superlinear speedup [Grama et al. 2003]. According to [Grama et al. 2003], superlinear speedup is a common anomaly to parallelizations of search algorithms such as branch-and-bound. Another possible analysis concerns to very hard graphs, which have not been solved by sequential or parallel implementations in up to 1800s (e.g. g300_90, g300_95, g400_90 g400_95). Note that for these experiments, up to the times of their interruptions, we have a large number of branchings and negations of donations, what means that the enumeration process begins slowly and accelerates gradually, following the growth of the enumeration tree. In this way, we conclude that for these executions a good balancing is maintained.

Finally, we can establish a comparison between the two distributed models with respect to the application in question. For very dense graphs (density between 0.9000 and 0.9999), the version in the pulse-driven model has revealed better performance. Because of this high density, there are a large number of candidate vertex lists, which favors the pulse-driven model since the pulses are generated as long as there are subproblems to be expanded, preventing the nodes from idling and thus emptying their stacks faster. However, for the other graphs with lower density, the event-driven version has been more efficient. Once there are fewer lists of candidate vertices, nodes become idle faster. This idleness causes nodes to perform more donation requests. Donation requests and their responses (donation or denial of donation) generate events on nodes, causing nodes to work (when they receive donations) or keep asking (when they receive donation denials).

## 5. Related Works

In the last decades, the search for efficient parallelization strategies for large-scale search problems such as branch-and-bound is notorious. This tremendous attention has motivated the development of several parallel frameworks for Combinatorial Optimization, classified according to the following major criteria: (1) the node search algorithm involved in the search process, (2) the metaheuristic employed to find out near-optimal solutions and (3) the programming environment they use to implement the parallelization. Most of the available parallel frameworks are specialized in a combination of items 1 and 3 or 2 and 3. In the first case, for example, BCP [Saltzman 2002] is an implementation of the branch-and-price-and-cut algorithm, which runs on the MPI programming environment, and Bob++ and its improvements [Djerrah et al. 2006, Menouer 2018] provide a variety of exact combinatorial optimization search methods and an interface to facilitate parallel implementation. In the second case, one can cite a wide variety of parallel implementations of intricate metaheuristics [Cho et al. 2015, Guzman et al. 2016]. The approach proposed in this article, however, is innovative in the sense that it does not employ in principle any problem-solving strategy, focusing exclusively on ensuring that the necessary asynchronism is easily achieved. Moreover, it differs from others by providing a minimalist programming interface, reproducible only from the MPI library and some

C files, what can configure an interesting alternative for users who desire to implement themselves their parallelization strategies for combinatorial optimization problems.

## 6. Conclusions and Future Work

In the previous sections, we have made a case for the introduction of a minimalist parallel framework for implementing combinatorial optimization applications, so-called MParCO. This has a *dual* character. On the one hand, the expected performance in parallelizations of this kind of applications must be compatible with *ad hoc* parallelization strategies already established in the state-of-the-art for the problems analyzed. On the other hand, the implementation of such strategies is usually complex, making architectural principles that aim to increase the power of abstraction be welcome.

Regarding the performance issue, in order to accurately measure the overhead incurred by the framework, it is necessary to evaluate in the case study graphs that have small unbalancing (donation negation percentage of at most 15%), since in this case there is a lower synchronization load, have sequential time lower than 1800s and have no superlinear speedup (efficiency at most 1). For instances in which this occurs (brock400_2, DSJC1000.5, g300_70, g400_70 and g1000_50), an average efficiency of 90% was observed. This reveals that the overhead incurred by the framework was close to 10%. Although further experimental evaluation is necessary to precisely establish the framework performance curve, our preliminary results are encouraging.

From a broader perspective, the adoption, at design time of a combinatorial optimization application, of a mechanism that, although not offering performance equivalent to a genuine parallelization strategy for the problem in question, simply abstracts some of the main concerns necessary to achieve a good parallelization, such as asynchronism, has proved to be worth exploring. In particular, it is part of our current work to investigate other sources/models of asynchronism and native load balancing support to be incorporated into the framework, and the proposed approach through HPC Shelf [de Carvalho Silva et al. 2018, de Oliveira Dantas et al. 2017], a cloud computing platform for HPC services, which makes it possible to deal more efficiently with the complexity of overlapping different hierarchies of parallelism.

## References

Bader, D. A., Hart, W. E., and Phillips, C. A. (2005). *Parallel Algorithm Design for Branch and Bound*, pages 5–1–5–44. Springer New York, New York, NY.

Barbosa, V. C. (1996). *An Introduction to Distributed Algorithms*. A The MIT Press, Cambridge, MA.

Cho, I., Park, S., Han, D., and Shin, J. (2015). Practical Message-passing Framework for Large-scale Combinatorial Optimization. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 24–31.

Corrêa, R. C. (2002). *Models for Parallel and Distributed Computation: Theory, Algorithmic Techniques, and Applications*. Kluwer Academic Publishers, Norwell, MA, USA.

Corrêa, R. C. and Barbosa, V. C. (2009). Partially Ordered Distributed Computations on Asynchronous Point-to-point Networks. *Parallel Computing*, 35(1):12–28.

Crainic, T. G., Le Cun, B., and Roucairol, C. (2006). *Parallel Branch-and-Bound Algorithms*, pages 1–28. John Wiley & Sons, Inc.

de Carvalho Silva, J., de Oliveira Dantas, A. B., and de Carvalho Junior, F. H. C. (2018). A Scientific Workflow Management System for Orchestration of Parallel Components in a Cloud of Large-Scale Parallel Processing Services. *Science of Computer Programming*.

de Oliveira Dantas, A. B., de Carvalho Junior, F. H., and Soares Barbosa, L. (2017). A Framework for Certification of Large-scale Component-based Parallel Computing Systems in a Cloud Computing Platform for HPC Services. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER*, pages 229–240. ScitePress.

Djerrah, A., Le Cun, B., Cung, V. D., and Roucairol, C. (2006). Bob++: Framework for Solving Optimization Problems with Branch-and-Bound methods. In *2006 15th IEEE International Conference on High Performance Distributed Computing*, pages 369–370.

Dongarra, J., Otto, S. W., Snir, M., and Walker, D. (1995). An Introduction to the MPI Standard. Technical Report CS-95-274, University of Tennessee.

Grama, A., Karypis, G., Kumar, V., and Gupta, A. (2003). *Introduction to Parallel Computing*. Addison-Wesley, second edition.

Guzman, L. G., Ruiz, E. D. N., Ardila, C. J., Jabba, D., and Nieto, W. (2016). A Novel Framework for the Parallel Solution of Combinatorial Problems implementing Tabu Search and Simulated Annealing Algorithms. In *2016 6th International Conference on Computers Communications and Control (ICCCC)*, pages 259–263.

Johnson, D. S. and Trick, M. A., editors (1996). *Second DIMACS implementation challenge : cliques, coloring and satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society.

Karp, R. M. and Zhang, Y. (1993). Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation. *Journal of the ACM*, 40(3):765–798.

Land, A. H. and Doig, A. G. (1960). An Automatic Method for Solving Discrete Programming Problems. *ECONOMETRICA*, 28(3):497–520.

Liu, Y., Lu, J., Yang, H., Xiao, X., and Wei, Z. (2015). Towards Maximum Independent Sets on Massive Graphs. *Proc. VLDB Endow.*, 8(13):2122–2133.

Menouer, T. (2018). Solving Combinatorial Problems Using a Parallel Framework. *Journal of Parallel and Distributed Computing*, 112(P2):140–153.

Misra, J. and Chandy, K. M. (1982). Termination Detection of Diffusing Computations in Communicating Sequential Processes. *ACM Trans. Program. Lang. Syst.*, 4(1):37–43.

Saltzman, M. J. (2002). *Coin-Or: An Open-Source Library for Optimization*, pages 3–32. Springer US, Boston, MA.

Tomita, E. and Kameda, T. (2007). An Efficient Branch-and-Bound Algorithm for Finding a Maximum Clique with Computational Experiments. *Journal of Global Optimization*, 37(1):95–111.

# P-TWDTW: Processamento Paralelo de Análises de Séries Temporais de Imagens de Sensoriamento Remoto utilizando Arquiteturas Manycore

**Sávio S. T. de Oliveira[1], Vagner J. do S. Rodrigues[1], Laerte G. Ferreira[2], Wellington S. Martins[1]**

[1]Instituto de Informática - Universidade Federal de Goiás (UFG)
131 - CEP 74001-970 - Goiânia - GO - Brasil

[2]LAPIG - CAMPUS II Samambaia - Cx. POSTAL 131
CEP: 74001-970 - Goiânia - GO - Brasil

{savioteles,vsacramento,lapig.ufg}@gmail.com, wellington@inf.ufg.br

***Abstract.*** *In the class of computationally complex problems, the time series analysis is one of those that has high demand for computational power, due to the complexity of the algorithms and the large volume of data to be analysed. The TWDTW algorithm stands out as of the best solution found in the literature in this field, but its time complexity $O(n^2)$ makes its unfeasible for large data sets. This work proposes a parallel algorithm, called P-TWDTW (Parallel TWDTW), that allows analyzing large scale time series exploring Manycore (GPU) architectures. In the evaluation of the algorithm, the P-TWDTW proved to be a promising solution with response time up to 11 times lower than TWDTW.*

***Resumo.*** *Dentro da classe de problemas computacionalmente complexos, a análise de séries temporais é uma das que possui maior demanda de poder computacional, devido a complexidade dos algoritmos e o grande volume de dados a serem processados. O algoritmo TWDTW tem se destacado como a melhor solução encontrada na literatura nesta área, mas sua complexidade de tempo $O(n^2)$ torna seu uso inviável para grandes conjuntos de dados. Este trabalho propôs uma solução paralela, denominada P-TWDTW (Parallel TWDTW), capaz de analisar grandes volumes de séries temporais explorando arquiteturas Manycore (GPU). Na avaliação do algoritmo, o P-TWDTW se mostrou uma solução promissora com tempo de resposta até 11 vezes menor do que o TWDTW.*

## 1. Introdução

A superfície do planeta Terra está mudando a uma taxa sem precedentes, onde ecossistemas de florestas diminuem a uma velocidade alarmante e áreas urbanas e agrícolas expandem em volta do espaço natural. Análise das séries temporais de imagens de sensoriamento remoto tem se tornado indispensáveis na identificação destas mudanças. Por isso, tem atraído grande interesse econômico e da área de meio ambiente no cenário mundial, se tornando um importante recurso em diversas aplicações [Kuenzer et al. 2015].

Uma série temporal é uma coleção de observações feitas sequencialmente ao longo do tempo. A análise das séries temporais de imagens de sensoriamento remoto é um dos problemas com maior demanda de poder computacional na área de sensoriamento remoto, devido a complexidade dos algoritmos e o grande volume de dados

[Rakthanmanon et al. 2013]. Na China, por exemplo, existem mais de 100.000 bases de estações meteorológicas que geram aproximadamente 100 bilhões de dados históricos multidimensionais por ano [Huang et al. 2016].

Dentre o grupo de algoritmos de análise de séries temporais, o algoritmo Time-Weighted Dynamic Time Warping (TWDTW) se destaca como a melhor solução encontrada na literatura para buscar todas as possíveis ocorrências de um determinado padrão dentro de uma série temporal de imagens de sensoriamento remoto [Maus et al. 2016]. O TWDTW é uma adaptação do Dynamic Time Warping (DTW) [Sakoe 1971], um método bastante conhecido para análise de séries temporais. O DTW compara um padrão temporal de um evento conhecido com uma série temporal não conhecida. Ele encontra todos os alinhamentos possíveis entre duas séries temporais e fornece uma medida de similaridade.

Ao contrário do DTW, o algoritmo TWDTW é sensível as mudanças sazonais dos diversos tipos de vegetações naturais e cultivadas, o que é extremamente importante na área de sensoriamento remoto [Maus et al. 2016]. O algoritmo TWDTW possui alto custo computacional, com complexidade de tempo $O(n^2)$, o que torna seu uso inviável em grandes conjuntos de dados [Xiao et al. 2013].

O presente trabalho propõe uma nova solução altamente paralela, denominada Parallel TWDTW (P-TWDTW), que tem como objetivo permitir o processamento de grande volume de dados utilizando a arquitetura Manycore (GPU) com o uso coordenado e apropriado da enorme quantidade de núcleos disponíveis. Para a comparação entre cada padrão e série temporal, o TWDTW utiliza um algoritmo de programação dinâmica que não é trivialmente paralelizável. Este trabalho implementou um novo algoritmo paralelo (P-TWDTW) com uma otimização no algoritmo original para que fosse possível explorar os núcleos da arquitetura Manycore. O P-TWDTW possui complexidade de tempo $O(\frac{n^2}{p})$, onde $p$ é o número de núcleos disponíveis.

O P-TWDTW processa as séries temporais como *batchs*, ou seja, comparando vários padrões e diversas séries temporais ao mesmo tempo na GPU. Estes *batchs* são, em alguns casos, maiores que a memória RAM disponível na máquina, o que traz um desafio de processar as análises de forma eficiente. Esse desafio aumenta com o uso da GPU cuja memória é geralmente menor que a memória RAM. Por isso, foi criado um sistema que automaticamente coordena o uso destes espaços de memória, para que seja possível processar grande volume de dados explorando o alto número de núcleos disponíveis nas GPUs. As contribuições deste trabalho estão listadas abaixo:

- Novo algoritmo paralelo para análise de séries temporais de imagens de sensoriamento remoto em arquiteturas Manycore;
- Sistema para coordenação dos espaços de memória RAM e da GPU para não exceder a capacidade das mesmas.

O artigo está organizado como se segue. A Seção 2 discorre a respeito do processamento de análises de séries temporais de imagens de sensoriamento remoto. A Seção 3 descreve o algoritmo TWDTW utilizado como base para este trabalho. O novo algoritmo proposto neste trabalho (P-TWDTW) é apresentado na Seção 4 e a Seção 5 valida o algoritmo P-TWDTW com os experimentos e discussões sobre o desempenho do algoritmo. A Seção 6 apresenta as conclusões e trabalhos futuros.

## 2. Análise de Séries Temporais de Imagens de Sensoriamento Remoto

A análise de séries temporais compreende métodos para analisar dados de séries temporais e extrair estatísticas e características importantes. O DTW é um dos métodos mais conhecidos nesta área e permite realizar o alinhamento entre duas séries temporais e, assim, reconhecer a similaridade entre pares de séries, mesmo se elas apresentarem comprimentos diferentes ou não estiverem alinhadas no eixo do tempo. Dadas as séries temporais $A = a_1, a_2, ..., a_m$ e $B = b_1, b_2, ..., b_n$ com tamanhos $m$ e $n$ respectivamente, o cálculo do DTW é realizado utilizando a Equação 1:

$$DTW(A, B) = min\sqrt{\sum_{k=1}^{K} w_k} \tag{1}$$

em que $w_k = (i, j)$ representa a associação entre as observações $a_i$ e $b_j$ das séries temporais A e B, sendo equivalentes a distância Euclidiana $d(i, j) = \sqrt{(a_i - b_i)^2}$. A sequência $w_1, w_2, ..., w_k$ representa a sequência de associações entre pares de observação das duas séries, denominada o caminho de ajuste. A Equação 1 segue as seguintes condições: i) $w_1 = (1, 1)$ e $w_k = (m, n)$ devem ser iguais; ii) dado $w_k = (i, j)$, então $w_{k+1} = (i', j')$, sendo $i' - i \leq 1$ e $j' - j \leq 1$; iii) dado $w_k = (i, j)$ e $w_{k+1} = (i', j')$, então $i' - i > 0$ e $j' - j \leq 0$.

O algoritmo DTW não é recomendável para análise de séries temporais de imagens de sensoriamento remoto, pois não é capaz de identificar as mudanças sazonais dos diversos tipos de vegetações naturais e cultivadas [Maus et al. 2016]. Por exemplo, o ciclo de plantação de soja varia entre setembro e dezembro, mas o DTW pode encontrar um padrão de plantação de soja em maio, mesmo sendo outro tipo de cultivo, por não ser capaz de avaliar a sazonalidade destas plantações.

Alguns trabalhos [Petitjean et al. 2012, Petitjean and Weber 2014, Maus et al. 2016] propuseram métodos utilizando o DTW para análise de séries temporais de imagens de satélite sem o uso do paralelismo. Alguns métodos [Petitjean et al. 2012, Petitjean and Weber 2014] utilizam um atraso de tempo máximo para evitar distorções temporais baseados na data das imagens de satélite. O método TWDTW [Maus et al. 2016] busca encontrar todas as possíveis ocorrências de um determinado padrão dentro de uma série temporal, introduzindo restrições temporais, e tem se destacado na acurácia de identificação do uso da cobertura do solo. Outros trabalhos [Verbesselt et al. 2010, Jamali et al. 2015] possuem suporte para execução paralela em arquiteturas Multicore utilizando a biblioteca *foreach*[1] do R. Mas, esta solução não explora o potencial de arquiteturas Manycores.

Com a alta capacidade de processamento, as GPUs estão conduzindo uma nova direção no processamento paralelo de imagens de satélite. Em [Christophe et al. 2011] é apresentado um *framework* que permite que algoritmos de processamento de imagens de satélite sejam implementados na GPU, mas não apresenta soluções para a análise de séries temporais. Alguns trabalhos [Xiao et al. 2013, João Jr et al. 2017, Zhu et al. 2018] apresentam soluções paralelas utilizando GPUs para análise de séries temporais, mas não

[1] https://cran.r-project.org/web/packages/foreach/index.html

abordam os problemas detectados por [Maus et al. 2016] na área de sensoriamento remoto.

Para conseguir processar imagens cujo tamanho é maior que a memória disponível na GPU, o trabalho de [Juan and Jianchao 2013] carrega toda a imagem para a memória RAM da máquina e gera diversos blocos da imagem com tamanho menor que a memória da GPU. Em [Liu et al. 2014] são propostas estratégias de otimização utilizando uma lista na memória RAM para armazenar blocos da imagem de entrada para serem processados na GPU. Estes dois trabalhos não propõem soluções para o processamento de séries temporais de imagens de sensoriamento remoto.

## 3. Time-Weighted Dynamic Time Warping (TWDTW)

O TWDTW [Maus et al. 2016] é uma variação do DTW e busca encontrar todas as possíveis ocorrências de um determinado padrão dentro de uma série temporal. Para evitar, por exemplo, que um padrão extraído do inverno não case com uma série temporal extraída do verão, o TWDTW introduz restrições temporais. Se existir uma grande diferença sazonal entre o padrão da amostra e as séries temporais que casam com esse padrão, um custo extra é adicionado ao DTW. Essa restrição controla as distorções ao longo do tempo e torna o alinhamento da série temporal dependente apenas das estações do ano.

O método TWDTW calcula a matriz de custo $\Psi$ com dimensão $n$ por $m$ a partir do padrão $U = (u_1, ..., u_n)$ e da série temporal $V = (v_1, ..., v_m)$. Os elementos $\psi_{i,j}$ da matriz são calculados adicionando um custo temporal $\omega$, tornando-se $\psi_{i,j}$ = $|u_i - v_j| + \omega_{i,j}$, onde $u_i \in U \; \forall \; i = 1, ..., n$ e $v_j \in V \; \forall \; j = 1, ..., m$. Para calcular o custo temporal, o modelo logístico é utilizado com um ponto médio $\beta$ e uma inclinação $\alpha$ da equação 2

$$\omega_{i,j} = \frac{1}{1 + e^{-\alpha(g(t_i,t_j)-\beta)}}, \tag{2}$$

onde $g(t_i, t_j)$ é o tempo decorrido em dias entre as datas $t_i$ no padrão $U$ e $t_j$ na série temporal $V$. Da matriz de custo $\Psi$ é calculada uma matriz de custo acumulado denominada $D$ utilizando uma soma recursiva das distâncias mínimas, tal como na equação 3

$$d_{i,j} = \psi_{i,j} + min\{d_{i-1,j}, d_{i-1,j-1}, d_{i,j-1}\}, \tag{3}$$

que está sujeita aos seguintes condições:

$$d_{ij} = \begin{cases} \psi_{i,j} & i = 1, j = 1 \\ \sum_{k=1}^{i} \psi_{k,j} & 1 < i \leq n, j = 1 \\ \sum_{k=1}^{j} \psi_{i,k} & i = 1, 1 < j \leq m \end{cases} \tag{4}$$

O $kth$ caminho de menor custo em D produz um alinhamento entre o padrão e uma subsequencia de $V$ com a distância associada $\delta_k$, onde $a_k$ é o ponto inicial e $b_k$ é o ponto final da subsequencia $k$. Cada ponto mínimo na última linha da matriz de custo acumulado, i.e. $d_{n,j} \; \forall \; j = 1, ..., m$, produz um alinhamento, com $b_k = argmin_k(d_{n,j}), k = 1, ..., K$ e $\delta_k = d_{n,b_k}$, onde K é o número mínimo de pontos na última linha da matriz $D$.

Um algoritmo reverso, Equação 5, mapeia o caminho $P_k = (p_1, ..., p_L)$ ao longo do $k$th "vale" de menor custo em $D$. O algoritmo inicia em $p_{l=L} = (i = n, j = b_k$

e termina com i = 1, i.e. $p_{l=1} = (i = 1, j = a_k)$, onde $L$ denota o último ponto do alinhamento. O caminho $P_k$ contém os pontos que tiveram casamento entre as séries temporais.

$$p_{l-1} = \begin{cases} (i, a_k = j) & \text{se } i = 1 \\ (i-1, j) & \text{se } j = 1 \\ argmin(d_{i-1,j}, d_{i-1,j-1}, d_{i,j-1}) & \text{caso contrário} \end{cases} \quad (5)$$

O método de classificação com o TWDTW é executado em dois passos. No primeiro passo, o algoritmo DTW é aplicado para cada padrão em $U \in Q$ e cada série temporal $V \in S$. Esse passo fornece informações de quantos padrões casam com intervalos das séries temporais. No segundo passo, o melhor padrão encontrado pelo DTW é utilizado para construir mapas com as séries temporais de uso e cobertura de solo. Portanto, ele compara todos os padrões $U$ com todas as séries temporais $V$.

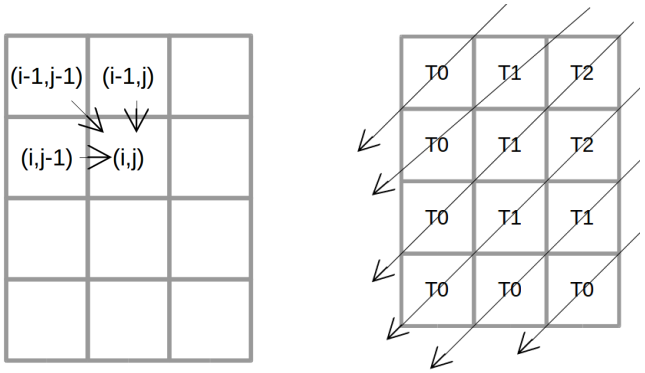
## 4. Processamento Paralelo de Análises de Séries Temporais de Imagens de Sensoriamento Remoto utilizando GPUs

O TWDTW é implementado por um algoritmo de programação dinâmica que possui complexidade de tempo $O(n^2)$, o que o torna inviável para um grande volume de dados. Esta seção apresenta a descrição da solução proposta neste trabalho para processamento paralelo de análises de séries temporais de imagens de sensoriamento remoto utilizando arquiteturas Manycore (GPU). Esta solução, denominada P-TWDTW (Parallel TWDTW), possui complexidade de tempo $O(\frac{n^2}{p})$, onde $p$ é o número de núcleos disponíveis.

A matriz de custo acumulado, denominada $D$, é calculada a partir da matriz de custo $\Psi$ utilizando a soma recursiva das distâncias mínimas, tal como apresentado na equação 3. A construção da matriz $D$ não pode ser paralelizada de forma trivial, já que o cálculo de cada elemento $(i, j)$ da matriz depende dos elementos $(i-1, j)$, $(i, j-1)$ e $(i-1, j-1)$ previamente calculados. Esta dependência pode ser vista na Figura 1(a). A ideia do algoritmo P-TWDTW para calcular a matriz $D$ de forma paralela é apresentada na Figura 1(b). Cada diagonal é calcula de forma paralela, onde cada *thread* fica responsável por uma célula da diagonal. Como os elementos dentro da diagonal não são dependentes entre si, o cálculo do custo acumulado pode ser realizado sem o risco de gerar inconsistências. Os detalhes do cálculo da matriz $D$ do P-TWDTW são apresentados no Algoritmo 1 na Seção 4.1.

### 4.1. Parallel Time-Weighted Dynamic Time Warping (P-TWDTW)

O Algoritmo 1 descreve o P-TWDTW, que recebe como entrada o conjunto de padrões $Q$ e o conjunto de séries temporais $S$ e calcula a matriz de custo acumulado entre cada padrão $U \in Q$ e cada série temporal $V \in S$. Como os conjuntos $Q$ e $S$ podem ser maiores que a memória RAM, a entrada deste algoritmo admite que estes conjuntos estão armazenados no disco. O P-TWDTW fica responsável por coordenar o carregamento de blocos de $Q$ e $S$ para a memória RAM e, posteriormente, para a memória da GPU, de forma que não exceda os limites das mesmas. O P-TWDTW também recebe como entrada o número que representa o tamanho máximo dos conjuntos $Q$ e $S$ que cabem na memória da GPU ($bQ$ e $bS$ respectivamente) e na memória RAM ($max\_bQ$ e $max\_bS$ respectivamente). O algoritmo tem como saída várias matrizes de custo acumulado $D$ que




(a) O cálculo dos elementos da matriz $D$ depende dos valores calculados dos elementos anteriores.

(b) P-TWDTW: Cálculo de $D$ de forma paralela

**Figura 1. Cálculo da matriz acumulada $D$**

serão sempre escritas no disco para liberar memória para o processamento dos próximos blocos.

O algoritmo inicia nas linhas 1 a 3 lendo os blocos dos padrões para a fila $queueQ$ e blocos das séries temporais para a fila $queueS$. Este trabalho é realizado por uma *thread* na CPU que fica coordenando o tamanho da fila e da memória RAM de forma que não exceda o seu limite disponível e que as filas não fiquem vazias. Desta forma, não é preciso esperar finalizar esta etapa para começar a carregar os blocos para a memória global da GPU. Entre as linhas 4 e 29, enquanto houver blocos para serem processados, o Algoritmo 1 carrega os blocos para a memória da GPU e calcula a matriz $D$.

Nas linhas 5 a 8, $bQ$ padrões $U$ e $bS$ séries temporais $V$ são carregados para a memória global, juntando os padrões $U$ em um único padrão $bigQ$ e as séries temporais em uma só, denominada $bigS$. Esta união permite que a GPU possa realizar o processamento em bloco utilizando todo o seu poder computacional. Na linha 9, a matriz de custo $\Psi$ é construída a partir de $bigQ$ e $bigS$, com cada *thread* da GPU responsável pelo cálculo de um elemento da matriz.

O cálculo da matriz $D$ é realizado seguindo a ideia apresentada na Figura 1(b), onde cada *thread* fica responsável pelo cálculo do custo de cada elemento da diagonal. Como cada elemento da diagonal depende apenas das duas diagonais anteriores, eles podem ser calculados de forma independente. Como $bigQ$ e $bigS$ possui vários padrões $U$ e séries temporais $V$, são lançados $bQ * bS$ blocos de *threads* na GPU, onde cada bloco de *thread* fica responsável pelo cálculo do custo acumulado entre um padrão $U$ e uma série temporal $V$. Cada bloco de *threads* na GPU possui $min(sizeU, sizeV)$ *threads* que é o tamanho máximo de uma diagonal na comparação entre $U$ e $V$, para que cada *thread* realize o cálculo de um elemento da diagonal.

Entre as linhas 14 e 20 são calculados os custos dos elementos das primeiras $sizeU$ diagonais superiores à diagonal secundária. Em uma matriz quadrada, estas primeiras $sizeU$ diagonais representam a matriz triangular superior. No Algoritmo 1 admite-se que o índice inicia na posição 0. Para identificar cada diagonal, o índice da linha na matriz

superior é calculado como $si - tid$ (*tid* é o índice da *thread*) e o índice da coluna é determinado pelo id da *thread*. A matriz $D$ é atualizada para cada elemento da diagonal utilizando o Algoritmo 2.

**Algoritmo 1:** ptwdtw($Q$, $bQ$, $max\_bQ$, $S$, $bS$, $max\_bS$)

**Entrada**: $Q$: conjunto de padrões $U$ a serem utilizados na consulta
$bQ$: número de padrões $U$ processados ao mesmo tempo
$max\_bQ$: número de padrões $U$ na memória
$S$: conjunto de séries temporais $V$
$bS$: número de séries temporais $V$ processadas ao mesmo tempo
$max\_bS$: número de séries temporais $V$ na memória
**Saída**: Matriz de custo acumulado $D$

```
1  while Tiver mais padrões U e séries temporais V no disco do
2      queueQ ← carrega max_bQ padrões U para a memória RAM
3      queueS ← carrega max_bS séries temporais V para a memória RAM
4      while Tiver mais padrões em queueQ e séries temporais em queueS do
5          gpu_queueQ ← carrega bQ padrões U para a memória global da GPU
6          gpu_queueS ← carrega bS séries temporais V para a memória global da
           GPU
7          bigQ ← junta todos os padrões U em gpu_queueQ
8          bigS ← junta todos as séries temporais V em gpu_queueS
9          Ψ ← calcula a matriz de custo, de forma trivialmente paralela, entre bigQ e
           bigS
10         sizeU ← calcula o tamanho do padrão U dentro de bigQ
11         sizeV ← calcula o tamanho da série temporal V dentro de bigS
12         tid ← id da thread
13         if tid < sizeU then
14             for si de 0 até sizeU - 1 do
15                 if tid ≤ min(si, sizeV − 1) then
16                     i ← si − tid
17                     j ← tid
18                     update_accumulated_cost_matrix(Ψ, D, i, j, sizeU, sizeV)
19                 end
20             end
21             for sj de sizeV − 2 até 0 do
22                 if tid ≤ min(sj, sizeU − 1) then
23                     i ← sizeU − tid − 1
24                     j ← sizeV − sj − tid − 1
25                     update_accumulated_cost_matrix(Ψ, D, i, j, sizeU, sizeV)
26                 end
27             end
28         end
29     end
30     Escreve a matriz D no disco
31 end
```

O algoritmo entre as linhas 21 e 26 calcula os elementos das próximas $sizeV - 1$ diagonais. Em uma matriz quadrada, por exemplo, estas $sizeV - 1$ diagonais representam

a matriz triangular inferior. Para identificar cada diagonal, o índice da linha na matriz superior é calculado como $sizeU - tid - 1$ e o índice da coluna é definido como $sizeV - sj - tid - 1$. A matriz $D$ é então atualizada para cada elemento da diagonal.

O Algoritmo 2 é responsável por atualizar cada elemento $D_{i,j}$ da matriz de custo acumulada $D$. O cálculo de $D_{i,j}$ segue a equação 3, que é o menor valor entre $D_{i-1,j}$, $D_{i-1,j-1}$ e $D_{i,j-1}$ somado ao custo $\Psi_{i,j}$. O índice nas matrizes $\Psi$ e $D$ relativo a $i, j$ das matrizes $U$ e $V$ devem ser calculados, pois as matrizes $\Psi$ e $D$ são enviadas como vetores para a GPU e os padrões $U$ e $V$ em blocos de tamanho $bQ$ e $bS$ respectivamente. Ou seja, é preciso encontrar o par $U$ e $V$ dentro de $D$ e $\Psi$ e depois deslocar o índice no vetor de forma a encontrar a posição correta na matriz relativa a $U$ e $V$. Na execução do programa na GPU são lançados $bQ * bS$ blocos de *threads*, sendo $bQ$ no eixo x e $bS$ no eixo y para que cada bloco de *threads* seja relativo a um bloco em $\Psi$ e $D$.

Para calcular o índice global das matrizes $\Psi$ e $D$ utilizamos a equação 6, onde $(i + blockIdx.x * sizeU) * sizeV * gridDim.y$ encontra a linha correta em $\Psi$ e $D$, sendo $blockIdx.x$ o id do bloco de threads no eixo x e $gridDim.y$ o número de blocos no eixo y. O trecho $(j + blockIdx.y * sizeV)$ encontra dentro da linha a posição correta do elemento $i, j$, sendo $blockIdx.y$ o id do bloco de *threads* no eixo y.

$$index = (i + blockIdx.x * sizeU) * sizeV * gridDim.y + (j + blockIdx.y * sizeV) \quad (6)$$

**Algoritmo 2:** update_accumulated_cost_matrix($\Psi$, $D$, $i$, $j$, $sizeU$, $sizeV$)

**Entrada**: $\Psi$: matriz de custo de entrada
$D$: matriz de custo acumulado
$i$: índice do padrão na matriz de custo
$j$: índice da série temporal na matriz de custo
$sizeU$: tamanho do padrão
$sizeV$: tamanho da série temporal

1 $index_{i,j} \leftarrow$ calcula o índice global em $\Psi$ do índice $i$ e $j$
2 $index_{i-1,j-1} \leftarrow$ calcula o índice global em $\Psi$ do índice $i - 1$ e $j - 1$
3 $index_{i-1,j} \leftarrow$ calcula o índice global em $\Psi$ do índice $i - 1$ e $j$
4 $index_{i,j-1} \leftarrow$ calcula o índice global em $\Psi$ do índice $i$ e $j - 1$
5 $D[index_{i,j}] \leftarrow min(D[index_{i-1,j}], D[index_{i-1,j-1}], D[index_{i,j-1}) + \Psi[index_{i,j}]$

Depois de construída a matriz de custo acumulada $D$, o algoritmo calcula o alinhamento final entre cada $U$ e $V$ seguindo a equação 5. Para cada alinhamento é calculado o custo total do alinhamento que será utilizado como métrica para os algoritmos de análise e classificação de séries temporais.

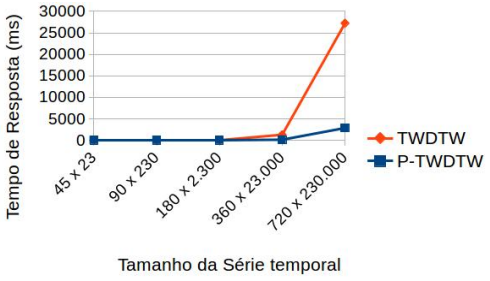
## 5. Avaliação de Desempenho

Diversos testes foram realizados para analisar o desempenho dos algoritmos P-TWDTW e TWDTW. Nos testes na CPU foi utilizada uma máquina com processador AMD FX-8320E (3.2 GHz, 8 MB Cache) e 8 GB de memória RAM. Na execução dos testes na GPU foi uma utilizada uma placa NVIDIA GeForce GTX 1050 Ti com 4 GB GDDR5 de memória disponível e 768 CUDA *cores* com *clock* máximo de 1392 MHz.

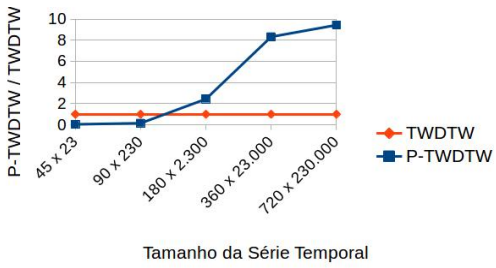
Como a complexidade de I/O da CPU e GPU é igual em relação a leitura e escrita do disco, o tempo de leitura e escrita dos blocos do disco foi desprezado do resultado final para analisar apenas a diferença de desempenho entre o P-TWDTW e o TWDTW. O TWDTW foi implementado para ser processado na CPU e desenvolvido na linguagem C++ e o P-TWDTW na GPU utilizando a linguagem CUDA da NVIDIA. As séries temporais $V$ e os padrões $U$ foram obtidos a partir de dados reais de várias regiões do Brasil [Maus et al. 2016]. Cada teste foi executado dez vezes e o tempo de resposta foi obtido a partir da média das dez execuções. A acurácia dos algoritmos TWDTW e P-TWDTW é igual, já que o algoritmo P-TWDTW gera a mesma saída.

Nos últimos anos houve um aumento da resolução temporal e espacial dos satélites [Battude et al. 2016]. A resolução temporal refere-se à frequência de passagem do satélite num mesmo local, num determinado intervalo de tempo. Alguns satélites, como os satélites da Planet (parceira do Google) [Strauss 2017], tem uma frequência diária de coleta do mesmo local e, em alguns casos, até mais de uma coleta por dia. Por isto, o tamanho dos padrões $U$ e das séries temporais $V$ tem aumentado consideravelmente.

Este cenário de alta resolução temporal está representado na avaliação da Figura 2 que apresenta o gráfico com a comparação do TWDTW com o P-TWDTW sobre um padrão $U$ e uma série temporal $V$, variando o tamanho de ambos. O tempo de resposta aumenta de forma considerável com o TWDTW a medida que aumenta o tamanho do padrão $U$ e da série temporal $V$ (Figura 2(a)). Para séries temporais menores (neste gráfico menores ou iguais a 90 x 230), é mais vantajoso utilizar o TWDTW. Quando o tamanho ultrapassa 90 x 230 tornou-se vantajoso usar o P-TWDTW. Isto se deve ao fato do P-TWDTW utilizar a GPU para o processamento e, neste caso, existe o tempo de transferência de $U$ e $V$ para a GPU que domina o tempo final do algoritmo. Como pode ser visto na Figura 2(b), para séries temporais maiores o P-TWDTW teve tempo de resposta até 10 vezes menor em relação ao TWDTW.




(a) Comparação do tempo de resposta entre o TWDTW e o P-TWDTW.

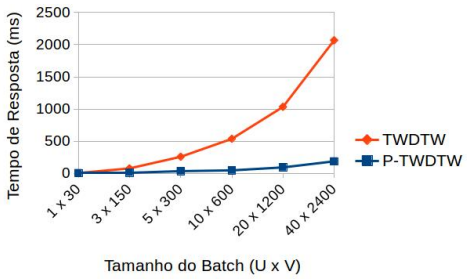



(b) Proporção do tempo de resposta do P-TWDTW em relação ao TWDTW com o aumento do tamanho das séries temporais.

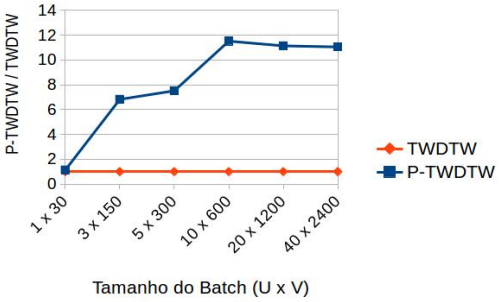
**Figura 2. Comparação do TWDTW com o algoritmo P-TWDTW com apenas um padrão $U$ e uma série temporal $V$ e variando o tamanho de ambos. O eixo $x$ contém o tamanho das séries no formato *tamanho de* $U$ x *tamanho de* $V$.**

A resolução espacial está relacionada com a capacidade de cada sensor em detectar os objetos da superfície terrestre levando em consideração o tamanho do *pixel* da imagem. Quanto maior a resolução espacial, mais *pixels* são gerados para a mesma região. Por isso, quanto maior a resolução espacial mais séries temporais são geradas para uma determinada área geográfica.

A Figura 3 representa este cenário de alta resolução espacial com a comparação do P-TWDTW com o TWDTW utilizando vários padrões $U$ e séries temporais $V$. Este teste utilizou séries temporais pequenas de $U$ e $V$ (45 e 23 respectivamente) para demonstrar a eficiência do P-TWDTW neste cenário de satélites com alta resolução espacial. Como pode ser visto na Figura 3(a), o P-TWDTW teve um desempenho um pouco melhor que o TWDTW com 1 padrão $U$ e 30 séries temporais $V$. A partir de 3 padrões $U$ e 150 séries temporais $V$, o P-TWDTW teve tempo de resposta 7 vezes menor. O P-TWDTW utilizou um tamanho de *batch* igual a 5 para $U$ e 300 para $V$ e, por isso, a partir de 10 padrões $U$ e 600 séries temporais $V$, passou a executar vários *batchs* para calcular o alinhamento entre cada $U$ e $V$. Neste cenário, o P-TWDTW obteve tempo de resposta 11 vezes menor que o algoritmo TWDTW, já que passou a explorar todo o poder computacional da GPU.




(a) Comparação do tempo de resposta entre o TWDTW e o P-TWDTW.




(b) Proporção do tempo de resposta do P-TWDTW em relação ao TWDTW com o aumento do número de padrões e séries temporais.

**Figura 3. Comparação do TWDTW com o algoritmo P-TWDTW com vários padrões $U$ e várias séries temporais $V$. O eixo $x$ contém o *número de padrões* $U$ x *número de séries temporais* $V$. Cada padrão tem tamanho 45 e a série temporal tamanho 23.**

Nos cenários apresentados na avaliação dos algoritmos P-TWDTW e TWDTW é possível perceber que com satélites com alta resolução temporal o P-TWDTW é uma solução promissora visando processar séries temporais maiores, tendo tempo de resposta até 10 vezes menor que o TWDTW. Para satélites com alta resolução espacial, ou seja, um grande número de padrões $U$ e séries temporais $V$, o P-TWDTW se mostrou uma alternativa melhor que o TWDTW com tempo de resposta até 11 vezes menor.

## 6. Conclusões

Na classe de problemas computacionalmente complexos, a análise de séries temporais é um dos problemas com maior demanda de poder computacional [Rakthanmanon et al. 2013], devido a complexidade dos algoritmos e o grande volume de dados a serem processados. O algoritmo TWDTW tem se destacado como a melhor solução encontrada na literatura para análise de séries temporais de imagens de sensoriamento remoto. Este algoritmo, entretanto, não foi projetado para processar de forma eficiente explorando o potencial da computação altamente paralela. Além disso, o algoritmo TWDTW possui complexidade de tempo $O(n^2)$, tornando seu uso inviável para grandes conjuntos de dados.

Este trabalho propôs uma solução paralela capaz analisar grandes volumes de dados de séries temporais de imagens de sensoriamento remoto explorando arquiteturas pa-

ralelas. Esta solução, denominada P-TWDTW (Parallel TWDTW), explorou a arquitetura Manycore (GPU) com o uso coordenado e apropriado da grande quantidade de núcleos disponíveis. O P-TWDTW possui complexidade de tempo $O(\frac{n^2}{p})$, onde $p$ é o número de núcleos disponíveis. O P-TWDTW processa as séries temporais como *batchs*, ou seja, comparando vários padrões e diversas séries temporais ao mesmo tempo na GPU. Estes *batchs* são, em alguns casos, maiores que a memória RAM e a memória da GPU. O P-TWDTW coordena o uso destes espaços de memória para que seja possível processar grande volume de dados explorando o alto número de núcleos disponíveis nas GPUs, sem exceder a capacidade das mesmas.

Na avaliação do algoritmo, o P-TWDTW se mostrou uma solução promissora visando processar séries temporais maiores (alta resolução temporal), com tempo de resposta de até 10 vezes em relação ao TWDTW. O P-TWDTW teve tempo de resposta até 11 vezes menor em relação ao TWDTW para casos onde existem um grande número de padrões e séries temporais para serem comparados (alta resolução espacial).

Em trabalhos futuros pretendemos criar uma versão multi-GPU e uma versão paralela na CPU do P-TWDTW para aproveitar os recursos computacionais presentes em um ambiente com várias GPUs. Também planejamos integrar o algoritmo P-TWDTW à plataforma de análise de séries temporais DistSensing [de Oliveira et al. 2017], explorando os recursos de um *cluster* de computadores. As séries temporais de *pixels* vizinhos espacialmente tem relação entre si [Costa et al. 2017] e a análise também do espaço geográfico pode aumentar a acurácia do P-TWDTW. Assim, pretendemos modificar o P-TWDTW para que explore a arquitetura Manycore adicionando o eixo espacial para aumentar a acurácia de análise das séries temporais.

## Referências

Battude, M., Al Bitar, A., Morin, D., Cros, J., Huc, M., Sicre, C. M., Le Dantec, V., and Demarez, V. (2016). Estimating maize biomass and yield over large areas using high spatial and temporal resolution sentinel-2 like remote sensing data. *Remote Sensing of Environment*, 184:668–681.

Christophe, E., Michel, J., and Inglada, J. (2011). Remote sensing processing: From multicore to gpu. *Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of*, 4(3):643–652.

Costa, W. S., Fonseca, L. M., Körting, T. S., SIMÕES, M., Bendini, H. N., and Souza, R. C. (2017). Segmentation of optical remote sensing images for detecting homogeneous regions in space and time. *In: XVIII BRAZILIAN SYMPOSIUM ON GEOINFORMATICS*, 18:40–51.

de Oliveira, S. S. T., de Castro Cardoso, M., dos Santos, W., Costa, P., do Sacramento Rodrigues, V. J., and Martins, W. S. (2017). Distsensing: A new platform for time series processing in a distributed computing environment. *Revista Brasileira de Cartografia*, 69(5).

Huang, X., Wang, J., Wong, R., Zhang, J., and Wang, C. (2016). Pisa: An index for aggregating big time series data. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 979–988. ACM.

Jamali, S., Jönsson, P., Eklundh, L., Ardö, J., and Seaquist, J. (2015). Detecting changes in vegetation trends using time series segmentation. *Remote Sensing of Environment*, 156:182–195.

João Jr, M., Sena, A. C., and Rebello, V. E. (2017). Implementação e avaliação de técnicas de paralelização no algoritmo de hirschberg para sistemas multicore. *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*.

Juan, W. and Jianchao, S. (2013). A new type of ndvi algorithm based on gpu dividing block technology. In *Computational and Information Sciences (ICCIS), 2013 Fifth International Conference on*, pages 709–712. IEEE.

Kuenzer, C., Dech, S., and Wagner, W. (2015). *Remote Sensing Time Series*. Springer.

Liu, P., Yuan, T., Ma, Y., Wang, L., Liu, D., Yue, S., and Kołodziej, J. (2014). Parallel processing of massive remote sensing images in a gpu architecture. *COMPUTING AND INFORMATICS*, 33(1):197–217.

Maus, V., Câmara, G., Cartaxo, R., Sanchez, A., Ramos, F. M., and de Queiroz, G. R. (2016). A time-weighted dynamic time warping method for land-use and land-cover mapping. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 9(8):3729–3739.

Petitjean, F., Inglada, J., and Gançarski, P. (2012). Satellite image time series analysis under time warping. *IEEE Transactions on Geoscience and Remote Sensing*, 50(8):3081–3095.

Petitjean, F. and Weber, J. (2014). Efficient satellite image time series analysis under time warping. *Ieee geoscience and remote sensing letters*, 11(6):1143–1147.

Rakthanmanon, T., Campana, B., Mueen, A., Batista, G., Westover, B., Zhu, Q., Zakaria, J., and Keogh, E. (2013). Addressing big data time series: Mining trillions of time series subsequences under dynamic time warping. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 7(3):10.

Sakoe, H. (1971). Dynamic-programming approach to continuous speech recognition. In *1971 Proc. the International Congress of Acoustics, Budapest*.

Strauss, M. (2017). Planet earth to get a daily selfie.

Verbesselt, J., Hyndman, R., Newnham, G., and Culvenor, D. (2010). Detecting trend and seasonal changes in satellite image time series. *Remote sensing of Environment*, 114(1):106–115.

Xiao, L., Zheng, Y., Tang, W., Yao, G., and Ruan, L. (2013). Parallelizing dynamic time warping algorithm using prefix computations on gpu. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 294–299. IEEE.

Zhu, H., Gu, Z., Zhao, H., Chen, K., Li, C.-T., and He, L. (2018). Developing a pattern discovery method in time series data and its gpu acceleration. *Big Data Mining and Analytics*, 1(4).

# Introducing drowsy technique to cache line usage predictors

**Rodrigo M. Sokulski, Emmanuell D. Carreno and Marco A. Z. Alves**

Department of Informatics - Federal University of Paraná, Brazil

{rms16, edcarreno, mazalves}@inf.ufpr.br

***Abstract.*** *In the last decades, with the continuous increase in the number of transistors on the same chip, a bigger die area inside the processor have been occupied by the cache memories, in some cases, the caches occupy close to half of the chip area in modern processors. For this reason, more energy is consumed by this dedicated circuit, making energy saving techniques for cache memories an important subject. In this paper, we evaluate the integration of a state-of-the-art dead cache line predictor with the Drowsy technique to enable static energy savings up to 70% in the last level cache.*

## 1. Introduction

For every new processor generation, the industry is providing an always increasing speedup, now surpassing 21% per year [Jeff Preshing 2012]. Such a speedup is achieved mostly by increasing the circuit complexity. The complexity growth is possible by the crescent number of transistors that can be placed in the same chip area and is also limited by it. Nevertheless, analyzing the processor's evolution, we can notice that a large amount of the chip area is being allocated to implement big cache memories.

Although, the processor speed is not limited just by its implementation technology. Also, the data processed by multiple cores need to arrive on the functional units in order to be computed. In this context, the DRAM memory circuits do not work as fast as the processor. As mentioned by [Chang 2017], the DRAM circuit's performance increases in an average rate of 1.5% per year. This discrepancy between processor and DRAM performance leads to the memory wall problem, where the slow memory access time impacts directly on the processor performance.

In order to quickly provide the necessary data to the processor, small and faster cache memories were added between the processor and the main memory. These memories occupy the remaining transistors from the processor chip. Nowadays, cache memories occupy near to 50% of the processors' chip area.

Such big cache memories entail a considerable high energy consumption. This energy consumption comes primarily of two different usage sources. The first is the dynamic energy, which corresponds to the part utilized to perform memory accesses, reads and writes. The second source is the static energy, which is the portion spent even when there is no interaction with the circuit, or the one that leaks energy when maintaining the cache contents stored.

The cache circuits growth is increasing its static energy waste, therefore growing the relevance of energy saving strategies. The main technique to reduce energy waste is the Gated-$V_{DD}$, which turns the cache off when its associated core is in the idle state. On the other hand, some techniques were proposed to save energy for caches

which are still in usage [Lai and Falsafi 2000, Kaxiras et al. 2001, Chen et al. 2004, Kharbutli and Solihin 2008, Khan et al. 2010, Alves et al. 2013], obtaining great energy savings. These mechanisms try to predict when the cache lines receive its last access, thus becoming dead, and turning these lines off right after this last access. The problem with such an approach is that when a cache line is mispredicted to be dead, it is turned off and lose its contents. Thus, in a case of a misprediction, its next access to this cache line will incur in a cache miss, leading to extra DRAM access, therefore, wasting dynamic energy and time.

Other approaches for reducing the static energy spent by a cache line were proposed by [Nii et al. 1998, Flautner et al. 2002]. These approaches, reduce the static energy consumption of a cache line without losing its contents. A direct consequence is that they prevent access to one of the lines in the saving mode, which generate unneeded access to the DRAM. Also, those mechanisms require only a small fraction of the DRAM access time to regain access to the stored data.

In this paper, we evaluate a blend of a state-of-the-art deadline predictor, called Skewed Dead Predictor (SDP) [Khan et al. 2010] and a method to reduce the static energy waste without losing the cache line contents [Flautner et al. 2002]. Our proposal presents the following main contributions:

- Mechanism blend: We propose a mechanism that uses the Skewed Dead Predictor (SDP) [Khan et al. 2010] and the Drowsy cache a static energy saving method [Flautner et al. 2002].
- Static energy and performance tradeoff: We conclude that by using SDP with Drowsy technique instead of Gated-$V_{DD}$ we reduce the static energy gains. However, we observed that we could reduce the execution time overhead when using this Drowsy technique.
- Dynamic energy saving: By preventing that the mispredictions of our deadline predictor generate extra DRAM accesses, we reduce the dynamic energy consumed by the memory circuits. This reduction in the dynamic energy balance the total energy consumption makes SDP with Drowsy technique achieve similar results than using Gated-$V_{DD}$.

The rest of this paper is organized as follows: Section 2 presents preliminary results showing the potential of dead cache line predictors. Section 3 brings the implementation details for the SDP mechanism and the Drowsy technique. Section 4 shows the simulation and benchmark parameters, also presenting the results comparing traditional SDP with Drowsy SDP. Related work is present in Section 5. Finally, Section 6 draws conclusions and future work ideas.

## 2. Motivation

In this section, we will show the results of executing an oracle mechanism in the last level cache (LLC), capable of turning off cache lines after they receive their last access. The purpose is to evaluate the maximum possible gains of using the mechanisms that perform deadline predictions. More details regarding the simulation parameters and the benchmarks used in our evaluation will be further presented on section 4.1.

The oracle mechanism implemented turns off the cache lines during the time between the last access and the cache line eviction (i.e., while the lines are dead). This

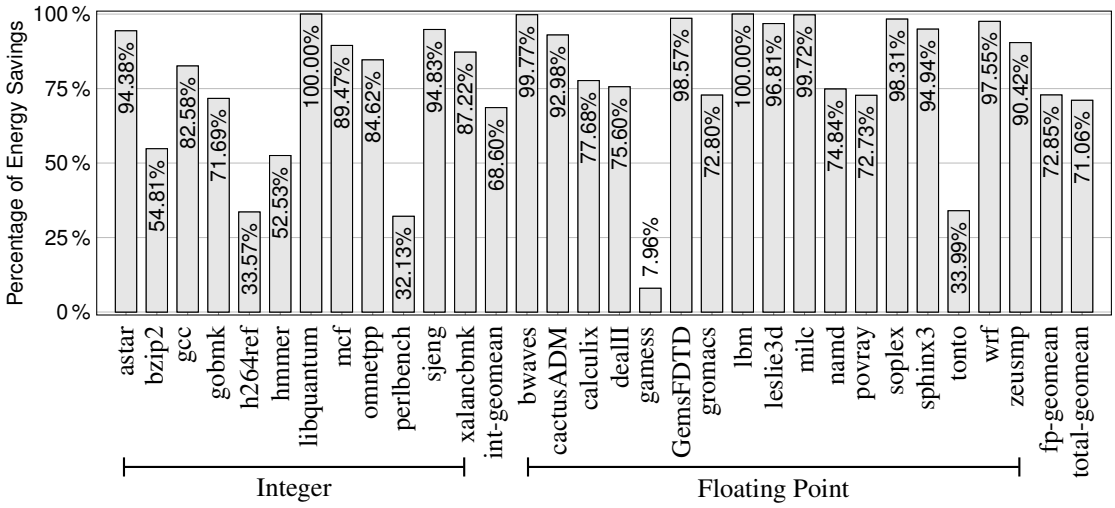



**Figure 1. Oracle results regarding static energy savings potential.**

mechanism acts on the last level cache, where the low access frequency leads the cache line to extended dead periods. Besides, no modification in the replacement policy is performed by the oracle, even for the lines predicted to be dead.

Figure 1 shows the results obtained using the oracle, showing the maximum energy savings in the LLC, therefore, the time that the LLC lines could be turned off. We can observe that on average more than 70% of the time, the cache lines could be turned off without impacting on the system performance. This preliminary result correlates with previously reported numbers [Alves 2014].

Previous work [Alves et al. 2013, Khan et al. 2010] proposed to predict when the cache line is dead and to turn it off. Although such mechanisms achieve energy savings in the cache context, during the underpredictions (i.e., to predict that a cache line is dead before its final access) these mechanisms negatively impact the number of DRAM accesses, requiring extra time and energy consumption on each access.

Considering this drawback from previous work, in this paper, we propose to combine deadline cache predictors with the Drowsy [Flautner et al. 2002] technique which reduces the cache line voltage in order to save some of the static energy spent by the line, while preserving its contents. The primary goal of our approach is to reduce the impact of mispredictions, as accesses to a line already in a low voltage state does not incur in access to DRAM since the line content is not lost.

Although multiple deadline predictors could be modified to use the Drowsy cache technique, we choose the Skewed Dead Predictor (SDP) due to its higher accuracy and low area overhead as pointed out in section 3.

## 3. Our proposal

Our proposed mechanism is based on the SDP prediction mechanism and the Drowsy cache line method. Therefore, in order to explain its operation, in the following subsections, we are going to expose these mechanisms.

### 3.1. Skewed Dead Predictor - SDP

The Skewed dead block predictor as proposed by [Khan et al. 2010] utilizes the instructions trace that leads a cache line to its eviction in order to predict its last access occurrence. This mechanism assumes that the instructions that make the last access to a cache line before its eviction tend to lead other cache line data into eviction. With this idea, the mechanism can track the instruction trace from the cache lines and predict a line as dead right after a common last access instruction trace is executed after it.

The mechanism utilizes at least two tables to make its predictions. These two tables are composed of entries of saturated counters in order to track the instruction trace that leads to the last access of a cache line content. In order to index these tables, the mechanism uses skewed functions, one function for each table. These functions use some bits from each instruction that accessed the cache line to generate different indexes for each table, thus, avoiding mispredictions caused by coincident table mappings.

Each time that a cache line is evicted, the SDP tables indexed position is incremented. Therefore, on each cache line access, the mechanism discovers if each access will lead into the last access of the cache line content just by verifying the sum of the tables indexed position. If this sum is greater than a fixed and manually defined threshold, the line is predicted as having received its last access, therefore becoming a deadline.

Among all the deadline predictors we choose SDP due to its low area overhead (near to one-third of required by DEWP [Alves et al. 2013]) while still providing accurate predictions.

### 3.2. Drowsy technique

The Drowsy technique [Flautner et al. 2002] permits the cache line voltage change between two possible voltages. The higher one is the normal cache line voltage. The lower one permits a reduction of static energy waste. However, the lower state prevents direct accesses to the line content. In other words, in order to access the cache line content while doing reads or writes, the cache line voltage must be increased to the normal voltage, consuming 1 or 2 processor cycles in this process.

In order to reduce the extra latency and dynamic energy spent by the SDP mechanism mispredictions, we propose to blend the SDP predictor with the Drowsy cache line technique. In this blend, the voltage of the lines predicted as dead is reduced, therefore saving static energy. In the following subsections, we are going to describe the additions required in the cache by our proposal and its operations.

### 3.3. Implementation details

In order to implement the SDP predictor, it requires some components. Figure 2 illustrates the SDP mechanism implementation and its details. The first addition is two tables indexed by 14 bits with 2-bit saturated counters entries needed to store the predictions. Moreover, in order to index these prediction tables, the mechanism requires an additional 15-bit entry in each cache line, that entry is used to store the instruction trace that leads to that last cache line access. The threshold used in our experiments was 2, the same proposed by [Khan et al. 2010].

The prediction tables are indexed with the skewed functions, one for each table, described in [Seznec 1993]. These functions are applied over the 15-bit trace from the

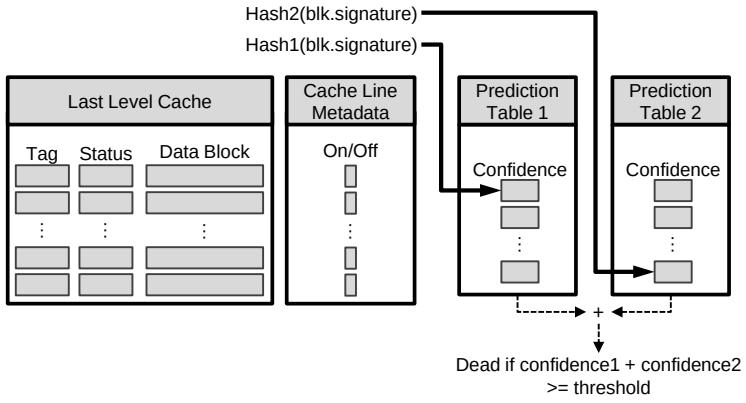



**Figure 2. Components necessary for SDP.**

cache lines, providing a different indexation to each prediction table, therefore, avoiding the same mapping for different traces.

The trace contained in each cache line is updated utilizing the addresses from each instruction that access that cache line content. The update procedure consists of performing an initial exclusive OR with the first 15 bits from the instruction address that is accessing the cache line with its next 15 bits and then add the resulting value to the current cache line trace.

The mechanism that implements the Drowsy lines technique also requires some other additions; a voltage controller in order to provide the different voltage levels and a word line gating circuit that is used to prevent access into Drowsy lines.

Regarding the storage area overhead, the SDP implementation of a 2 MB last level cache requires 4 KB to store the on/off bit for each cache line. Plus, it also requires two prediction tables with the total size of 8 KB (32,768 entries). Besides, the Drowsy technique is expected to cause a 3.35% overhead on the data array of the cache memory [Flautner et al. 2002]. Thus, we can conclude that a total LLC area (tag plus data arrays) overhead of 2.8% is required to implement SDP with Drowsy technique.

### 3.4. Mechanism operations

The main operations performed by our proposal are described as follows:

**Cache Line Access:** The prediction table entries indexed by cache line trace are decremented. Then the trace is updated with the new accessing instruction, and the sum of the corresponding prediction table entries is compared with a threshold. If it is bigger than the threshold, the cache line Drowsy mode is activated.

**Cache line Eviction:** The corresponding entries from the prediction tables are incremented in order to save that trace as an eviction source.

**Cache line Arrival:** The arriving cache line trace is updated, then the sum of the prediction table entries indexed is verified. If it exceeds a defined threshold, it is one dead on arrival line, then that line is put in the Drowsy mode.

It is important to notice that in our implementation of SDP we choose to use the simple LRU replacement policy, without sending dead cache lines for early eviction. This implementation choice was supported by the fact that in our preliminary evaluations

the simple LRU SDP implementation achieved near to 3× higher gains concerning LLC energy savings.

## 4. Methodology and results

This section will present the simulation setup for experiments and the results regarding our proposal to merge dead cache line prediction mechanism and Drowsy caches technique.

### 4.1. Experimental setup

Our simulator was built over the Intel dynamic binary instrumentation tool Pin [Intel 2018]. It assumes that non-memory instructions are executed in a single cycle (ideal IPC equals to 1), and it models all the cache hierarchy of memory operations.

This single cycle simulation approach minimizes the simulation time, maintaining the error compared to a cycle-accurate simulator in less than 5% [Lee et al. 2010].

In order to obtain the latency and the power consumption of the cache memories, we used CACTI 6.5 [Muralimanohar et al. 2008]. The baseline cache hierarchy we model (based on Intel Coffee Lake's cache hierarchy) is described in Table 1. The line size is 64 bytes.

| Memory | Size | Associativity | Cycles | Avr. Idle Power | Avr. Dynamic Energy |
|---|---|---|---|---|---|
| L1 | 32 KB | 8 Ways | 4 | Not modeled | Not modeled |
| L2 | 256 KB | 4 Ways | 8 | Not modeled | Not modeled |
| L3 | 2 MB | 16 Ways | 26 | 263 mW | 0.198 nJ |
| RAM | 8 GB | – | 250 | 372 mW | 3.270 nJ |

**Table 1. General parameters for the baseline cache model.**

The results for the mechanisms execution include three average columns calculated using the geometric mean, one column for the average of integer-based applications (*int-geomean*), another one for the floating point-based applications (*fp-geomean* and the final column at each figure with the average of all the applications executed (*total-geomean*).

### 4.2. Mechanism accuracy

After implementing the SDP mechanism, we performed the first experiment in order to show the precision of such a mechanism. In order to understand the potential of SDP, whenever we evicted a cache line from the LLC, we analyzed it in order to classify as correct or wrongly predicted. If the cache line gets evicted and it is turned off, then it accounts for a correct prediction. In case the cache line is turned on by the time it gets evicted (losing energy savings opportunities), or when the cache line is turned off and receives new access (degrading the performance), in both cases, the cache line accounts for wrongly predicted. Notice that whenever the mechanism is learning a new pattern, it may also lose energy saving opportunities, and so, it will be reported as mispredictions. This classification occurs due to the nature of the mechanism which does not have a way to distinguish between training or regular prediction mode.

Figure 3 shows the precision results for the SPEC-CPU 2006 benchmark applications split into integer and floating point. On average, the mechanism prediction was

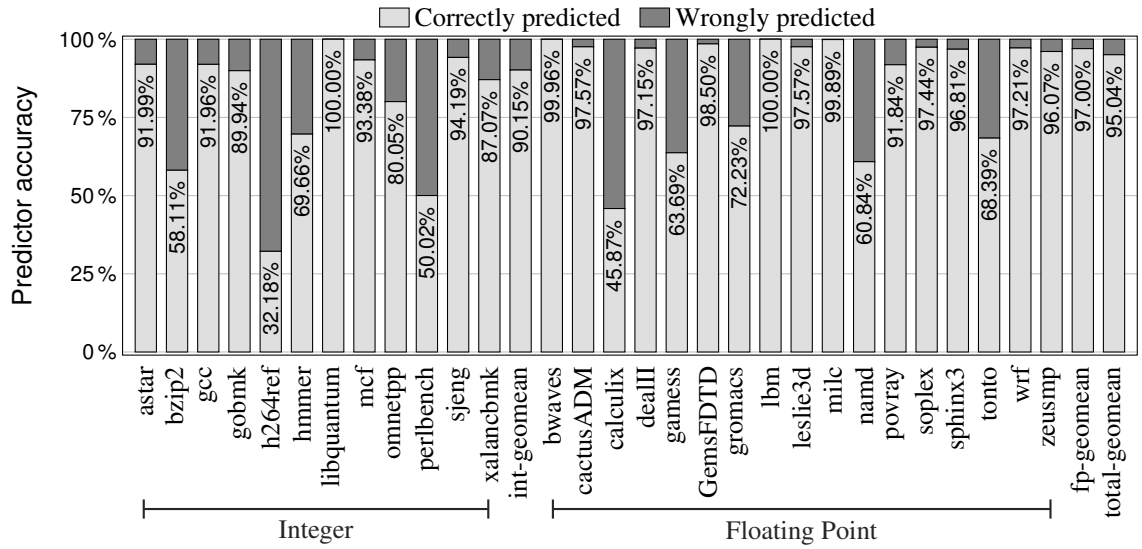



**Figure 3. SDP accuracy results for SPEC-CPU 2006 benchmark applications.**

correct nearly 90% of the times. In some cases as with libquantum and lbm benchmarks, we achieved 100% accuracy on the predictions. Analyzing the accuracy results we could notice that those top 10 applications with more LLC evictions present the highest accuracy for SDP, and the contrary is also true, the mechanism present less accuracy for those applications with a lower amount of LLC evictions. Meaning that streaming behavior applications tend to have more repetition in the access pattern thus have higher accuracy in SDP.

### 4.3. Maximum energy savings

In order to understand the maximum gains possible using SDP, figure 4 present the percentage of time the cache lines could be turned off by our mechanism. We can notice that on average 46% of the time the cache lines could be turned off (or changed to Drowsy mode).That means that the mechanism can achieve 65% of the potential demonstrated by the oracle mechanism.

Nevertheless, it is important to note that such gains can be reduced if the number of accesses to the DRAM increases. Thus, the next results will present the SDP using Gated-$V_{DD}$ or Drowsy techniques.

### 4.4. Energy consumption

In this section, we present energy results from the SDP mechanism performing Gated-$V_{DD}$ or using Drowsy techniques over the predicted to be dead cache lines.

Figure 5 shows the LLC energy consumption results normalized to the baseline without any deadline predictor. We can observe that such results directly correlates with the previous results, regarding the percentage of time the LLC lines were predicted as dead.

However, we estimate that the Drowsy technique would reduce the LLC static energy consumption by 75% while the Gated-$V_{DD}$ would reduce by 100%. Thus, we can observe that on average, the original SDP with Gated-$V_{DD}$ saved near to 4$\times$ more LLC static energy than using the Drowsy technique.

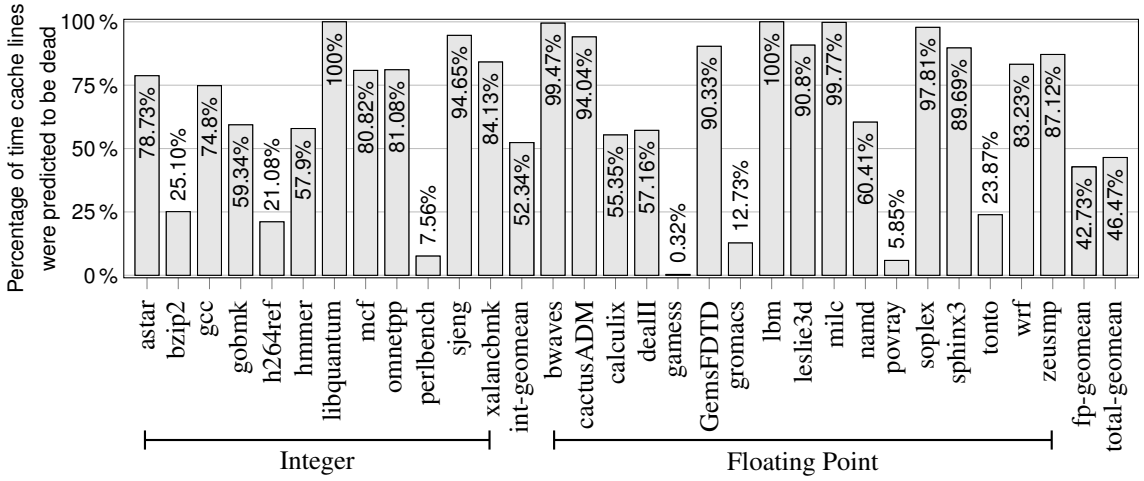



**Figure 4. Percentage of time the cache lines were predicted to be dead.**

In Figure 6 we can observe that the amount of static energy saved by both mechanisms was similar. For this experiment, we take into account the static energy spent by the LLC and the static plus dynamic energy for the DRAM. Considering that the number of accesses received by the cache memories will remain stable, with or without our proposal, we did not consider the dynamic energy for LLC nor L1 and L2.

We can observe that on average both mechanisms achieved energy savings of near 25%. However, the Drowsy technique allows marginally lower gains most of the time. Again, this happens because Drowsy technique would reduce the LLC static energy consumption by 75% while the Gated-$V_{DD}$ would reduce it by 100%. Nevertheless, considering that Drowsy technique reduces the number of DRAM accesses, the difference in the final energy savings compared to Gated-$V_{DD}$ is less than 2% on average.

### 4.5. DRAM accesses and execution time

In order to fully understand the sources of gains regarding energy consumption, in this section, we present the results concerning DRAM accesses and execution time for SDP with Gated-$V_{DD}$ or Drowsy technique.

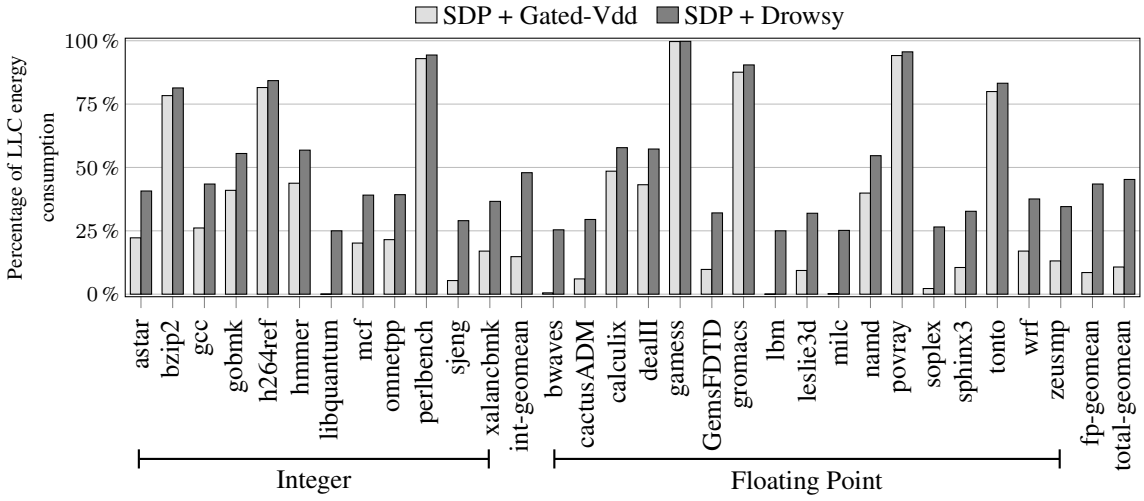



**Figure 5. LLC static energy consumption.**

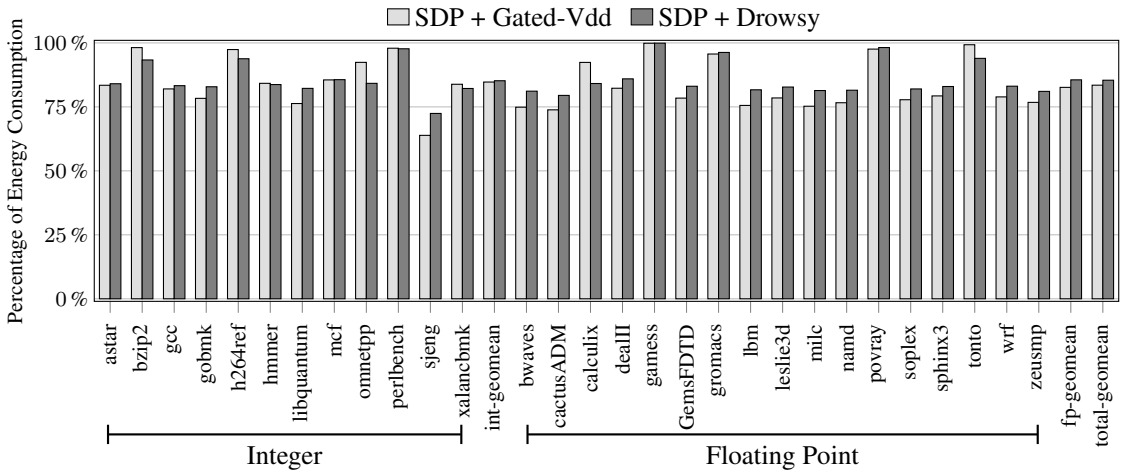



**Figure 6. Total energy consumption considering LLC static energy plus DRAM static and dynamic energy.**

Figure 7 shows the percentage of overhead accesses to DRAM generated by using SDP with Gated-$V_{DD}$. The overhead means the percentage of time that the SDP Gated-$V_{DD}$ mechanism generated extra access to the DRAM when the cache line was turned off and the data at the line previously stored was needed. Because Drowsy does not turn off cache lines, the amount of accesses is the same as the baseline. For this reason, only the overhead of SDP with Gated-$V_{DD}$ is shown. As shown in the Figure 7, on average, is possible to reduce the excess of DRAM accesses by 15% by correctly predicting the deadlines using drowsy.

The impact on the application' execution time is shown in Figure 8. On average, SDP with Gated-$V_{DD}$ creates a higher overhead than with Drowsy, (4% on average). Meanwhile, the overhead caused by using Drowsy was less than 1% on average.

## 5. Related work

Some other work over the cache lines static energy saving problem has already been done; in this section, we present some of these mechanisms.

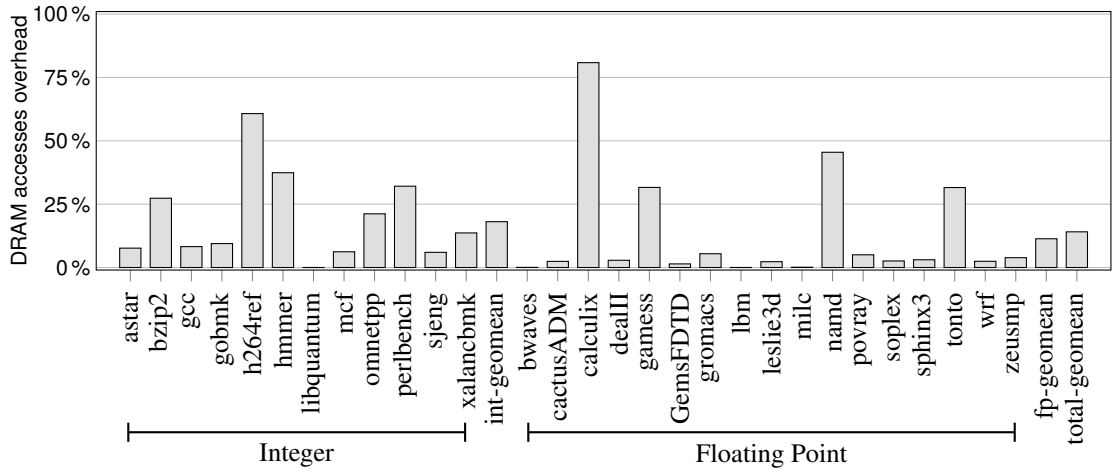



**Figure 7. DRAM accesses for SDP with Gated-$V_{DD}$ normalized to baseline.**

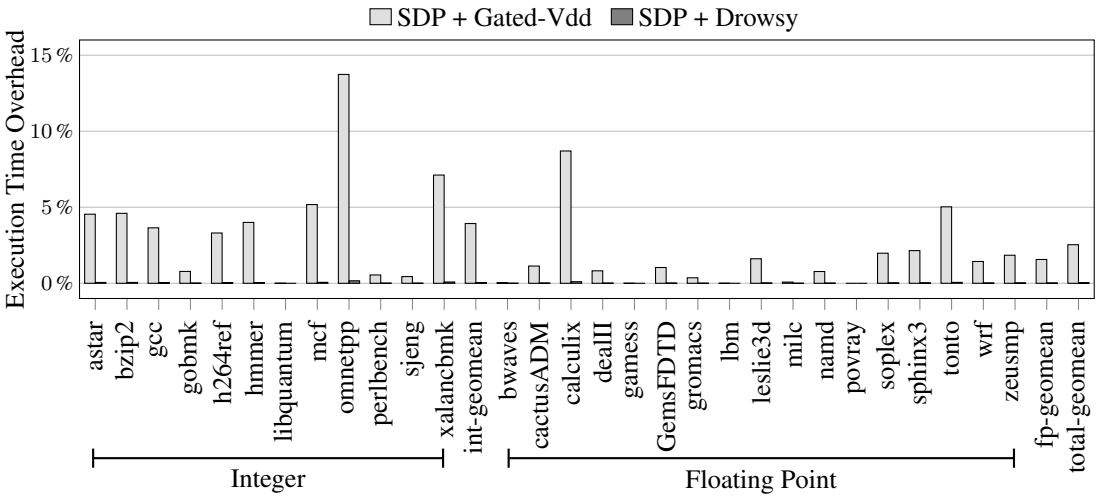



**Figure 8. Execution time overhead normalized to baseline.**

## 5.1. Cache line usage predictors

[Kaxiras et al. 2001] has proposed to use the number of cycles since the last cache line access in order to define the death of a cache line. Two methods were proposed, a manually defined threshold and an adaptive threshold, dependent on the application behavior. This mechanism differs from our propose by using the cycle number between consecutive accesses to define if a cache line is dead, while our proposal uses the instruction trace that leads the cache lines to define a deadline.

[Chen et al. 2004] proposes a predictor that uses its predictions in order to just bring to the cache, the lines that are going to have future accesses. It differs from our proposal since we bring all data to the cache and just turn off the predicted as dead.

The Dead Line and Early Write-Back Predictor (DEWP) [Alves et al. 2013] is used to detect the cache line death, turning them off, and the last write received by a cache line, sending it to an early write-back. The predictions made by its mechanism are based on the information contained in an access history table (AHT), that contains information about the number of accesses received from the content of the cache line in the last time it was taken into the cache. The main difference between this mechanism and our proposal is that we utilize the instructions trace to make our predictions, while this previous work utilizes the number of accesses to the cache lines.

## 5.2. Leakage energy reduction techniques

[Powell et al. 2000] proposed a mechanism that turns off the cache lines in order to save the static energy spent by them. It consists of a transistor between the memory cell and the ground path or the supply voltage that can be used to cut the energy provided to the cell, therefore saving almost all the static energy spent by it. The main difference between this technique to ours is that we do not turn off the cache line, so its content is preserved.

[Nii et al. 1998] proposed a technique that can reduce the static energy spent by the cache lines without losing its contents, adding some few components to each SRAM cell. The difference between this method and the Drowsy lines adopted by our proposal

is that this method requires more energy and time to make the state transition, while ours just requires 1 or 2 cycles.

The prediction mechanisms mentioned above apply the policy of turning off the lines predicted as dead while the methods of saving static energy mentioned do not make forecasts about the lines that should be turned off. On the other hand, our proposal utilizes a mechanism to predict the deadlines and do not turn them off, using a technique to reduce their static energy waste without losing its contents. This combination, to the best of our knowledge, never was evaluated.

## 6. Conclusion and future work

Current multicore processors have a large amount of chip area being occupied by ever-increasing cache memories. Such memories spend static and dynamic energy during its operation. Although dynamic energy is spent at reading and writing operations, static energy is spent whenever the cache line is turned on.

In order to save static energy, previous work proposed dead cache line predictors that turn cache lines off whenever it receives its last access. However, such mechanisms impose overhead in terms of time and DRAM energy during mispredictions.

In this paper, we propose to combine the Drowsy cache technique to deadline predictors in such a manner that instead of turn dead cache lines off, we can reduce its leakage by using the Drowsy technique.

In our evaluations, we can observe energy savings of 70% on average with no extra DRAM accesses, leading to negligible performance loss of less than 1%.

As future work, we consider evaluating the Drowsy integration to other dead cache line predictors in order to obtain the best mechanism combination.

## 7. Acknowledgment

The authors would like to acknowledge the generous support by the serrapilheira Institute (grant number Serra-1709-16621). This work was also financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

## References

[Alves 2014] Alves, M. A. Z. (2014). *Increasing Energy Efficiency of Processor Caches via Line Usage Predictors*. PhD thesis, Rio Grande do Sul Federal University, Porto Alegre, RS, Brazil.

[Alves et al. 2013] Alves, M. A. Z., Villavieja, C., Diener, M., and Navaux, P. O. A. (2013). Energy efficient last level caches via last read/write prediction. In *2013 25th International Symposium on Computer Architecture and High Performance Computing*, pages 73–80.

[Chang 2017] Chang, K. K. (2017). *Understanding and Improving the Latency of DRAM-Based Memory Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA.

[Chen et al. 2004] Chen, C. F., Yang, S. H., Falsafi, B., and Moshovos, A. (2004). Accurate and complexity-effective spatial pattern prediction. In *Software, IEE Proceedings-*, pages 276–287.

[Flautner et al. 2002] Flautner, K., Kim, N. S., Martin, S., Blaauw, D., and Mudge, T. (2002). Drowsy caches: simple techniques for reducing leakage power. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 148–157.

[Intel 2018] Intel (2018). Pin - a dynamic binary instrumentation tool.

[Jeff Preshing 2012] Jeff Preshing, H. P. (2012). A look back at single-threaded cpu performance.

[Kaxiras et al. 2001] Kaxiras, S., Hu, Z., and Martonosi, M. (2001). Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proceedings 28th Annual International Symposium on Computer Architecture*, pages 240–251.

[Khan et al. 2010] Khan, S. M., Jiménez, D. A., Burger, D., and Falsafi, B. (2010). Using dead blocks as a virtual victim cache. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 489–500, New York, NY, USA. ACM.

[Kharbutli and Solihin 2008] Kharbutli, M. and Solihin, Y. (2008). Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447.

[Lai and Falsafi 2000] Lai, A.-C. and Falsafi, B. (2000). Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 139–148.

[Lee et al. 2010] Lee, H., Jin, L., Lee, K., Demetriades, S., Moeng, M., and Cho, S. (2010). Two-phase trace-driven simulation (tpts): a fast multicore processor architecture simulation approach. *Software: Practice and Experience*, 40(3):239–258.

[Muralimanohar et al. 2008] Muralimanohar, N., Balasubramonian, R., and Jouppi, N. (2008). Architecting efficient interconnects for large caches with cacti 6.0. *IEEE Micro (IEEE MICRO)*, 28(1):69–79.

[Nii et al. 1998] Nii, K., Makino, H., Tujihashi, Y., Morishima, C., Hayakawa, Y., Nunogami, H., Arakawa, T., and Hamano, H. (1998). A low power sram using auto-backgate-controlled mt-cmos. In *Proceedings. 1998 International Symposium on Low Power Electronics and Design (IEEE Cat. No.98TH8379)*, pages 293–298.

[Powell et al. 2000] Powell, M., Yang, S.-H., Falsafi, B., Roy, K., and Vijaykumar, T. N. (2000). Gated-vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, ISLPED '00, pages 90–95, New York, NY, USA. ACM.

[Seznec 1993] Seznec, A. (1993). A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 169–178, New York, NY, USA. ACM.

# Solução de Monitoramento Térmico em *Data Centers*

**Ademir Camillo Jr., Maurício A. Pillon, Charles C. Miers, Guilherme P. Koslovski**

[1]Programa de Pós Graduação em Computação Aplicada (PPGCA) - LabP2D
Universidade do Estado de Santa Catarina - UDESC - Joinville, SC - Brasil
*ademir.junior@edu.udesc.br,* {*mauricio.pillon, charles.miers, guilherme.koslovski*}*@udesc.br*

O fornecimento crescente de serviços de Tecnologia da Informação (TI) demanda cada vez mais por capacidade computacional, tipicamente fornecida por *Data Center*s (DCs). A temperatura interna de DCs sofre influência de diversos fatores, tais como: localização geográfica, o projeto de sistema de refrigeração, a localização física dos equipamentos de processamento e de rede, ou ainda da carga de trabalho dos equipamentos de TI em geral. No Brasil, apenas 8% dos DCs utilizam *Computer Room Air Conditioning* (CRAC) [Schneider 2014]. Alguns fatores que podem influenciar na formação de zonas térmicas quentes que excedam as especificadas nas normas: má distribuição física dos equipamentos de TI, ausência de condução das correntes de ar frio/quente ou a variação da carga de processamento dos equipamentos.

O monitoramento térmico *online* é uma importante ferramenta para auxiliar a tomada de decisão do administrador do DC [Levy and Hallstrom 2017, Sasakura et al. 2017]. O mapeamento do comportamento térmico em um DC depende da identificação das correntes de ar no ambiente [Sasakura et al. 2017, Camillo et al. 2017]. A arquitetura *Monitoramento Término de Data Centers* (MonTerDC) tem como foco de atuação ambientes de DC de pequeno e médio porte, sem sistemas de refrigeração complexos, com restrições orçamentárias. Muitas vezes, estes ambientes são implantados em locais adaptados e sem projetos físico, lógico ou de refrigeração próprios.

Os testes realizados com MonTerDC permitiram uma análise da relação do consumo de energia com climatização e as zonas térmicas criadas dinamicamente dentro do DC. Identificou-se como as cargas de processamento dos servidores afetam as correntes de ar e zonas térmicas. Em um estudo de caso real, MonTerDC proporcionou análise fina das zonas térmicas, constatando que o consumo energético para uma mesma carga de processamento pode ser 57% superior devido a localização física do servidores.

## Agradecimentos

Os autores agradecem o apoio do Laboratório de Processamento Paralelo e Distribuído (LabP2D)/UDESC e da FAPESC.

## Referências

[Camillo et al. 2017] Camillo, A. J., Miers, C. C., Koslovski, G. P., and Pillon, M. A. (2017). Análise de zonas térmicas em data center não-crac. In *WSCAD 2017*, Campinas, SP.

[Levy and Hallstrom 2017] Levy, M. and Hallstrom, J. O. (2017). A new approach to data center infrastructure monitoring and management (DCIMM). In *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1–6.

[Sasakura et al. 2017] Sasakura, K., Aoki, T., and Watanabe, T. (2017). Study on data center optimal management by utilizing data center infrastructure management. In *2017 IEEE International Telecommunications Energy Conference (INTELEC)*, pages 604–608.

[Schneider 2014] Schneider, E. (2014). Soluções em climatização para data center. Brasília/Brasil. XIV Encontro Nacional de Empresas Projetistas e Consultores da Abrava.

# Arquitetura Híbrida de Armazenamento para Internet das Coisas

**Braulio L. D. C. Junior[1], Douglas D. J. de Macedo[2], Diego Kreutz[3], Edward David Moreno[1], Mario A. R. Dantas[4]**

[1]Universidade Federal de Sergipe (UFS)
Departamento de Ciência da Computação (DCOMP)
São Cristóvão, SE – Brazil

[2]Universidade Federal de Santa Catarina (UFSC)
Departamento de Ciência da Informação (CIN)
Florianópolis, SC – Brazil

[3]Universidade Federal do Pampa (UNIPAMPA)
Laboratório de Estudos Avançados (LEA)
Alegrete, RS – Brazil

[4]Universidade Federal de Juiz de Fora (UFJF)
Departamento de Ciência da Computação (DCC)
Juiz de Fora, MG – Brazil

{bldcjunior,edwdavid}@gmail.com, douglas.macedo@ufsc.br, diego.kreutz@unipampa.edu.br, mario.dantas@ice.ufjf.br

***Abstract.*** *Internet of Things (IoT) is becoming part of our life. Some studies predict a considerable market growth with an increase of up to 5 times on the number of connected devices by 2020. A significant challenge is how to store and manage this amount of data generated. In this paper, a hybrid storage architecture for IoT is proposed. The most used NoSQL databases types and two essential types of data of IoT sensors were selected. The results suggest that the hybrid storage architecture is a feasible and promising approach to address the ever-growing amount of data generated in IoT. Additionally, after an architecture evaluation, the results have shown that Redis achieved a better overall performance.*

***Resumo.*** *Internet of Things (IoT) está se tornando parte de nossa vida. Alguns estudos prevêem um crescimento de mercado considerável com um aumento de até 5 vezes no número de dispositivos conectados até 2020. Um significativo desafio é como gerenciar essa grande quantidade de dados gerados. Neste artigo, uma arquitetura de armazenamento híbrido para IoT é proposta. Os tipos de armazenamento NoSQL mais usados e dois tipos essenciais de dados em IoT foram utilizados. Os resultados obtidos sugerem que a arquitetura de armazenamento híbrido é uma abordagem viável para abordar a grande quantidade de dados gerados em IoT. Além disso, após uma avaliação da arquitetura, os experimentos mostraram que o Redis alcançou um melhor desempenho geral.*

# Otimização de Recursos em ICNs através de Cache Distribuído usando Software Defined Networking – SDN

**Erick B. Nascimento**[1], **Douglas D. J. de Macedo**[2], **Edward D. Moreno**[1]

[1]Programa de Pós-Graduação em Ciência da Computação (PROCC)
Universidade Federal de Sergipe (UFS)
Avenida Marechal Rondon, S/n - Jardim Rosa Elze, São Cristóvão - SE, 49100-000

[2]Departamento de Ciência da Informação (CIN)
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brasil

erick.nascimento@dcomp.ufs.br,douglas.macedo@ufsc.br,edwdavid@gmail.com

***Abstract.*** *The reduction of the traffic of network segment by the implementation of caches has been research object by the exponential increase of data request through the network. Even though we have high-speed connections, the conventional model still depends on end-to-end communication for established the exchange between two end systems. According to Cisco Virtual Networking Index (VNI), its estimated that by 2021 we will have around 4.6 billion Internet users, with 27.1 billion connected devices and connections. Therefore, the Information Centric Networks (ICN) has been widely discussed as the new content distribution model for the Future of the Internet, alongside this, the paradigm research of Software Defined Networks (SDN) proposes the flexibility in the development of content networks. Therefore, an architecture for ICN networks orchestrated through SDN is proposed focusing on the use of redundant caching to reduce network degradation caused by the replication of data in the same segment.*

***Resumo.*** *A redução do tráfego de segmentos de rede pela implementação de caches, tem sido objeto de pesquisa devido o aumento exponencial de requisição de conteúdos através da rede. Mesmo que tenhamos conexões de alta velocidade, o modelo convencional ainda depende da comunicação fim a fim para estabelecer a troca entre dois sistemas finais. De acordo com o Cisco Virtual Networking Index (VCNI), estima-se que até 2021 teremos cerca de 4,6 bilhões de usuários de Internet, com 27,1 bilhões de dispositivos conectados e conexões. Por esse motivo, Information Centric Networks (ICN) tem sido amplamente discutido como modelo de distribuição de conteúdo para o futuro da internet [Xylomenos et al. 2014], paralelamente, o paradigma de redes programáveis Software Defined Networks (SDN) propõe a flexibilidade necessária no desenvolvimento de redes dinâmicas. Logo, neste trabalho é proposta uma arquitetura para ICN orquestrada através de SDN, com foco na utilização de cache redundante, buscando reduzir a degradação da rede causado pela duplicação de conteúdo no mesmo segmento.*

# Aplicação do Balanceador de carga SmartLB para redução do tempo de execução e do consumo de energia de aplicações em ambientes paralelos *

**Vinicius R. S. dos Santos[1], Edson L. Padoin[1,2], Philippe O. A. Navaux[1,2]**

[1]Universidade Reg. do Noroeste do Estado do Rio G. do Sul (UNIJUI) - Ijuí - RS - Brasil

{vinicius.ribas,padoin}@unijui.edu.br

[2]Universidade Federal do Rio Grande do Sul (UFRGS) - Porto Alegre - RS - Brasil

navaux@inf.ufrgs.br

***Abstract.*** *This paper presents a proposal of a new load balancer which aims reducing the runtime and power consumption of parallel applications when these are runned in shared memory environments. The algorithm of the balancer collects system and application information in real time and then uses it to make task migration decisions. For the implementation of strategy the* CHARM++ *parallel programming model was used. Preliminary results show reductions of up to* $35.36\%$ *on runtime and energy consumption for three benchmarks used in the tests.*

***Resumo.*** *Este artigo apresenta a proposta de um novo balanceador de carga para a redução do tempo de execução e o consumo de energia de aplicações paralelas quando executadas em ambientes de memória compartilhada. O algoritmo do balanceador coleta informações do sistema e da aplicação em tempo real e as utiliza para tomar decisões de migrações de tarefas. Para a implementação foi utilizado o modelo de programação paralela* CHARM++. *Os resultados preliminares apresentaram redução em média de de até* $35,36\%$ *do tempo de execução e no consumo de energia para três benchmarks utilizados nos testes.*

## 1. Introdução

A quantidade de aplicações científicas simuladas em ambientes paralelos tem aumentado cada vez mais. Estas simulações computacionais representam atualmente um percentual significativo no total das demandas por processamento nos sistemas de HPC e, somente são possíveis com o emprego de programação paralela. Com paralelismo, o processamento das aplicações é dividido em partes e executado em paralelo nas unidades de processamento disponíveis nos atuais de sistemas de alto desempenho.

Para atender às grandes demandas de processamento, diferentes arquiteturas paralelas tem sido projetas e construídas empregando processadores compostos de múltiplas unidades de processamento. No entanto, a maioria das aplicações paralelas apresentam comportamento dinâmico, que acabam gerando desbalanceamento de cargas, ou excessiva comunicação entre os processos, o que dificulta uma eficiente utilização dos sistemas de computação que geralmente possuem unidades de processamento homogêneos.

*Trabalho que tem recebido recursos pelo projeto Petrobras número 2016/00133-9 e que tem sido apoiado pelo programa EU H2020 e do MCTI/RNP-Brasil sob o projeto HPC4E de número 689772.

# Melhorando o Desempenho de Operações de E/S do Algoritmo RTM Aplicado na Prospecção de Petróleo *

**Pablo J. Pavan[1], Matheus S. Serpa[1], Edson L. Padoin[2], Lucas M. Schnorr[1], Philippe O. A. Navaux[1], Jairo Panetta[3]**

[1] Universidade Federal do Rio Grande do Sul (UFRGS) - Porto Alegre - RS - Brasil

{pablo.pavan,msserpa,schnorr,navaux}@inf.ufrgs.br

[2]Universidade Reg. do Noroeste do Estado do Rio G. do Sul (UNIJUI) - Ijuí - RS - Brasil

padoin@unijui.edu.br

[3]Divisão de Ciência da Computação ITA - São José dos Campos – SP – Brasil

jairo.panetta@gmail.com

***Abstract.*** *Computational simulations of physical phenomena are only possible due to the computational resources available in High Performance Computing (HPC) systems. These applications generally generate a large volume of data which is read and written to disk during the simulation. Since these operations are costly, this becomes one of the performance bottlenecks of these applications. In this way, this work analyzes and proposes optimizations for the input and output operations of a geophysical application that uses the Reverse Time Migration (RTM) algorithm. The results show that using checkpoints and increasing the size of the request reduces application execution time by up to* $17.33\%$.

***Resumo.*** *As simulações computacionais de fenomênos físicos só são possíveis devido aos recursos computacionais disponíveis em sistemas de Computação de Alto Desempenho (CAD). Essas aplicações, geralmente geram um grande volume de dados os quais são lidos e escritos no disco durante a simulação. Uma vez que essas operações são custosas, isso se torna um dos gargalos de desempenho dessas aplicações. Neste sentido, este trabalho analisa e propõe otimizações para as operações de entrada e saída de uma aplicação geofísica que utiliza o algoritmo Reverse Time Migration (RTM). Os resultados mostram que a utilização de checkpoints e o aumento do tamanho da requisição reduz o tempo de execução da aplicação em até* $17,33\%$.

*Este trabalho foi parcialmente financiado pela Intel sobre o projeto Intel Modern Code e pela Petrobras sobre o projeto 2016/00133-9.

# Análise de Algoritmos Paralelos e Vetorizados para a Migração Kirchhoff Pré-empilhamento em Tempo em Ambientes Virtualizados

**Rodrigo Alves Prado da Silva**[1], **Cristiana Barbosa Bentes**[2] **e Lúcia Maria de Assumpção Drummond**[1]

[1]Universidade Federal Fluminense (UFF), Niterói, RJ, Brasil
[2]Universidade do Estado do Rio de Janeiro (UERJ), Maracanã, RJ, Brasil

rodrigo_prado@id.uff.br, cris@eng.uerj.br, lucia@ic.uff.br

***Resumo.*** *Um dos métodos mais populares de migração sísmica é a Migração Kirchhoff Pré-empilhamento em Tempo (PKTM) que é computacionalmente intensiva. Como a computação em nuvem provou ser uma alternativa promissora para executar aplicações de computação de alto desempenho, neste trabalho, são avaliados os impactos introduzido pelas camadas de virtualização, que normalmente oferecem suporte a serviços em nuvem, sobre as diferentes implementações do PKTM. Assim, foi analisado o desempenho de diferentes versões paralelas do PKTM, utilizando MPI e OpenMP, e vetorizadas, automaticamente ou manualmente, quando executadas com tecnologias de virtualização comumente usadas em nuvens: KVM, Docker e Singularity. Nossos resultados mostraram que, em relação a todas as versões do PTKM, vetorizadas ou não, todas as tecnologias de virtualização testadas introduziram pequenos overheads, apresentando diferenças em tempos de execução de aproximadamente 2% em média, quando comparados com a execução sem virtualização. Também foi possível observar que as virtualizações baseadas no Singularity e Docker tiveram um desempenho melhor do que a gerenciada pelo KVM.*

***Palavras-chave****: Kirchhoff, paralelismo, vetorização, virtualização.*

***Abstract.*** *One of the most popular methods of seismic migration is the Kirchhoff Migration Pre-Stacking in Time (PKTM) which is computationally intensive. Because cloud computing has proven to be a promising alternative forrunning high-performance computing applications, the impacts introduced by virtualization layers, which typically support cloud services, on the different implementations of the PKTM are evaluated in this paper. Thus, the performanceof different parallel versions of PKTM, using MPI and OpenMP, and vectorized, automatically or manually when performed with virtualization technolo-gies commonly used in clouds: KVM, Docker and Singularity were analyzed. Our results showed that, in relation to all PTKM versions, vectorized or not, all virtualization technologies tested introduced small overheads, with differences in execution times of approximately 2% on average when compared to running without virtualization. It was also possible to observe that virtualization based on Singularity and Docker performed better than the one managed by KVM.*

***Keywords****: Kirchhoff, Parallelism, Vectorizing, Virtualization.*

# Acceleration of a Computational Simulation Application for Radiofrequency Ablation Procedure using GPU

**Marcelo Miletto**[1], **Claudio Schepke**[1]

[1]Laboratório de Estudos Avançados – Universidade Federal do Pampa (UNIPAMPA)
Av. Tiarajú, 810, 97546-550, Alegrete – RS – Brasil

marcelocm97@gmail.com, claudioschepke@unipampa.edu.br

***Abstract.*** *Computational simulation is a technique used in several research areas. In the medicine area, the RAFEM program (Radiofrequency Ablation Finite Element Method) [Jiang et al. 2010] was developed, which is used to simulate the RFA (RadioFrequency Ablation) process, that is a medical procedure to treat hepatic cancer. This program presents a high computational time to perform a simulation, taking up to 20 hours for a simulation while the RFA procedure itself lasts from four to six minutes. Some efforts have already been carried out to obtain a better performance for the program [Miletto and Schepke 2017], however, none of them considered the use of GPUs for acceleration of the application, which is something quite studied for finite element method programs as reported in the work of [Georgescu et al. 2013]. The aim of this work is to propose a parallelization approach to explore ways to obtain better performance for the application through the use of GPUs using the CUDA parallel programming interface. Thus CUDA kernels were developed to parallelize the assembly step on the program, which is one of its most costly steps. In the assembly stage of the matrix of each element of the finite element mesh there are no race conditions, however, the same is not true for the grouping of these matrices into a single matrix, the global matrix. This matrix represents the problem as a system of linear equations that models the behavior of the characteristics to be simulated. To perform the assembly of this matrix the technique of graph colouring was used, thus avoiding the race conditions generated by neighboring elements that write in a same position of the matrix. The computational environment used for experiments consists of two Intel Xeon E5-2650 processors and two Nvidia GPUs: one Tesla C2075 and one Quadro M5000. This approach was applied in the parallel version of the application assembly step, improving performance up to 17 times against the sequential code.*

***Keywords —*** *RAFEM, Computational Simulation, Finite Element Method, GPU, CUDA.*

## Referências

Georgescu, S., Chow, P., and Okuda, H. (2013). GPU acceleration for FEM-based structural analysis. *Archives of Computational Methods in Engineering*, 20(2):111–121.

Jiang, Y., Mulier, S., Chong, W., Diel Rambo, M. C., Chen, F., Marchal, G., and Ni, Y. (2010). Formulation of 3d finite elements for hepatic radiofrequency ablation. *International Journal of Modelling, Identification and Control*, 9(3):225–235.

Miletto, M. C. and Schepke, C. (2017). Uso do método multi frontal para acelerar uma aplicação de ablação ao por radiofrequência. *Anais do WSCAD-WIC 2017*.

# On the Performance of Multithreading Applications under Private Cloud Conditions

**Anderson M. Maliszewski[1], Dalvan Griebler[1,2], Adriano Vogel[1,2], Claudio Schepke[3]**

[1] Laboratório de Pesquisas Avançadas para Computação em Nuvem (LARCC)
Faculdade Três de Maio (SETREM) – Três de Maio – RS – Brasil

[2]Programa de Pós-Graduação em Ciência da Computação
Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Porto Alegre – RS – Brasil

[3]Universidade Federal do Pampa (UNIPAMPA),
Laboratório de Estudos Avançados (LEA), Alegrete – RS – Brasil

Email: andersonmaliszewski@gmail.com

***Abstract.*** *Cloud computing paradigm is changing the way that services are delivered over the Internet, providing elastic resource capabilities and ensuring QoS [Vogel et al. 2016]. Through virtualization technologies such as KVM or LXC, a cloud can dynamically abstract computational resources and offer them on-demand. Our goal is to compare two cloud environments (*dedicated *- one instance with full machine resources scaling up to 8 threads and* shared *- two instances with half machine resources scaling up to 4 threads) and native (Ubuntu 14.04) as baseline, using different virtualization technologies (LXC v.1.0.8 and KVM v.2.0.0) under a private cloud (CloudStack 4.8). The tests were performed with a well-known benchmark suite of multithreading applications (PARSEC 3.0), in which five benchmarks compiled with native inputs (*`Canneal, Ferret, x264, Bodytrack and Dedup`*) were chosen. This work extends the studies on cloud performance evaluation of [Griebler et al. 2018]. In addition, the number of threads were limited to the number of vCPUs available and Hyperthreading was intentionally disabled. The results demonstrated that performance in cloud varies according to specific characteristics of each application, environment, and virtualization technology. In some applications, a negligible overhead is noticed in both cloud scenarios. On the other hand, specific characteristics like memory locality and management, and I/O bound operations tend to introduce significant overheads both in KVM-based (*`Canneal`* about 18% in* dedicated *and 12% in* shared machine resources *with 2 threads) and LXC-based cloud (*`Dedup`* overhead up to 53% with 7 threads) instances.*

***Keywords—*** *Cloud Computing, Virtualization, IaaS, Performance, Cloustack, PARSEC.*

## References

[Griebler et al. 2018] Griebler, D., Vogel, A., Maron, C. A. F., Maliszewski, A. M., Schepke, C., and Fernandes, L. G. (2018). Performance of Data Mining, Media, and Financial Applications under Private Cloud Conditions. In *23rd IEEE Symposium on Computers and Communications (ISCC)*, Natal, Brazil. IEEE.

[Vogel et al. 2016] Vogel, A., Griebler, D., Maron, C. A. F., Schepke, C., and Fernandes, L. G. (2016). Private IaaS Clouds: A Comparative Analysis of OpenNebula, CloudStack and OpenStack. In *24th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 672–679, Heraklion Crete, Greece. IEEE.

# Uma Análise sobre Utilização de CPU da Aplicação Wordcount em Nós Hadoop YARN Virtualizados Utilizando a Plataforma Xen.

**Marcela T. G. Santos[1], Katyusco de F. Santos[1], Edlane de O. G. Alves[1], Ana Cristina A. de O. Dantas[1]**

[1]Instituto Federal de Educação, Ciência e Tecnologia da Paraíba (IFPB)

{marcela.tassyany,katyusco,edlane.oliveira,ana.oliveira}@ifpb.edu.br

A integração das tecnologias de computação em nuvem e de processamento distribuído tem permitido que empresas e pessoas em geral processem seus conjuntos de dados para atingir os mais diversos objetivos. O Apache Hadoop é um *framework open source* que fornece um modelo de programação simples que torna transparente para programadores o desenvolvimento de aplicações que necessitam de processamento distribuído. O projeto Hadoop é composto por dois principais módulos: o sistema de arquivos distribuídos Hadoop ( HDFS - *Hadoop Distributed File System*) que fornece acesso de alto rendimento e armazenamento de dados e um gerenciador de recursos (YARN - *Yet Another Resource Negotiator*) que possibilita o planejamento de tarefas e gerenciamento de recursos de *cluster*. Este trabalho surge com o objetivo de investigar qual a influência do número de nós virtualizados (VMs) e do tamanho da base de dados a ser processada, no percentual de utilização de CPU dos nós durante a execução de uma aplicação distribuída (*Wordcount*) sob o *Framework* Hadoop. O ambiente virtualizado implementado neste trabalho utilizou um servidor PowerDell 2950 com 32GB de Memória Principal RAM, dois Processadores Intel Xeon 1596 MHz e 750 GB de disco. Uma das VMs foi instanciada como *Master* e as demais como *Slaves*, todas conectadas pelo *switch* virtual do Xen (vSwitch). Sob este servidor foi executado o Hipervisor Citrix Xen Server 7.2.0 x64 com a finalidade de instanciar 11 VMs utilizadas para compor o *cluster* Hadoop. Todas as VMs idênticas em configuração (2GB de Memória Principal RAM, 1 CPU Virtual e 50GB de Disco). *Scripts* foram desenvolvidos com três repetições e em cada uma delas o programa *WordCount* foi executado, enquanto o percentual de utilização de CPU de cada nó foi coletado. Os fatores que variaram foram o tamanho dos dados de entrada (2GB, 4GB, 6GB, 8GB, 10GB, 12GB, 14GB, 16GB) e o número de nós (1 a 10 Slaves). Os resultados evidenciam que sobretudo para bases de dados maiores, o percentual de utilização de CPU tende a diminuir conforme o número de nós aumenta (Correlação de *Pearson* de -0.98 quando processado 16GB). Foi constatado, também, que para as bases de dados menores há uma variação no percentual de CPU alocado entre os *Slaves*, uma vez que o *Resource Manager* opta por não realizar o balaceamento da carga equanimemente entre os nós *Slaves*. Isso posto, avalia-se que, para cenários em que o objetivo seja reduzir o percentual de utilização de CPU, como ambientes em que aplicações distribuídas concorrem por processamento entre si, os resultados confirmaram que o acréscimo de nós reduz o percentual de utilização de CPU, entretanto, a distribuição e alocação de processamento não se dá de forma equânime em todos os cenários.

Palavras-chave: Big Data. Hadoop. Virtualização.

# Lista de Autores