Anais

WSCAD-WIC 2017

Workshop de Iniciação Científica do WSCAD 2017 (XVIII Simpósio em Sistemas Computacionais de Alto Desempenho)

17 a 20 de outubro de 2017
Campinas – SP, Brasil

Organização e Edição
Ricardo dos Santos Ferreira
Wellington Santos Martins

ISSN 2358-6613

Promoção

Co-Sponsor

Organização

Patrocínio Diamante

Patrocínio Ouro

Patrocínio Prata

Patrocínio Bronze

Patrocínio Básico

Agências de Fomento

FICHA CATALOGRÁFICA

Workshop de Iniciação Científica do WSCAD 2017 (XVIII Simpósio em Sistemas Computacionais de Alto Desempenho) WSCAD-WIC 2017
(17-20 outubro 2017: Campinas – SP, Brasil)

Anais / Organizadores:
Ricardo dos Santos Ferreira, Wellington Santos Martins.

Campinas, SP: SBC, 2017 159f. : il.

ISSN 2358-6613

1. Processamento de Alto Desempenho. 2. Arquitetura de Computadores. 3. Programação Paralela. 4. Agoritmos Paralelos e Distribuídos. 5. Sistemas Distribuídos. I. WSCAD (17-20 outubro 2017): Campinas, SP. II. SBC. III. Ricardo dos Santos Ferreira. IV. Wellington Santos Martins.

WSCAD 2017

XVIII Simpósio em Sistemas Computacionais de Alto Desempenho
Workshop de Iniciação Científica
17 a 20 de outubro de 2017
Centro de Convenções do Expo Dom Pedro, Campinas, São Paulo, Brasil
http://wscad.sbc.org.br

O Simpósio de Sistemas Computacionais de Alto Desempenho (WSCAD) é um evento anual que apresenta, desde o ano 2000, os principais desenvolvimentos, aplicações e tendências nas áreas de Arquitetura de Computadores, Processamento de Alto Desempenho e Sistemas Distribuídos.

Em sua décima oitava edição, o WSCAD foi realizado em conjunto com o *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* em Campinas, São Paulo. A cidade de Campinas foi fundada em 14 de julho de 1774. Entre o final do século XVIII e o começo do século XX, a cidade teve o café e a cana-de-açúcar como importantes atividades econômicas. Porém, desde a década de 1930, a indústria e o comércio são as principais fontes de renda, sendo considerada um polo industrial regional. Décima cidade mais rica do Brasil, hoje é responsável por pelo menos 15% de toda a produção científica nacional, sendo o terceiro maior polo de pesquisa e desenvolvimento brasileiro. Ela também possui diversos atrativos turísticos, com valor histórico, cultural ou científico, como museus, parques e teatros.

Além das sessões técnicas da trilha principal do WSCAD e dos minicursos, o WSCAD 2017 contou com os seguintes eventos co-alocados:

- Workshop de Iniciação Científica em Arquitetura de Computadores e Computação de Alto Desempenho (WSCAD-WIC)
- Concurso de Teses e Dissertações em Arquitetura de Computadores e Computação de Alto Desempenho (WSCAD-CTD)
- Workshop sobre Educação em Arquitetura de Computadores (WEAC)
- Maratona de Programação Paralela

A programação do WSCAD-WIC contou com 2 sessões técnicas com 12 apresentações orais e 13 apresentações de pôsteres, os quais foram publicados digitalmente na Biblioteca Digital Brasileira de Computação (BDBCOMP).

Índice

Mensagem dos Coordenadores

Bem-vindos ao WSCAD-WIC 2017, o Workshop de Iniciação Científica em Arquitetura de Computadores e Computação de Alto Desempenho (WSCAD-WIC). O Workshop é uma oportunidade para os alunos de graduação apresentarem o que aprenderam através de suas experiências de pesquisa para uma audiência maior. Ele também representa um fórum para estudantes, professores e comunidade discutir tópicos de pesquisa de ponta e para examinar a relação entre pesquisa e ensino. O evento inclui apresentações orais e de pôsteres de estudantes de graduação de diferentes universidade brasileiras.

O WSCAD-WIC 2017 teve nesta edição 34 artigos submetidos por estudantes de graduação de mais de 20 instituições de ensino e pesquisa nacionais. Cada artigo foi avaliado por pelo menos 3 avaliadores. Ao final do processo de avaliação, composto por um comitê de 48 pesquisadores da área, 35% dos artigos foram selecionados para apresentação oral e 38% dos artigos foram selecionados para apresentação na forma de pôsteres.

Para finalizar gostaríamos de agradecer a todos jovens pesquisadores que submeteram o resultado de seus trabalhos para o WSCAD-WIC 2017, assim como aos seus orientadores que se dedicaram à escrita e submissão dos artigos. Não podemos deixar de agradecer também a todos os membros do comitê de programa e a todos os revisores pelo excelente trabalho realizado.

Desejamos a todos um excelente e proveitoso evento!

Ricardo dos Santos Ferreira (UFV)
Wellington Santos Martins (UFG)
Coordenadores do WSCAD-WIC 2017

Comitês Organizadores

Coordenação Geral

- Rodolfo Azevedo (UNICAMP)

Coordenação do Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)

- César Augusto Fonticielha De Rose (PUCRS)
- Márcio Castro (UFSC)

Coordenação do Workshop de Iniciação Científica em Arquitetura de Computadores e Computação de Alto Desempenho (WSCAD-WIC)

- Ricardo dos Santos Ferreira (UFV)
- Wellington Santos Martins (UFG)

Coordenação do Concurso de Teses e Dissertações em Arquitetura de Computadores e Computação de Alto Desempenho (WSCAD-CTD)

- Aleardo Manacero (UNESP)
- Edward David Moreno (UFS)

Coordenação do Workshop sobre Educação em Arquitetura de Computadores (WEAC)

- Gabriel P. Silva (UFRJ)
- Ivan Saraiva (UFPI)

Coordenação da Maratona de Programação Paralela

- Calebe de Paula Bianchini (Mackenze)

Coordenação dos Minicursos

- Alexandro Baldassin (UNESP)

Comitê de Programa

- Alexandre Carissimi (UFRGS)
- Alexandro Baldassin (UNESP-IGCE)
- Anderson Faustino (UEM)
- Andre Du Bois (UFPel)
- Andrea Charao (UFSM)
- Arlindo Conceição (UNIFESP)
- Calebe Bianchini (Mackenzie)
- Carlos Holbig (UPF)
- Carlos Augusto Martins (PUC Minas)
- Claudio Schepke (UNIPAMPA)
- Cristiana Bentes (UERJ)
- Denise Stringhini (Unifesp)
- Douglas Macedo (UFSC)
- Edson Midorikawa (USP)
- Edward Moreno (UFS)
- Enrique Carrera (ESPE)
- Gerson Geraldo H. Cavalheiro (UFPel)
- Guilherme Galante (Unioeste)
- Gustavo Girão (UFRN)
- Helio Guardia (UFsCar)
- Henrique C. Freitas (PUC Minas)
- Ivan Silva (UFPI)
- Jorge Barbosa (Unisinos)
- Josemar Souza (UNEB)
- Joubert Lima (UFOP)
- Luís Fabrício Wanderley Góes (PUC Minas)
- Lucas Schnorr (UFRGS)
- Marcelo Lobosco (UFJF)
- Marcos Cavenaghi (Humber IT&AL)
- Maria Clicia Castro (UERJ)
- Mario Dantas (UFSC)
- Mauricio Pilla (UFPEL)
- Márcio Kreutz (UFRN)
- Monica Pereira (UFRN)
- Nahri Moreano (UFMS)
- Renato Ishii (UFMS)
- Ricardo Menotti (UFsCar)
- Rodrigo Bortoletto (IFSP)
- Rodrigo Righi (Unisinos)
- Silvio Fernandes (UFERSA)
- Tiago Ferreto (PUCRS)

Revisores Convidados

- Eduardo Inacio (UFSC)
- Emmanuel Marrocos (UFSCar)
- Fernando Puntel (UFSM)
- José Saito (UFSCar)
- José Augusto Nacif (UFV)
- Ricardo da Rocha (UFG)
- Sand Correa (UFG)
- Sergio T. Carvalho (UFG)

Sessões Técnicas

Sessão I – Sala: Carvalho III
Terça-feira, 17/10, 09h30-10h30

Sessão II – Sala: Carvalho III
Terça-feira, 17/10, 10h30-12h00

Sessão de Pôsteres – Local: Coffee-break
Quinta-feira, 19/10, 09h30-10h00 e 15h30-16h00

Usando o *benchmark Rodinia* para comparação de *OpenCL* e *OpenMP* em aplicações paralelas no coprocessador *Intel Xeon Phi*

Leonardo Tavares Oliveira[1], **Ricardo Menotti**[1]

[1]Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 – 13.565-905 – São Carlos – SP – Brazil

{628174@comp, menotti@dc}.ufscar.br

Abstract. *The use of accelerators has become popular in HPC, being present in several of the computers in the Top 500. Among them, the Intel Xeon Phi coprocessor stands out due to its large number of cores and use of x86 architecture, allowing an easier writing and adaptation of codes for use with the coprocessor. This paper uses the Rodinia benchmark to compare 3 different ways of programming parallel, two using the OpenMP library and one using the OpenCL language. By testing different number and types of thread affinity, one can see the non-uniformity of scalability, as well as the influence of overhead on codes with a small number of iterations.*

Resumo. *O uso de aceleradores vem se popularizando em HPC, estando presentes em diversos dos computadores presentes na Top 500. Dentre eles o coprocessador Intel Xeon Phi se destaca devido à sua grande quantidade de núcleos e arquitetura x86, permitindo maior facilidade para escrita e adaptação de códigos para uso com o coprocessador. Esse trabalho utiliza o benchmark Rodinia para comparar 3 formas diferentes de utilizar programação paralela, sendo duas usando a biblioteca OpenMP e uma utilizando a linguagem OpenCL. Por meio do teste de diferente número e tipo de afinidade de threads, é possível ver a não uniformidade de escalabilidade, assim como a influência do overhead em códigos com um número pequeno de iterações.*

1. Introdução

Com o avanço da Computação de Alto Desempenho (HPC[1]) surgiram diversas bibliotecas e linguagens para a otimização e aumento de produtividade para a programação paralela. Dentro desse contexto, destacam-se a biblioteca *OpenMP* e a linguagem *OpenCL*, possuindo diferentes formas e modelos de programação. O *OpenMP* é focado na criação e uso de *threads* em *CPUs*, já o *OpenCL* no uso de hierarquia de memória em *GPUs* e aceleradores por meio de *kernels*.

A partir dessa realidade, este trabalho busca realizar a comparação de três algoritmos com características diferentes em três modelos distintos de execução, sendo os dois primeiros usando *OpenMP*: Somente CPU, Offload para acelerador e *OpenCL*. A organização do restante deste artigo é descrita a seguir.

[1]Do inglês: *High-Performance Computing*

Na Seção 2 são apresentados trabalhos relacionados da área. Na Seção 3 é descrito o *benchmark* utilizado nesse artigo, assim como o procedimento para escolha dos algoritmos a serem testados. Na Seção 4 são expostos os resultados experimentais obtidos por meio do *benchmark*. Na Seção 5 são apresentadas as conclusões desse trabalho, juntamente com propostas para futuros trabalhos.

2. Trabalhos Relacionados

[Misra et al. 2013] realizam uma análise de dois algoritmos presentes no benchmark Rodinia, ambos implementados em *OpenMP*, rodando-os e comparando o resultado em três formas diferentes: Somente CPU, Nativo e *Offload*. Chegando à conclusão que para nem todas as aplicações o uso de *offload* é viável, devido principalmente ao *overhead* causado pela transferência de dados pela *PCIe*.

Em [Ramachandran et al. 2013] os autores utilizam o coprocessador para acelerar códigos da *NASA Advanced Supercomputing Division*. O código é executado somente em modo nativo, removendo o problema do *overhead* causado pela passagem de dados pelo barramento *PCIe*, permitindo o foco em aspectos diferentes para otimização.

Não pudemos, entretanto, encontrar artigos que utilizassem o coprocessador *Intel Xeon Phi* para um *benchmark* com *OpenCL*. O mais próximo a essa situação é em [Memeti et al. 2017], onde dois computadores, um deles utilizando a *GPU GTX Titan X* e o outro com um coprocessador *Intel Xeon Phi*, rodam versões em *OpenCL* e *CUDA* (no primeiro) e versões em *OpenMP* (no segundo), não possuindo assim dados do desempenho dos algoritmos em *OpenCL* com o coprocessador.

Esta carência de artigos que utilizam *OpenCL* com a *Intel Xeon Phi* pode ser atribuída principalmente ao enfoque do *OpenCL* ser para uso em *GPUs* e aceleradores (como *FPGAs*). Este fato demonstra uma carência e necessidade de analisar a viabilidade do uso de *OpenCL* em uma arquitetura como a x86 (presente no coprocessador *Intel Xeon Phi*).

3. Benchmark Rodinia

O *Rodinia Benchmark Suite* foi criado pela *University of Virginia* com o intuito de ajudar arquitetos de sistemas e programadores a testar e comparar *hardware* paralelo com diversos tipos de algoritmos computacionalmente intensivos [Che et al. 2009]. Atualmente o Rodinia está na sua versão 3.1 possuindo 23 algoritmos implementados em *CUDA*, sendo 22 desses algoritmos implementados em *OpenCL* e 19 em *OpenMP*.

Devido à não uniformidade de versões disponíveis, foi necessária uma análise prévia dos algoritmos, selecionando os que possuiam implementação em ambos *OpenMP* e *OpenCL*. Posteriormente, foram escolhidas as implementações em *OpenMP* que possuiam versões com a opção de *offload* para o coprocesssador, e por fim, implementações que utilizam arquivos externos como fonte de dados, de forma a garantir a uniformidade dos cálculos.

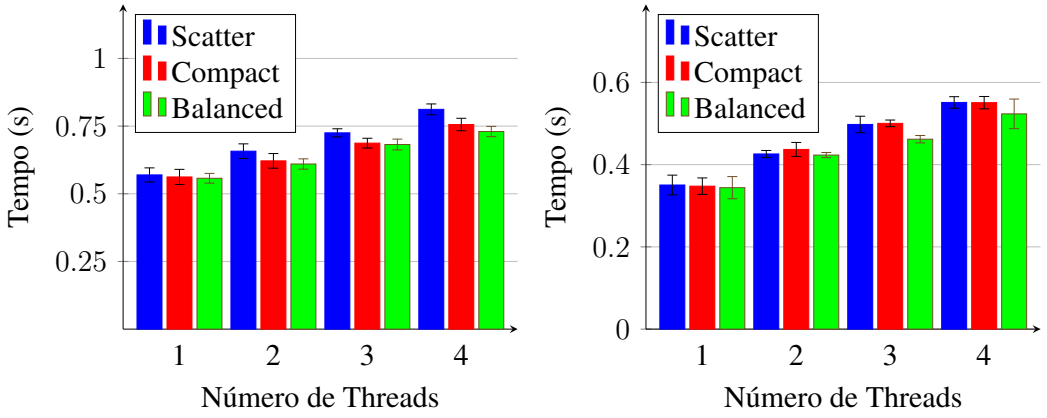
4. Resultados experimentais

Escolhidos os algoritmos CFD, NW e HotSpot, foi realizado o teste de afinidade de *threads* em cada um dos *benchmarks*, buscando encontrar o melhor tipo de afinidade juntamente com o melhor número de *threads* por núcleo. Foi utilizado um computador com

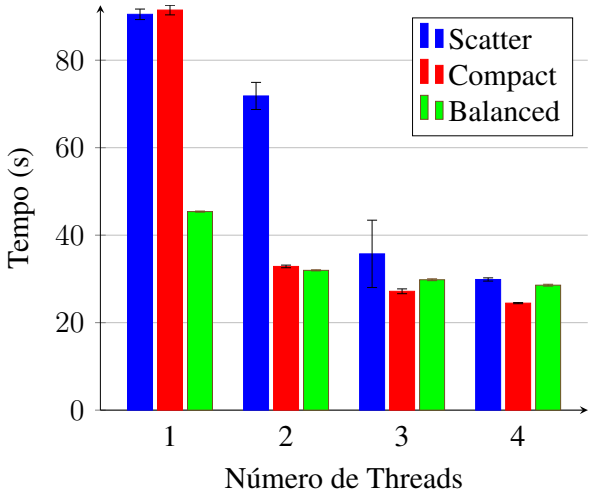
Processador Intel® Xeon® E5-1607 v3 (4 cores, 3.1 GHz, 10 Mb Cache), Linux CentOS 7.1.1503 x86_64, 8 GB DDR4 2133 MHz, HD 500GB SATA (7200 RPM) e um Coprocessador Intel® Xeon Phi™ 3210A.

A biblioteca *Intel® OpenMP** permite a variação do numero de *threads* e tipo de afinidade por meio da alteração de variáveis de ambiente. O *Intel Xeon Phi* possui 3 tipos de afinidade de *threads*, sendo eles *balanced*, *compact* e *scatter*. Cada método distribui os *threads* de forma diferente, permitindo que haja a concentração de *threads* consecutivos em um mesmo núcleo (*compact*) ou espalhando-os entre os núcleos (*scatter*) [Reinders 2013].

Para compilação dos programas em todas as formas (*CPU*, *Offload* e *OpenCL*) foram adaptados os arquivos de *Makefile* de forma a utilizarem o compilador da Intel (ICC). Utilizando as configurações padrões do arquivo *./run* dos *benchmarks* foram realizados 10 testes para cada tipo de afinidade e número de *threads* por núcleo, utilizando um intervalo de confiança de 99.5%.



(a) Afinidade de threads no NW **(b) Afinidade de threads no HotSpot**



(c) Afinidade de threads no CFD

Figura 1: Teste de afinidade de threads

A Figura 1 ilustra o resultado da execução dos três algorítmos. É possível perceber que nas Figuras 1a e 1b, referentes respectivamente aos algoritmos *Needleman-Wunsch* (do domínio da bioinformática e que utiliza de conceitos de programação dinâmica) e *HotSpot* (do domínio das simulações físicas e que utiliza de *structured grids*), não houve boa escalabilidade do código devido ao número baixo de iterações no algoritmo, gerando resultados piores para quanto mais *threads* por núcleo, causados pelo tempo e recursos gastos com a criação e gerenciamento de *threads*.

Por outro lado, a Figura 1c, referente ao algoritmo *CFD Solver* (pertencente ao domínio da dinâmica de fluídos e que utiliza de *unstructured grids* para resolução de equações de Euler) possuiu alto grau de escalabilidade, resultando em uma relação de aumento de performance no aumento do número de *threads*.

Tendo como base os dados conseguidos por meio dos *benchmarks* utilizados e fazendo uso do mesmo número de iterações e dos mesmos arquivos para entrada de dados, foi gerada a Figura 2 com o eixo de Tempo (s) representado de forma logarítmica. Para a execução de códigos em *OpenCL* com o *Xeon Phi* foi necessária a alteração de todas as ocorrências de *CL_DEVICE_TYPE_GPU* nos códigos para *CL_DEVICE_TYPE_ACCELERATOR*, permitindo assim a utilização do coprocessador.

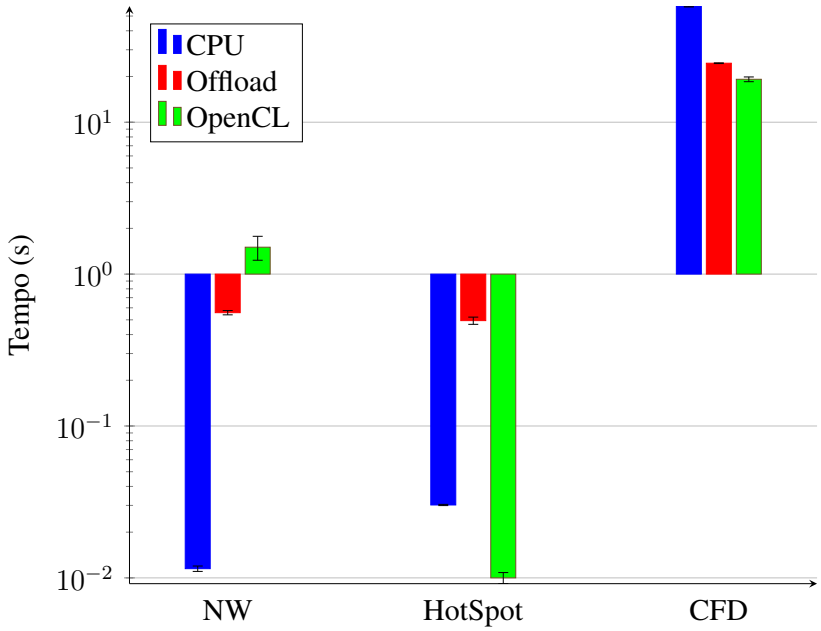


Figura 2: Melhores tempos de execução dos algoritmos

As barras azuis representam os tempos gastos para a execução dos algoritmos utilizando a versão *OpenMP*, porém executando apenas na CPU. As barras vermelhas utilizam a mesma metodologia descrita anteriormente, contudo fazem uso do coprocessador por meio de offload. As barras verdes representam a utilização do coprocessador por meio da linguagem *OpenCL*.

É possível notar que para algoritmos com baixa escalabilidade para número pequeno de iterações, como o *NW* e *HotSpot*, somente o uso da *CPU* se mostrou bastante favorável, obtendo bons resultados em ambos os casos, enquanto que houveram resulta-

dos instáveis para *OpenCL* e resultados ruins para *Offload*. Por outro lado, no algoritmo com maior escalabilidade (*CFD*) o uso de *OpenCL* e *offload* mostraram-se favoráveis, chegando a uma melhoria de desempenho de aproximadamente 300% e 237%, respectivamente.

Por fim, devido à má escalabilidade e facilidade em alterar o número de iterações (por meio da alteração do arquivo *./run*), foi realizada a comparação do tempo de execução por iterações no *HotSpot*, utilizando o número de *threads* e tipo de afinidade ótimo determinado empiricamente na Figura 1b, foi gerado o gráfico da Figura 3. O valor padrão de iterações para este algoritmo é de duas iterações (dessa forma, o primeiro ponto no gráfico possui os mesmos valores de tempo da Figura 1b para o *HotSpot*), aumentando exponencialmente esse valor por um fator de 10.

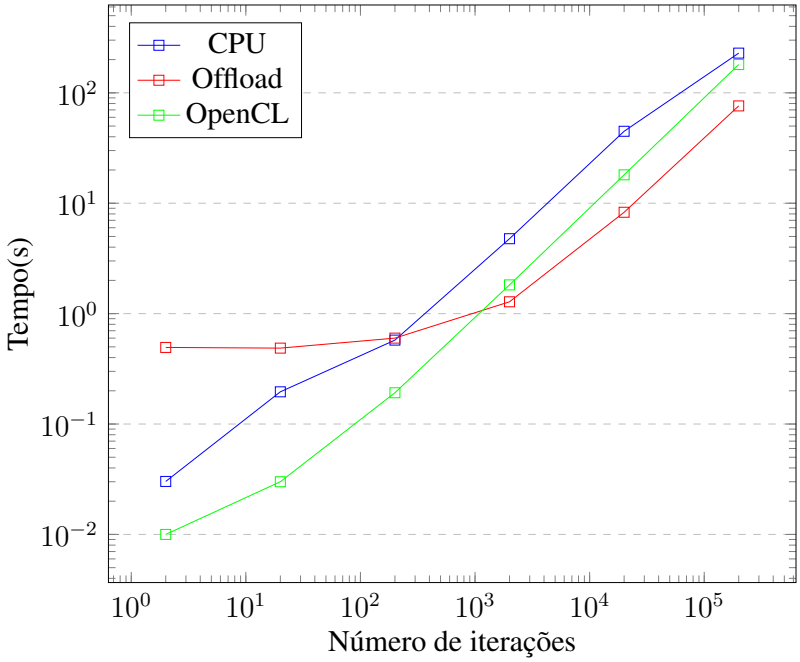


Figura 3: Tempo por iterações no HotSpot

É possível notar a melhoria de desempenho da técnica de *offload* utilizando *OpenMP* para um número alto de iterações, causando a redução do *overhead* derivado do uso da porta PCIe para a passagem de dados. Para valores altos de iterações o tempo de execução do método de *offload* torna-se linear, obtendo melhores resultados que o uso de *OpenCL* ou somente *CPU*.

5. Conclusão

Com base nos resultados é possível notar a viabilidade de utilizar códigos escritos em *OpenCL* com o *Xeon Phi*, necessitando de poucas alterações para adaptar o programa. Apesar de ser uma linguagem que utiliza de conceitos e recursos de baixo nível, os algoritmos testados em *OpenCL* não obtiveram resultados suficientemente bons quando comparados com *OpenMP*, principalmente dado um número de iterações alto, como visto na Figura 1b, de forma a justificar uma reescrita de códigos em C e C++ com uso de *OpenMP* para *OpenCL*.

Além disso, a utilização de três algoritmos de domínios diferentes se mostrou importante de forma a demonstrar que não há uma correlação direta entre maior número de *threads* e melhor desempenho, conforme visto nas Figuras 1a e 1b. Contudo, em algoritmos com alta escalabilidade (Figura 1c), o alto número de cores (e consequentemente *threads*) do *Xeon Phi* gera resultados ótimos, com aceleração de aproximadamente 300% em tempo de execução. Em todos os casos, a afinidade *balanced* se provou a melhor opção, obtendo na média o melhor desempenho.

Por fim, conforme exposto na Figura 1b, é possível notar que há um *overhead* causado pelo uso do barramento *PCIe* para transferência de dados para o coprocessador. O tempo de execução se mantém praticamente estável entre 2 e 2000 iterações, somente obtendo crescimento linear a partir de 20000 iterações, chegando assim à superação do tempo de *overhead*. Desta forma, pode-se concluir que o uso da técnica de *offload* utilizando o coprocessador só é viável para um número suficientemente grande de iterações, dando preferência para o uso de somente a *CPU* nos outros casos.

Os resultados obtidos neste artigo abrem espaço para a realização de futuros trabalhos buscando maior paralelização, escalabilidade e redução de *overhead* nos códigos em *OpenMP* que utilizam do *offload* para o coprocessador, assim como otimização de *kernels* e de parâmetros nos códigos em *OpenCL*.

Referências

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.

Memeti, S., Li, L., Pllana, S., Kolodziej, J., and Kessler, C. (2017). Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption. *arXiv preprint arXiv:1704.05316*.

Misra, G., Kurkure, N., Das, A., Valmiki, M., Das, S., and Gupta, A. (2013). Evaluation of rodinia codes on intel xeon phi. In *Intelligent Systems Modelling & Simulation (ISMS), 2013 4th International Conference on*, pages 415–419. IEEE.

Ramachandran, A., Vienne, J., Wijngaart, R. V. D., Koesterke, L., and Sharapov, I. (2013). Performance evaluation of nas parallel benchmarks on intel xeon phi. In *2013 42nd International Conference on Parallel Processing*, pages 736–743.

Reinders, J. J. . J. (2013). *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., Boston, MA, USA.

Explorando o sistema de memória heterogênea da arquitetura Intel *Knights Landing* (KNL)

Jefferson Fialho[1,2], Silvio Stanzani[1], Raphael Cóbe[1], Rogério Iope[1], Igor Freitas[3]

[1]Núcleo de Computação Científica – Universidade Estadual Paulista (UNESP)
São Paulo – SP – Brasil

[2]Graduando do Curso de Análise e Desenvolvimento na FATEC-SP
São Paulo – SP – Brasil

[3]Intel - Software and Services Group
São Paulo – SP – Brasil

{jfialho,silvio,rmcobe,rogerio}@ncc.unesp.br,igor.freitas@intel.com

Abstract. *This work presents the impact of using the heterogeneous memory system of the Intel Xeon Phi KNL architecture in different scenarios, in order to assist in a programming approach for high performance computing (HPC) on this architecture. A set of four applications with specific characteristics have been executed with different memory configurations, and a discussion of the results is presented.*

Resumo. *Este trabalho avalia o impacto do uso do sistema de memória heterogênea da arquitetura Intel Xeon Phi KNL em diferentes cenários, a fim de auxiliar em uma abordagem de programação para Computação de Alto Desempenho (HPC) nesta arquitetura. Foram executadas quatro aplicações com características próprias e diferentes configurações de memória, e uma discussão dos resultados é apresentada.*

1. Introdução

A arquitetura de processadores Intel Xeon Phi de segunda geração, também conhecida como Knights Landing (KNL), introduziu um novo nível de memória, o *Multi-Channel Dynamic Random Access Memory* (MCDRAM). Combinada com a memória principal *Dynamic Random Access Memory* (DRAM), a MCDRAM pode ser utilizada como último nível de cache (L3) e/ou como memória endereçável com grande largura de banda [Sodani et al., 2016]. Considerando esse novo cenário, um novo desafio se apresenta: como mapear adequadamente uma aplicação para as diferentes unidades de memória de modo a obter ganhos de desempenho [Li et al., 2016].

Motivado por uma análise anterior, com resultados publicados em [Coelho et al., 2017], este trabalho avalia o impacto do uso da MCDRAM em diferentes modos/configurações, a fim de procurar padrões que possam auxiliar o usuário a utilizar o sistema memória heterogênea do KNL.

O restante deste trabalho se organiza da seguinte forma: a Seção 2 apresenta uma visão geral em relação à arquitetura Intel KNL, descrevendo os modos de operação de processador. Na Seção 3 é apresentada a organização e as possíveis configurações do

sistema de memória heterogênea da arquitetura Intel KNL. A Seção 4 descreve o conjunto de aplicações utilizadas nos experimentos, o ambiente de testes, as cargas de entrada e apresenta os resultados obtidos. Por fim, a Seção 5 mostra nossa visão acerca dos resultados obtidos.

2. Organização do processador Intel KNL

Na arquitetura Intel KNL os núcleos de processamento são organizados em pares chamados *tiles* (Figura 1). Cada núcleo possui um cache de nível 1 (L1) e compartilha o cache de nível 2 (L2) com o outro núcleo que compõe o *tile.* Os *tiles* são interconectados utilizando um padrão do tipo *mesh*, e são organizados de acordo com os seguintes modos de agrupamento, identificados nesta arquitetura como *cluster modes* [Sodani, 2016]:

- *All-to-All*: os *tiles* não possuem subdivisão e os endereços de memória são distribuídos uniformemente entre eles. Normalmente esse modo é usado apenas para depuração [Sodani et al., 2016].
- Quadrante / Hemisfério: no modo quadrante, os *tiles* são divididos em 4 partes, cada uma local ao seu respectivo controlador de memória. O modo hemisfério funciona da mesma forma, com a diferença que é subdividido em 2 partes.
- SNC-4 / SNC-2: semelhante ao Quadrante / Hemisfério, os modos SNC-4 / SNC-2 também subdividem o total de *tiles* em 4 ou 2 partes. Nestes modos, cada subdivisão é vista pelo sistema operacional como um nó independente do tipo Non-Uniform Memory Access (NUMA).

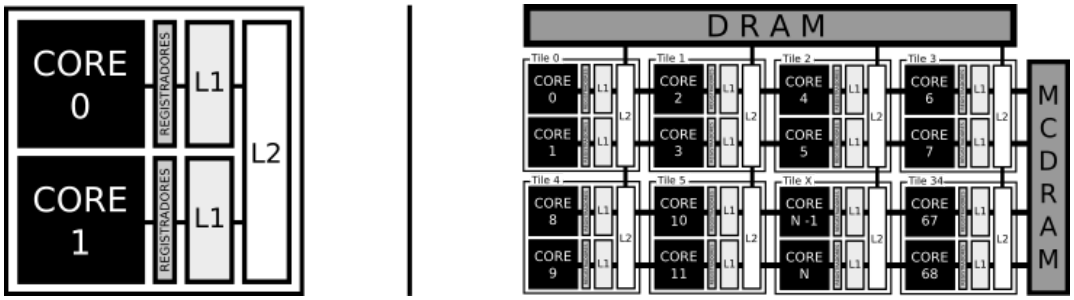


Figura 1: A esquerda, a estrutura de um *tile*. A direita, organização dos *tiles* combinado com o acesso ao sistema de memória

3. Organização do sistema de memória heterogênea

O subsistema de memória desenvolvido para o processador KNL consiste de uma memória principal DRAM e uma unidade de memória de alta banda adicional chamada MCDRAM, que pode ser configurada de 3 diferentes modos:

- *Cache*: neste modo, a MCDRAM é vista como o último nível de cache, sendo gerenciado somente pelo sistema operacional.
- *Flat*: a MCDRAM é vista como uma memória endereçável que pode ser utilizada por APIs ou utilizando a ferramenta *numactl*.
- *Hybrid*: através deste modo é possível usufruir parcialmente das duas configurações de memória anteriormente citadas. Este modo subdivide a MCDRAM em duas partes onde uma será configurada como memória endereçável (*flat*) utilizando 75% ou 50% do total da MCDRAM e a outra como

último nível de cache (*cache mode*) utilizando 50% ou 25% da capacidade da MCDRAM.

A Figura 2 exemplifica, de maneira lógica, a organização entre o sistema de memória e os *tiles.* Apesar da memória MCDRAM possuir uma capacidade bem menor que a DRAM, esta conta com largura de banda cerca de quatro vezes maior. Nesse sentido, o uso da MCDRAM pode trazer ganhos de desempenho, armazenando estruturas de dados que sejam acessadas por códigos que exijam grande uso de largura de banda [Li, 2016].

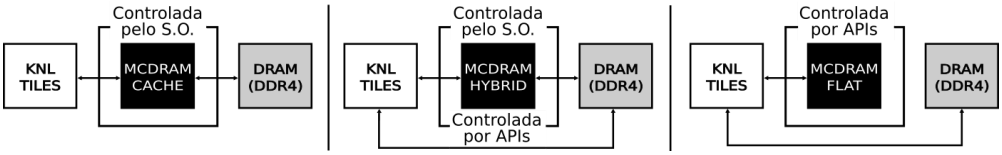


Figura 2: Modos de uso da MCDRAM

4. Avaliando o Uso da MCDRAM

Para avaliar o impacto do uso de diferentes modos da MCDRAM no desempenho da execução das aplicações, foi escolhido um conjunto de aplicações paralelizadas utilizando OpenMP [Chandra, 2001], que foram compiladas e mapeadas para DRAM e MCDRAM utilizando seus diversos modos. Os experimentos têm por objetivo levantar padrões para uso de modelos de sistema de memória heterogênea, de modo a obter ganhos de desempenho.

4.1. Ambiente de Teste

O ambiente de teste é composto por um servidor *multicore* Intel Xeon e um servidor *manycore* KNL, cada um com as seguintes configurações:

- Xeon (ambiente Xeon Core): dois processadores E5-2699v3 @ 2.3 GHz, com 18 núcleos físicos cada, duas *threads* por núcleo e 128GB de memória principal.
- KNL (Ambiente KNL): processador Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz, composto por 68 núcleos físicos, executando quatro *threads* por núcleo e 192 GB de memória principal. Os *tiles* estão configurados no modo Sub NUMA-4 e a MCDRAM foi utilizada utilizando os 3 modos (*cache, hybrid* e *flat)*. A modalidade SNC-4 foi adotada em razão do controle e segmentação de memória propiciado pelas configurações NUMA.

4.2. Carga de Entrada de Testes

Para essa avaliação foi utilizado o seguinte conjunto de aplicações:

- Transposição de Matrizes com duas matrizes (TranspMatAB) [Vladimirov, 2015]. Esta aplicação foi escolhida porque sua implementação apresenta grande fluxo de dados na memória (*Memory Bound*).
- Transposição de matrizes com somente uma matriz (*In-place)* [Gustavson, 2012]. Esta aplicação possui menor reescrita de cache em relação a TranspMatAB.
- Multiplicação de Matrizes em blocos/*tiles* (MultiMat) [Vladimirov, 2015]. Essa aplicação foi escolhida pois permitiu uma implementação computacionalmente

intensiva (*CPU bound*).

- *Option Price* (OptionPrice) [Meyerov, 2015]. Esta implementação foi escolhida por ser computacionalmente intensiva com necessidade de alocar uma grande quantidade de dados na memória.

Cada aplicação foi executada cem vezes, e calculada a média do tempo de execução, utilizando todos os núcleos de cada arquitetura. Foram desconsiderados os tempos de alocação e liberação da memória, pois o cerne dos experimentos é analisar o impacto da utilização de diferentes memórias durante as etapas de intenso processamento das aplicações. Com isso, espera-se identificar padrões que auxiliem na identificação das melhores estratégias de uso do sistema de memória heterogênea.

A Tabela 1 descreve os ambientes em que as aplicações foram executadas a fim de obter dados dos tempos de execução dos diferentes modos de configuração da MCDRAM, variando os tipos de memória em que os dados foram alocados.

Ambiente	KNL					XEON
Modo de Memória	FLAT		HYBRID		CACHE	PADRÃO
Tipo de memória	DRAM	MCDRAM	DRAM	MCDRAM	DRAM	DRAM

Tabela 1: Descrição dos cenários explorados em relação ao tipo de memória utilizado

Após a caracterização, cada aplicação foi executada no servidor Xeon, no servidor KNL usando DRAM e no mesmo servidor KNL explorando os modos MCDRAM.

Para explorar a MCDRAM, foi utilizada a biblioteca *hbwmalloc* do pacote Intel memkind[1]. Esta biblioteca tem por objetivo fornecer uma interface para utilização de memória de alta banda.

A Figura 3 mostra a alocação dinâmica da variável "a" na DRAM, feita utilizando a função malloc() e desalocada utilizando a função free(). No código à direita da Figura 3, a variável "a" é alocada na MCDRAM adicionando o prefixo *hbw_* as mesmas funções utilizadas para alocar memória na DRAM: utilizando as funções *hbw_malloc()* para alocar "a" e *hbw_free()* para desalocar.

```c
#include <stdlib.h>
int main () {
        double *a;
        a = (double*)malloc(sizeof(double));
        ...
        free(a);
        return 0;
}
```

```c
#include <hbwmalloc.h>
int main () {
        double *a;
        a = (double*)hbw_malloc(sizeof(double));
        ...
        hbw_free(a);
        return 0;
}
```

Figura 3: Comparação entre a utilização de MALLOC e HBW_MALLOC

4.3. Resultados dos experimentos

As aplicações TranspMatAB, *In-place* (Figura 4A) e OptionPrice (Figura 4B)

[1] Memkind Library: http://memkind.github.io/memkind/. Acesso em: 10 de jun. 2017.

tiveram desempenho superior no KNL comparado com o Xeon, e os seguintes aspectos foram observados:

- Com a API *hbwmalloc*, a utilização da MCDRAM permitiu que as aplicações movimentassem os dados utilizando maior banda, impactando positivamente no desempenho quando comparado ao KNL.
- Mesmo a MCDRAM possuindo 16GB de capacidade no modo *flat* e 8GB no modo *hybrid*, quando a aplicação necessitou de mais memória, a API *hbwmalloc* proveu automaticamente a utilização de DRAM para alocar o contingente de dados.
- Mesmo evitando a reescrita dos dados na cache, a aplicação In-Place obteve um desempenho inferior a TranspMatAB no testes realizados na arquitetura KNL.

No que diz respeito à aplicação MultiMat (Figura 4B), o uso da memória cache foi mais ineficiente no KNL do que no Xeon, o que provocou muitos atrasos.

Em razão da diferente escala, os gráficos na Figura 4 foram separados para melhor visualização. O rótulo das legendas apresentam o modo de configuração da MCDRAM seguido do tipo de memória onde os dados das aplicações foram alocados.

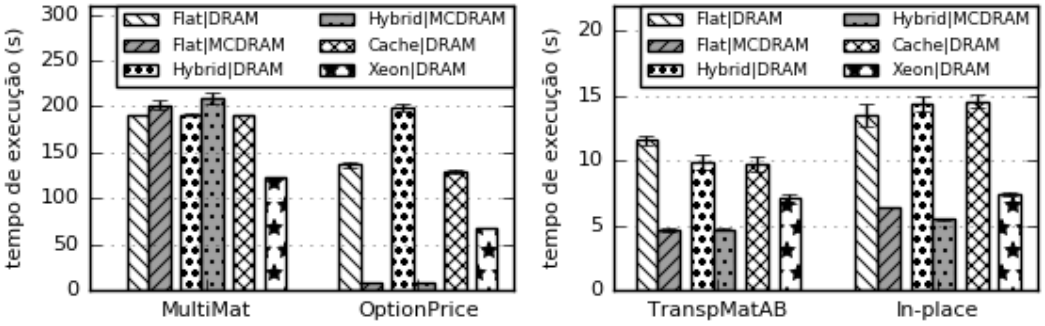


Figura 4A: Desempenho TranspMatAB e In-place explorando diferentes modos de memória
Figura 4B: Desempenho MultiMat e OptionPrice explorando diferentes modos de memória

A fim de investigar se a combinação de alta velocidade da DRAM com a alta banda da MCDRAM pode melhorar o desempenho, a aplicação TranspMatAB foi executada no ambiente KNL nos modos *flat* e *Hybrid,* variando a alocação dinâmica das matrizes A e B (respectivamente entrada e saída) entre os diferentes tipos de memória, conforme descrito na Tabela 2. A coluna "A (DRAM) e B (MCDRAM)" apresenta os dados da execução alocando a matriz de entrada "A" em memória DRAM e a matriz de saída (matriz resultado) "B" na MCDRAM. A coluna "A (MCDRAM) e B (DRAM)" apresenta as alocações das matrizes de entrada e saída em MCDRAM e DRAM respectivamente. O desvio padrão máximo observado nas medições inseridas na tabela 2 foi de 4,68%.

MCDRAM\Local Matriz	A (DRAM) e B (MCDRAM)	A (MCDRAM) e B (DRAM)
Flat Mode	5,508s	13,526s
Hybrid Mode	3,833s	13,724s

Tabela 2: Tempo de execução de TranspMatAB variando a localidade das matrizes de entrada e saída nos modos *Flat* e *Hybrid*

Com relação à variação das matrizes de entrada e saída de TranspMatAB, os seguintes aspectos foram observados:

- O teste no cenário *Hybrid* com a matriz de entrada alocada na DRAM e a matriz de saída alocada na MCDRAM obteve o melhor desempenho observado de todos os cenários em que TranspMatAB foi avaliado.
- Os testes com a matriz de entrada alocada na MCDRAM e a matriz de saída alocada na DRAM obtiveram os piores desempenhos de avaliação de TranspMatAB na arquitetura KNL.

5. Conclusão

Este artigo apresentou uma avaliação do sistema de memória heterogênea do processador Intel KNL. Para isso foram escolhidas quatro aplicações executadas em processadores Intel Xeon e KNL, visando a identificação de padrões que pudessem auxiliar numa melhor utilização de MCDRAM. Com base nos cenários explorados, notou-se que em três dos quatro cenários avaliados a utilização da MCDRAM aumentou a vazão de dados da memória, possibilitando um desempenho superior ao apresentado pelo processador Intel Xeon.

Em trabalhos futuros pretende-se explorar os modos *cache* e *hybrid,* utilizando diferentes aplicações variando seus parâmetros e forma, a fim de se obter um *cache-fit* adequado (ajuste ótimo dos dados da aplicação em relação ao tamanho do cache do ambiente de execução).

Referências

Li S., Raman, K. and Sasanka, R., "Enhancing application performance using heterogeneous memory architectures on a many-core platform," *2016 Conference on High Performance Computing & Simulation (HPCS)*, 2016.

Sodani, A. *et al*., "Knights Landing: Second-Generation Intel Xeon Phi Product," em *IEEE Micro*, vol. 36, 2016.

Chandra, R. “Parallel programming in OpenMP”, Morgan Kaufmann, 2001.

Vladimirov, A., Capítulo 24, “Profiling-Guided Optimization, In High Performance Parallelism Pearls”, editado por Reinders, J. e Jeffers, J., Morgan Kaufmann, 2015.

Coelho, J., Stanzani, S., Cóbe, R., Iope, R. and Freitas, I., "Avaliação de desempenho do sistema de memória heterogênea da arquitetura Intel Knights Landing (KNL)", em Escola Regional de Alto Desempenho - São Paulo (ERAD-SP), 2017.

Gustavson, F., Karlsson, L. and Kågström, B., "Parallel and cache-efficient in-place matrix storage format conversion.", ACM TOMS, 2012.

Meyerov, I., Sysoyev, A., Astafiev, N. and Burylov, I., Capítulo 19 - “Performance Optimization of Black-Scholes Pricing”, In High Performance Parallelism Pearls, editado por Reinders, J. e Jeffers, J., Morgan Kaufmann, 2015.

Impacto do Emprego da Afinidade de Processador em uma Arquitetura com Tecnologia *Clustered MultiThreading*

Carlos Alexandre de Almeida Pires[1], **Marcelo Lobosco**[1]

[1]Grupo de Educação Tutorial do Curso de Engenharia Computacional
Universidade Federal de Juiz de Fora (UFJF) – Juiz de Fora, MG – Brasil

carlos.alexandre@engenharia.ufjf.br, marcelo.lobosco@ice.ufjf.br

Abstract. *This paper evaluates the impact of affinity scheduling on the performance of parallel applications executing on multicore processors with the Clustered MultiThreading technology. The results have shown gains up to 1.8 times when affinity scheduling was used.*

Resumo. *Este trabalho investiga o impacto da afinidade de processador no desempenho de aplicações paralelas executadas em ambientes com a tecnologia Clustered MultiThreading. Os resultados mostram que o uso de afinidade pode levar a ganhos de desempenho de até 1,8 vezes.*

1. Introdução

A microarquitetura Bulldozer, desenvolvida pela AMD para o mercado de computadores pessoais e servidores, foi lançada em 2011 e representou uma grande mudança em relação à microarquitetura anterior, K10, por ter sido desenvolvida do zero. De fato, se considerado que K10 é baseada em outra microarquitetura, K8, pode-se considerar que Bulldozer representou a primeira grande alteração de projeto nas microarquiteturas dos processadores AMD desde 2003. Uma das principais características da microarquitetura Bulldozer é a sua construção modular. Cada módulo é composto, dentre outros, por duas unidades de inteiros, uma unidade ponto-flutuante (*Floating Point Unit* ou FPU), uma *cache* L1 de instruções, duas *caches* L1 de dados (uma para cada unidade de inteiro, mas ambas compartilhadas com a FPU), uma *cache* L2 (unificada para dados e instruções), uma unidade de busca, uma unidade de decodificação e uma unidade de despacho de instruções. A partir de um agrupamento de módulos são criados os processadores, adicionando-se a eles a *cache* L3, que é compartilhada por todos os módulos. A Figura 1 esboça os principais componentes do *pipeline* da microarquitetura Bulldozer, as unidades de inteiros e ponto-flutuante. Um processador com 16 núcleos, como o ilustrado na figura, será composto por oito destes módulos, já que o sistema operacional reconhece cada módulo como dois núcleos lógicos. Do ponto de vista do desempenho para aplicações *multithreading*, cada módulo pode comportar-se de modo distinto, dependendo das características da aplicação. Se a aplicação paralela executa majoritariamente instruções com tipos de dados inteiros, o módulo parecerá para a aplicação, de fato, como um processador com dois núcleos, visto que ambas as unidade de inteiros podem ser empregadas para executar as instruções. Contudo, caso a aplicação paralela execute majoritariamente instruções com tipos de dados ponto-flutuante, cada módulo parecerá com um único núcleo, visto que existe apenas uma FPU disponível para uso. Este esquema também é conhecido como *Clustered MultiThreading* (CMT), onde algumas partes do processador são compartilhadas por duas

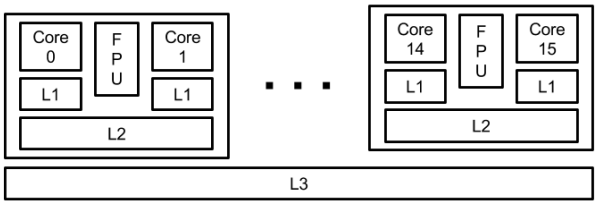


Figura 1. Módulos da microarquitetura Bulldozer, com destaque para as unidades de inteiros, ponto-flutuante e caches.

threads, como é o caso da FPU, enquanto outras são de uso privativo para cada *thread*, como a unidade de inteiros.

Neste tipo de processador, mesmo em um cenário onde existam FPUs disponíveis em número suficiente para atender uma determinada aplicação paralela com demandas para processamento ponto-flutuante, podemos ter o desempenho comprometido por conta das opções de escalonamento efetuadas pelo sistema operacional (SO). Esse cenário pode ocorrer, por exemplo, quando o SO decide escalonar duas *threads* da aplicação paralela no mesmo módulo, ainda que existam outros módulos disponíveis para uso. A hipótese deste trabalho é que uma das formas de resolver esse problema, para aplicações que demandem FPUs em número menor ou igual ao número de módulos disponíveis no sistema, é empregar a técnica conhecida como afinidade de processador (AP) (*Processor Affinity*) [Squillante and Lazowska 1993] no escalonamento de processos. Com afinidade de processador habilitada, determina-se um núcleo preferencial para execução de uma *thread*. Neste caso, pode-se orientar o SO a escalonar as *threads*, na medida do possível, em módulos diferentes. O objetivo do trabalho é avaliar a validade da hipótese, utilizando para isso o seguinte método: executa-se um *benchmark* de computação paralela, com e sem o emprego de AP, e medem-se os tempos de computação, comparando-os em seguida. Esse trabalho está organizado como descrito a seguir. A Seção 2 descreve brevemente trabalhos correlatos. A Seção 3 apresenta com mais detalhes o método empregado para avaliar o impacto da AP, e a Seção 4 apresenta os resultados obtidos. Por fim, a seção 5 apresenta as conclusões e planos para trabalhos futuros.

2. Trabalhos Relacionados

Vários trabalhos anteriores avaliaram, sob diversos aspectos, o impacto da AP no desempenho de aplicações paralelas. Um dos trabalhos avaliou diretamente o impacto da AP no desempenho de aplicações paralelas utilizando a arquitetura Intel [Pires and Lobosco 2016]. Seus resultados mostraram que o uso da AP pode impactar positivamente o desempenho nesta arquitetura, desde que haja definição criteriosa da ordem em que os núcleos destes processadores sejam utilizados. Deve-se destacar que se propõe, nesse artigo, avaliar o desempenho em um processador com características distintas, visto que no processador Intel as FPUs não são compartilhadas entre os núcleos. Outro trabalho avaliou o impacto de diversas políticas de escalonamento no desempenho de aplicações paralelas [Gupta et al. 1991], sendo uma delas a política baseada na AP. Os resultados do artigo mostraram que a AP foi responsável pela melhoria das taxas de acerto da *cache* e pelo aumento da utilização do processador. Como o estudo foi reali-

zado no início da década de 1990, a arquitetura utilizada nos testes difere da utilizada neste estudo. Dois estudos mais recentes [Ribeiro et al. 2009, Pousa Ribeiro et al. 2011], realizados em uma arquitetura NUMA, apresentaram o impacto da AP em um conjunto de aplicações científicas. A microarquitetura Bulldozer, estudada nesse artigo, é uma arquitetura UMA, com a particularidade de empregar a tecnologia CMT. Outro estudo mostrou que o uso da AP pode impactar positivamente o desempenho das aplicações [Squillante and Lazowska 1993]. Esse artigo fez um estudo teórico, sem realizar uma análise voltada para a execução de aplicações paralelas, nem baseada em máquinas reais, como a feita neste trabalho.

3. Método

Para avaliar o impacto do uso da AP no desempenho de aplicações *multithreaded* em uma arquitetura que empregue a tecnologia CMT foi escolhido o *benchmark* NPB (*NAS Parallel Benchmarks*) em sua versão memória compartilhada implementada em OpenMP [Jin et al. 1999]. Foram então coletados os tempos de execução das aplicações paralelas com e sem o uso de AP no escalonamento das *threads* pelo SO. As aplicações do *benchmark* foram executadas com até 64 *threads* e diferentes tamanhos de entrada, conforme detalhado nesta seção.

3.1. NPB 3.3

Foi utilizada nos testes deste trabalho a versão 3.3 do NPB, um conjunto de *benchmarks* criado pela Divisão de Supercomputação Avançada da NASA (*National Aeronautic and Space Administration*). Os *benchmarks* são compostos por *kernels* e pseudo-aplicações. *Kernels* são trechos de códigos que podem ser encontrados em diversas aplicações. Neste artigo, escolhemos três *kernels* e duas pseudo-aplicações para os testes. Os *kernels* escolhidos foram EP, CG, e FT. As pseudo-aplicações utilizadas foram BT e SP. Os *kernels* e pseudo-aplicações foram implementados em Fortran e utilizam tipos de dados ponto-flutuante de precisão dupla. Todos os *benchmarks* possuem um parâmetro, titulado de classe, que define o tamanho dos problemas a serem resolvidos. São 7 classes de problema: S, W, A, B, C, D e E, sendo S a classe com menor demanda de memória e E a classe com maior demanda. Para os testes realizados nesse trabalho, foram utilizadas duas classes intermediárias de problemas, A e C. A classe C foi escolhida por ser a maior classe de problemas em que todos os *benchmarks* escolhidos executam com a memória disponível no ambiente experimental. A classe A foi utilizada para permitir inferir os impactos da AP na *cache*.

Block Tri-diagonal solver (BT) implementa um caso de dinâmica de fluidos computacional (CFD - *Computational Fluid Dynamics*) utilizando as equações de Navier-Stokes em três dimensões. É empregado na solução o método das diferenças finitas com aproximações pelo método ADI (*Alternating Direction Implicit*). *Conjugate Gradient* (CG) implementa o método do gradiente conjugado para calcular uma aproximação para o menor autovalor em uma matriz esparsa, não estruturada e com valores gerados aleatoriamente. Possui acesso irregular à memória. *Embarrassingly Parallel* (EP) gera pares de valores aleatórios seguindo uma distribuição gaussiana a partir do esquema polar de Marsaglia. Este *kernel* realiza poucas operações de comunicação ao longo de sua execução. *Fourier Transform* (FT) implementa o algoritmo da transformada rápida de Fourier em três dimensões. O *kernel* resolve três transformadas separadamente, uma para

cada dimensão. Os acessos à memória são bem distribuídos e regulares. Por fim, *Scalar Penta-diagonal solver* (SP) é uma aplicação semelhante a BT, porém usando um método de aproximação distinto, o método de Beam-Warming. A aplicação acessa a memória de modo regular.

3.2. Ambiente Experimental

Os testes foram feitos em um nó de um *cluster*, cujo acesso exclusivo é feito através da submissão de *jobs* a uma fila de tarefas gerenciada pelo OGE (*Oracle Grid Engine*). O nó escolhido para execução possui quatro processadores AMD Opteron 6272 (microarquitetura Bulldozer), cada um com dezesseis núcleos, totalizando assim 64 núcleos lógicos de processamento (32 módulos). Cada núcleo possui 16KB de *cache* L1 de dados. Dois núcleos compartilham 64 KB de *cache* L1 de instruções, 2MB de *cache* L2 de instruções e dados. Cada processador possui uma *cache* L3 de 16MB, usada para armazenar tanto dados como instruções, e compartilhada entre os dezesseis núcleos de cada processador. A máquina executa o SO Linux com *kernel* na versão 2.6.32. O compilador gFortran na sua versão 6.1.0 foi usado para compilar os programas. Para habilitar o uso da AP, foi utilizada a variável de ambiente do OpenMP **GOMP_CPU_AFFINITY**[1], de modo que as *threads* fossem distribuídas pelo mapa da topologia do sistema. As *threads* foram alternadas entre módulos diferentes de processadores diferentes. Apenas quando não é mais possível alocar uma única *thread* por bloco, são alocadas duas *threads* por módulo, o que ocorre apenas em aplicações com mais de 32 *threads*. O tempo de execução foi coletado pelo aplicativo **time**, disponível no SO e que possui precisão de 0,001 segundos. Em todas as configurações, todas as aplicações executaram no nó de modo exclusivo, sem concorrência com outros processos de outros usuários. Todas as aplicações foram executadas no mínimo 5 vezes, até que o desvio-padrão fosse menor do que 5%. Os valores reportados na seção de resultados representam a média aritmética simples para os tempos de execução coletados. Para comparar os impactos do uso da AP no desempenho das aplicações que executem em um ambiente com a tecnologia CMT, foi utilizada a fórmula clássica do desempenho [Patterson and Hennessy 2014], em que o desempenho é definido como o inverso do tempo de computação.

4. Resultados

As Tabelas 1 e 2 apresentam, respectivamente para as classes A e C, os tempos médios de execução para cada *benchmark*. Nesta tabela são apresentados os tempos com e sem o uso da AP para configurações com 1, 2, 4, 8, 16, 32 e 64 *threads*. Os resultados com uma única *thread* indicam, como seria de se esperar, tempos equivalentes para execuções com e sem o uso de AP. Deve-se ressaltar que as pequenas variações de tempo são menores que o desvio-padrão. Como regra geral para todas as aplicações e classes de problemas, pontuadas algumas exceções, observa-se que para configurações com número de *threads* entre 1 e 16 não existem grandes alterações no tempo de computação com o uso de AP. Já os melhores resultados com o uso da AP foram obtidos para a configurações com 32 e 64

[1]**GOMP_CPU_AFFINITY="0 16 32 48 8 24 40 56 2 18 34 50 10 26 42 58 4 20 36 52 12 28 44 60 6 22 38 54 14 30 46 62 1 17 33 49 9 25 41 57 3 19 35 51 11 27 43 59 5 21 37 53 13 29 45 61 7 23 39 55 15 31 47 63"**. Equivale ao uso da estratégia *compact* para uso com a variável de ambiente **KMP_AFFINITY**, com *flags granularity=fine*, *scatter*, disponível apenas no compilador Intel (este trabalho usou o compilador gFortran). Esta configuração foi empregada em todos os experimentos com afinidade habilitada.

threads. Resultados próximos ou iguais para os tempos com e sem o uso de AP para 4, 8 e 16 *threads* seriam esperados, visto que a probabilidade de duas *threads* serem alocadas em um mesmo módulo é baixa. Contudo, essa probabilidade aumenta significativamente para 32 *threads*, o que faz com que o uso da AP seja mais efetivo para melhorar o desempenho das aplicações, visto que a AP elimina a chance de duas *threads* compartilharem a FPU ao não alocá-las no mesmo módulo.

Para 64 *threads* e configurações entre 2 e 16 *threads* que tiveram ganhos de desempenho com o uso de AP, o melhor desempenho é explicado pelo impacto da *cache*: por fixar as *threads* sempre nos mesmos núcleos, temos a garantia que eliminaremos falhas a frio decorrentes de escalonamento da *thread* em um núcleo distinto ao que se encontrava na fatia de tempo anterior, o que pode ocorrer quando não se usa a AP. Essa hipótese é reforçada quando se observa o comportamento das aplicações com tamanhos distintos de classes. Com uma classe que demanda menor uso de memória (classe A), o uso da AP leva a melhores desempenhos do que para a classe com maior demanda por memória (classe C). Se uma parte significativa do conjunto de dados utilizados pela aplicação cabe na *cache* L1, o que provavelmente ocorre para aplicações regulares com boa localidade temporal/espacial e baixa demanda de memória, o uso da AP torna-se muito atrativo, ao reduzir falhas a frio. Por outro lado, uma situação em que não haveria vantagem no uso da AP, mesmo para configurações com 32 *threads*, é quando as taxas de falha de *cache* são altas. Neste cenário, pode não fazer tanta diferença usar sempre os mesmos núcleos para executar as *threads*, visto que a memória principal sempre deverá ser acessada para recuperar os dados necessários para a execução da aplicação. Apesar dos indícios fortes obtidos pelos resultados, tal hipótese ainda precisa ser confirmada pela instrumentação das aplicações para coletar o percentual de falhas de *cache* com e sem o uso da AP.

Tabela 1. Tempos médios de execução (s) para cada *benchmark* da classe A.

Benchmark	Uso de AP	Número de *Threads*						
		1	2	4	8	16	32	64
BT	Não	92,11	47,72	24,06	12,11	6,60	5,11	4,77
	Sim	90,56	46,79	23,72	12,10	6,55	4,79	4,52
Desempenho:		1,02	1,02	1,01	1,00	1,01	1,07	1,06
CG	Não	3,29	1,97	0,80	0,43	0,24	0,19	0,41
	Sim	3,29	1,83	0,80	0,43	0,24	0,18	0,22
Desempenho:		1,00	1,08	1,01	1,00	1,00	1,05	1,83
EP	Não	36,86	18,47	9,25	4,63	2,34	1,30	1,36
	Sim	36,84	18,48	9,25	4,64	2,33	1,21	1,32
Desempenho:		1,00	1,00	1,00	1,00	1,00	1,08	1,03
FT	Não	7,82	3,93	2,04	1,03	0,53	0,40	0,35
	Sim	7,89	3,94	1,97	1,02	0,53	0,33	0,31
Desempenho:		0,99	1,00	1,03	1,01	1,00	1,21	1,13
SP	Não	59,87	34,98	16,39	8,39	5,42	6,35	14,83
	Sim	60,00	32,37	15,91	8,34	5,31	5,52	11,88
Desempenho:		1,00	1,08	1,03	1,01	1,02	1,15	1,25

5. Conclusão

Este trabalho apresentou uma análise do impacto do escalonamento baseado em AP no desempenho de um conjunto de aplicações paralelas executadas em um ambiente de memória compartilhada distribuída que implementa a tecnologia CMT. A análise mostrou que todas as cinco aplicações e *kernels* avaliados tiveram ganhos de desempenho de até 1,4 vezes quando executadas com a AP habilitada em uma configuração com 32 *threads*. Houve também ganho na configuração com 64 *threads*, de até 1,8 vezes, mas

Tabela 2. Tempos médios de execução (s) para cada *benchmark* da classe C.

Benchmark	Uso de AP	Número de *Threads*						
		1	2	4	8	16	32	64
BT	Não	1550,21	782,95	395,18	199,32	106,83	81,37	68,74
	Sim	1547,48	783,80	393,26	199,54	107,35	65,67	65,69
Desempenho:		1,00	1,00	1,00	1,00	1,00	1,24	1,05
CG	Não	242,91	124,65	63,41	34,43	21,82	25,01	33,51
	Sim	242,78	123,43	62,91	33,60	21,39	23,36	32,15
Desempenho:		1,00	1,01	1,01	1,02	1,02	1,07	1,04
EP	Não	589,86	294,92	147,94	73,75	36,88	19,73	19,35
	Sim	588,92	294,89	147,39	73,79	36,88	18,58	19,50
Desempenho:		1,00	1,00	1,00	1,00	1,00	1,06	0,99
FT	Não	362,00	181,64	93,81	45,91	24,87	19,26	12,52
	Sim	353,39	180,24	90,70	46,12	24,21	13,94	11,51
Desempenho:		1,02	1,01	1,03	1,00	1,03	1,38	1,09
SP	Não	1152,96	595,20	311,86	192,74	291,82	366,87	455,56
	Sim	1155,62	593,35	303,75	169,31	205,41	291,19	421,29
Desempenho:		1,00	1,00	1,03	1,14	1,42	1,26	1,08

provavelmente decorrente da redução do número de falhas de cache a frio. Não foram observados ganhos expressivos de desempenho com as configurações com 1, 2, 4, 8 e 16 *threads*, com exceção de SP, que provavelmente também se beneficiou da redução das taxas de falhas a frio. Como trabalhos futuros, os códigos das implementações serão instrumentados para coletar o percentual de falhas de *cache*, de modo a confirmar ou refutar essa hipótese.

Referências

Gupta, A., Tucker, A., and Urushibara, S. (1991). The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. *SIGMETRICS Perform. Eval. Rev.*, 19(1):120–132.

Jin, H., Jin, H., Frumkin, M., Frumkin, M., Yan, J., and Yan, J. (1999). The openmp implementation of nas parallel benchmarks and its performance. Technical report, NASA.

Patterson, D. and Hennessy, J. L. (2014). *Arquitetura de Computadores: uma abordagem quantitativa*. Elsevier Brasil, 5 edition.

Pires, C. A. A. and Lobosco, M. (2016). Avaliação do impacto da afinidade de processador no desempenho de aplicações paralelas executadas em ambientes de memória compartilhada. In *Workshop de Iniciação Científica do XVII Simpósio de Sistemas Computacionais de Alto Desempenho*, pages 50–55.

Pousa Ribeiro, C., Castro, M., Méhaut, J.-F., and Carissimi, A. (2011). *Improving Memory Affinity of Geophysics Applications on NUMA Platforms Using Minas*, pages 279–292. Springer Berlin Heidelberg, Berlin, Heidelberg.

Ribeiro, C. P., Mehaut, J. F., Carissimi, A., Castro, M., and Fernandes, L. G. (2009). Memory affinity for hierarchical shared memory multiprocessors. In *2009 21st International Symposium on Computer Architecture and High Performance Computing*, pages 59–66.

Squillante, M. S. and Lazowska, E. D. (1993). Using processor-cache affinity information in shared-memory multiprocessor scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 4(2):131–143.

Aplicação CHARM++ na Paralelização da Simulação do Movimento da Água no Solo

Pablo J. Pavan[1], Edson L. Padoin[1,2], Philippe O. A. Navaux[2]

[1]Universidade Reg. do Noroeste do Estado do Rio G. do Sul (UNIJUI) - Ijuí - RS - Brasil

[2]Universidade Federal do Rio Grande do Sul (UFRGS) - Porto Alegre - RS - Brasil

{pablo.pavan,padoin}@unijui.edu.br, navaux@inf.ufrgs.br

Abstract. *High Performance Computing (HPC) has a very important role to solve problems that require high computational power. In this way, new tools are created so that applications written in parallel can be more efficient with load balancing and asynchronous message The present work makes a study on the model of Parallel Programming* CHARM++*, as well as the development of an application that simulates the water movement of soils without load balancing. The results showed that the use of parallelism through the* CHARM++ *model led to a reduction in the execution time of up to 11 times compared to the sequential algorithm, without task migration.*

Resumo. *A Computação de Alto Desempenho (HPC) tem um papel muito importante em problemas que exigem alto poder computacional. Desta forma, novas ferramentas são criadas para que aplicações paralelas possam ser mais eficientes através de balanceamento de carga e troca de mensagens. O presente trabalho faz um estudo sobre a aplicação do modelo de Programação Paralela* CHARM++ *sobre uma aplicação que simula a movimentação da água no solo sem a utilização de balanceamento de carga. Os resultados mostraram que o uso deste modelo levou a uma redução no tempo de execução e consumo de energia de até 11 vezes se comparado com o algoritmo sequencial, sem migração de tarefas.*

1. Introdução

Dado o aumento de aplicações de simulações de fenômenos reais, tem-se um crescimento significativo por sistemas paralelos que atendam a demanda computacional destas aplicações. Assim, novos sistemas para a computações de alto desempenho vem sendo desenvolvidos para suprir tal demanda. Tais sistemas possuem milhares de processadores multicores proporcionando que aplicações sejam divididas em tarefas menores e executadas paralelamente nos núcleos dos processadores.

Assim, surge uma nova demanda de se implementar aplicações que utilizem de maneira eficiente a capacidade computacional paralela destes sistemas, uma vez que as aplicações possuem tarefas com cargas computacionais de diferentes tamanhos. Nesse contexto, novas tecnologias de programação, mapeamento e migração de tarefas vem sendo desenvolvidas, dentre elas, destaca-se a linguagem de programação CHARM++.

Baseada na linguagem C++, CHARM++ oferece um ambiente com suporte a multiplataformas que permite que programas sejam executados tanto em ambientes de memória compartilhada quando com distribuída. Esta ferramenta utiliza uma técnica que consiste em dividir um problema em diversas tarefas menores que são migráveis, e que podem ser

executados em vários núcleos de processamento. Estas tarefas migráveis são chamados de chares [Pilla e Meneses 2015, Arruda e Padoin 2015].

Por outro lado, simulações reais demandam grande poder de processamento. Um exemplo são as aplicações de áreas com a da agricultura, onde estimar o movimento da água no solo é de suma importância. Isto porque, é a partir do movimento da água é que os nutrientes vão ser absorvidos pelas raízes das plantas. Nesse contexto, o desenvolvimento de uma aplicação que possa simular a movimentação de forma mais rápida e precisa possibilita pesquisas na área, principalmente para sistemas de irrigação por gotejamento.

Assim, almejando utilizar todo recurso computacional disponível nos sistemas paralelos, nós aplicamos CHARM++ na paralelização uma aplicação de simulação do movimento da água no solo. Neste trabalho apresentamos a proposta e realizamos uma análise de ganhos de desempenho alcançados com a paralelização da aplicação utilizando o modelo de programação paralela CHARM++.

O restante do trabalho está organizado da seguinte forma. A Seção 2 discute trabalhos relacionados. A Seção 3 descreve o estudo de caso. A metodologia é apresentada na Seção 4. Resultados são discutidos na Seção 5, seguidos das conclusões e perspectivas de trabalhos futuros.

2. Trabalhos Relacionados

Na literatura diversos trabalhos apresentam estudos sobre a simulação movimentação da água no solo. Borges apresenta uma análise do comportamento da água do solo saturado e não saturado utilizando a equação de Richards [Borges et al. 2005]. Complementando o referido trabalho, em [Borges e Padoin 2006] uma nova implementação do algoritmo foi realizada utilizando o método de diferencias. Porém devido ao elevado tempo de processamento, utilizou-se somente uma simulação com matriz de ordem 21.

Outras abordagens tem empregado balanceamento de carga para aumentar a eficiência de utilização dos sistemas paralelos. Kale propõem a utilização de CHARM++ para balancear as cargas das tarefas entre os processadores [Kalé et al. 1998]. Neste trabalho, os testes foram realizados com um algoritmo paralelo de dinâmica molecular em CHARM++. Além disso, à medida que a simulação evolui o movimento de átomos causa mudanças nas distribuições de carga. Assim o trabalho busca estratégias de balanceamento de carga e analisa o impacto que isso causa no desempenho da aplicação.

Similar ao trabalho de Kale, Tesser utiliza tal abordagem na paralelização de um simulador de ondas sísmicas utilizando o modelo de programação AMPI [Tesser et al. 2014]. Os resultados alcançados apresentam ganhos de desempenho de até $23,85\%$ quando comparado com a implementação em MPI. Nesse contexto, percebe-se que os balanceadores de carga podem ser utilizados para redução do desequilíbrio de carga durante a execução de aplicações paralelas.

Outras bibliotecas de paralelização são amplamente utilizadas quando não é realizado o balanceamento de carga. Kang compara OpenMP, MPI e MapReduce utilizando um benchamrk [Kang et al. 2015]. Os resultados de Kang, indicam que quando o problema pode ser resolvido utilizando recursos de somente uma máquina, OpenMP é aconselhado já que não suporta memória distribuída. Para problemas que utilizam grande poder computacional, é aconselhado o uso do MPI já que este suporta memória distribuída. A utilização de MapReduce é aconselhada quando o número de dados utilizado seja grande, porém com poucas

iterações.

Satarić apresenta a implementação de um programa híbrido utilizando OpenMP e MPI para resolver a equação Gross-Pitaevskii [Satarić et al. 2016]. Para esta implementação é utilizado MPI para distribuir as tarefas entre os nós de processamento e em cada nó as tarefas são paralelizadas utilizando OpenMP. Os resultados alcançados por Satarić, apresentam um *speedup* e uma escalabilidade quase que linear para o problema abordado.

Diferente destes trabalhos relacionados, nossa proposta é avaliar a utilização de CHARM++ na paralelização de aplicações reais sem a utilização de balanceamento de carga.

3. Caso de Estudo: Sistema de Movimentação da Água nos Solos

Na agricultura é grande a necessidade de determinação da quantidade ótima de água nos sistemas de irrigação. Neste contexto, o problema do movimento da água no solo foi modelado pela equação de Richards, cuja a dedução é encontrada nos trabalhos de [Padoin et al. 2011]. A absorção da água depende do tempo e é limitada pela quantidade disponível de água no solo. Deste modo, foi proposta a Equação 1 que considera a variação das condições climáticas cíclicas normais do período diário. O parâmetro β é uma constante de proporcionalidade e foi determinado por ajuste (Problema Inverso) em função dos dados experimentais [Padoin et al. 2011].

$$S = \beta(\Theta - \Theta_r)^{b(t-c)} \tag{1}$$

onde, S é taxa de absorção de água pela raiz (cm^3/h), β é a constante de proporcionalidade (cm^3/h), b é o parâmetro obtido experimentalmente ($1/h$), c é o instante de tempo da máxima absorção diária (h), t é o tempo (h), Θ é o teor de água no ápice (adimensional) e Θ_r é o teor de água no residual (adimensional).

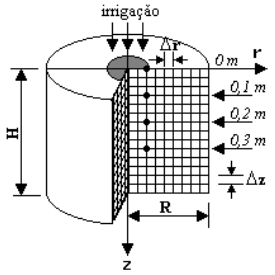


Figura 1. Cilindro do solo submetido à irrigação

4. Metodologia

Considerando que o sistema de irrigação de solo já tenha sido modelado em outros trabalhos [Padoin et al. 2011, Borges e Padoin 2006], fez-se um estudo dos algoritmos para o desenvolvimento desta nova implementação. Este estudo foi realizado almejando determinar os fatores que demandam grande processamento e podem ser divididos entre os chares. Os principais fatores são:

- o número ótimo de células da malha, necessário para obtenção dos resultados com a precisão desejada; e
- o número de execuções do problema inverso (chute).

Para esta nova versão paralela, diferente das demais, optou-se por paralelizar o cálculo do problema inverso em busca do menor erro. Deste modo, a implementação consiste em que cada *chare* execute paralelamente um chute do problema inverso.

Na implementação em CHARM++, foi realizada a transformação do código que estava de forma procedural para o paradigma orientado a objeto. Para isso foram criadas duas classes utilizando a linguagem C++, uma classe *main* e outra classe *Irrigação*. As mesmas foram declaradas nos arquivos de cabeçalho (extensão.*h*), onde todas as variáveis foram declaradas como privadas e os métodos declarados como público.

Após definir os métodos de cada classe, foi necessário definir quais métodos seriam os *métodos de entradas* de cada classe, em seguida os mesmos foram declarados nos arquivos de extensão *.ci*. Toda a implementação das classes e dos métodos foi realizada nos arquivos com extensão *.C*.

A classe *main*, ficou responsável em realizar o cálculo da evaporação, inicializar o vetor de *chares* da classe *Irrigação*, fazer o *broadcast* para todos os *chares* através do método de entrada da classe *Irrigação*, passando por parâmetro os dados do cálculo da evaporação e por fim, receber os dados calculados da classe *Irrigação* pelo seu método de entrada e realizar o cálculo do melhor caso. Sendo assim, a classe *Irrigação* recebe os dados de evaporação da classe *main*, realiza o problema inverso e devolve os dados para a classe *main*.

Para a análise de desempenho e consumo buscou-se variar os dois fatores que influenciam a aplicação, o tamanho da malha e o número de chutes do problema inverso. Assim para o primeiro fator buscou-se trabalhar com três tamanhos de malha: 128, 256 e 512. Para o segundo fator trabalhou-se com 100, 250 e 500 chutes, totalizando assim 9 configurações diferentes. Cada configuração foi testada com diferentes números de cores do ambiente de execução, variando-se a quantidade de cores em 1, 2, 4, 8, 16 e 32 o que resultou em 54 testes diferentes. Cada um destes testes foi realizado 10 vezes para se obter a média aritmética, esta que é multiplicada pela potência média(W) para o cálculo da energia consumida(J).

O ambiente de execução é composto de um equipamento com dois processadores Intel Xeon E5-2640 v2 com 8 cores SMT da microarquitetura *Ivy Bridge*, totalizando 32 cores de 2.00 GHz de frequência. Este equipamento possui 64 GB de memória RAM DDR3 de frequência 1600 MHz, utilizou-se do sistema operacional Ubuntu com versão de *kernel* $3.16.0 - 70$. A versão do CHARM++ utilizada foi a $6.7.0$ e do compilador g++ a $4.8.4$.

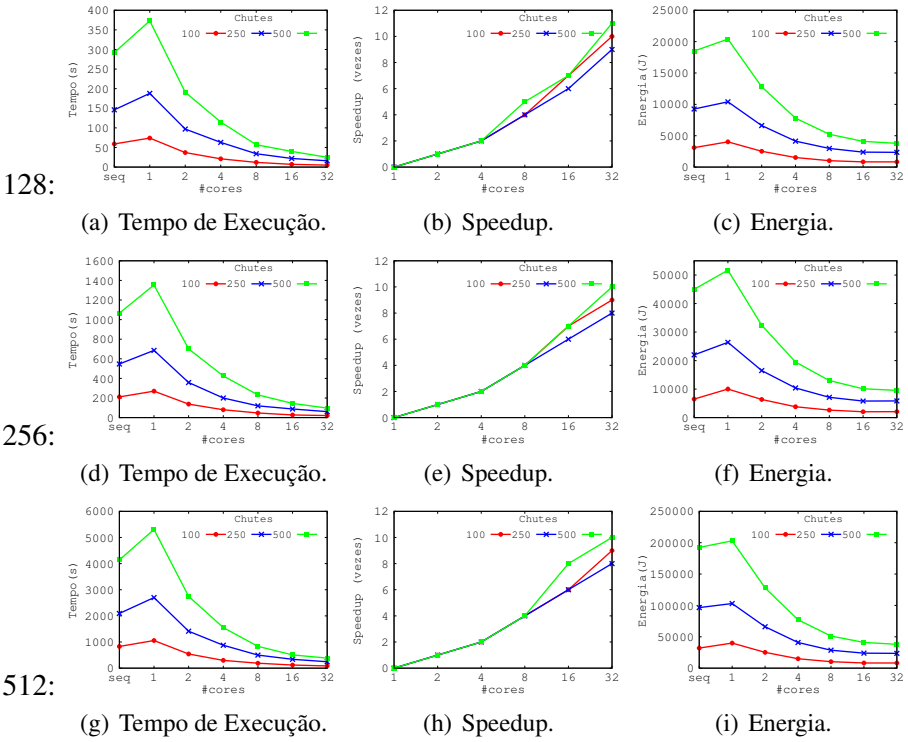
5. Resultados

Esta seção apresenta o desempenho alcançado nos experimentos utilizando a versão paralela do sistema. São analisados os resultados de tempo de execução, *speedup* e consumo de energia dos testes para 3 diferentes tamanhos de problema. Em cada teste é analisado as melhorias obtidas para diferentes quantidades de chutes variando-se o número de cores utilizados.

Analisando os resultados obtidos, percebe-se que o aumento da ordem da matriz tem impacto linear no tempo total de execução da aplicação com o algoritmo sequencial. Nos testes com 100 chutes do problema inverso (linha vermelha), e matriz de ordem 128 (Figura 2(a)) o tempo foi de $58,95$ segundos. Para a matriz de ordem 256 (Figura 2(d)) o tempo foi de $211,46$ segundos e para a matriz de ordem 512 (Figura 2(g)) foi de $830,46$ segundos,

Quando executada a versão paralela do algoritmo em apenas 1 core, o tempo total de execução é aumentado em média $27,5\%$. Este incremento representa o *overhead* do

CHARM++ para criação do *array* de *chares*.



(a) Tempo de Execução. (b) Speedup. (c) Energia.

(d) Tempo de Execução. (e) Speedup. (f) Energia.

(g) Tempo de Execução. (h) Speedup. (i) Energia.

Figura 2. Resultados da versão paralela com CHARM++.

Entretanto, ganhos já são alcançados com a versão paralela com 2 core em todos os testes. A partir deste ponto, o *overhead* passa ser menor que a redução no tempo total de execução, obtendo-se um *speedup* quase linear para todas as ordem testadas (Figuras 2(b), 2(e) e 2(h)). A redução do tempo em relação ao sequencial a partir da utilização de 2 core é em média 35% para todos os testes realizados.

Nos testes realizados, o maior *speedup* foi alcançado na execução de matriz de ordem 128, 500 chutes utilizando 32 cores. Para este teste a redução no tempo foi de $11,36$ vezes sobre o algoritmo sequencial, reduzindo assim o tempo de execução de $291,30$ segundos para $25,6$ segundos (Figuras 2(b).

Para o cálculo da energia consumida (Figuras 2(c), 2(f) e 2(i)) foi utilizado o software PCM Intel, que apresenta a demanda de potência da execução da aplicação. Os dados demostram que quanto mais cores são utilizados, maior é demanda de potência, porém esta não é influênenciada pelo tamanho da matriz e nem pela quantidade de chutes. Apesar do aumento da potência, o consumo de energia diminui pelo fato que ao utilizar mais cores reduz-se o tempo de execução.

6. Conclusões e Trabalhos Futuros

Neste trabalho foi analisado os ganhos de desempenho com a paralelização da aplicação de simulação de irrigação de solos utilizando o modelo de programação paralela CHARM++ sem o recurso de balanceamento de carga.

Utilizando-se diferentes configurações de simulação com a versão paralela, obteve-se redução de até 90% sobre o tempo da versão sequencial da aplicação. Mesmo sem ter

realizado balanceamento de carga, os resultados indicam que a utilização de CHARM++ para a paralelização de problemas reais em ambientes de multiprocessadores apresenta melhorias no tempo de execução. Se tomado como base uma simulação com matriz de ordem 65.536, a execução da versão sequencial com 500 chutes tomaria $547,9$ dias para chegar ao resultado. Com a versão paralela implementada em CHARM++, executando em 32 cores, o tempo seria reduzido para 50 dias. Mesmo assim, teríamos um tempo elevado para pesquisas científicas.

Como trabalhos futuros, almeja-se trabalhar em duas frentes. A primeira é na migração de tarefas, o que permitirá um aumento da utilização dos recursos computacionais, e segunda é a execução de testes com configurações de ambientes reais em equipamento paralelos com uma maior quantidade de recursos computacionais.

Agradecimentos

Trabalho parcialmente apoiado por UNIJUI, CNPq, CAPES, FAPERGS. Pesquisa realizada no contexto do Laboratório Internacional Associado LICIA e tem recebido recursos do edital PIBIC/UNIJUI e do MCTI/RNP-Brasil sob o projeto HPC4E de número 689772.

Referências

Arruda, G. H. S. e Padoin, E. L. (2015). Balanceamento de carga em sistemas multiprocessadores utilizando o modelo de programação charm++. *Salão do Conhecimento, Ijuí, RS, Brasil*, 1(1).

Borges, P. A., Coelho, G., e Buligon, S. (2005). Análise do comportamento da água em solos saturados e não saturados. Em *XVIII Congresso Nacional de Matemática Aplicada e Computacional, CNMAC, Santo Amaro, SP*.

Borges, P. A. e Padoin, E. L. (2006). Exemplos de métodos computacionais aplicados a problemas na modelagem matemática. *ERAD–Escola Regional de Alto Desempenho*, 6:5–20.

Kalé, L. V., Bhandarkar, M., e Brunner, R. (1998). Load balancing in parallel molecular dynamics. Em *International Symposium on Solving Irregularly Structured Problems in Parallel*, páginas 251–261. Springer.

Kang, S. J., Lee, S. Y., e Lee, K. M. (2015). Performance comparison of openmp, mpi, and mapreduce in practical problems. *Advances in Multimedia*, 2015:7.

Padoin, E. L., Pilla, L. L., Boito, F. Z., Kassick, R. V., e Navaux, P. O. (2011). Análise do consumo energético do algoritmo de irrigação de solos em arquiteturas heterogêneas. *Conferencia Latino Americana de Computación de Alto Rendimiento, Colima, Mexico. CLCAR*, páginas 1–7.

Pilla, L. L. e Meneses, E. (2015). Programação paralela em charm++. *ERAD/RS, Gramado, RS, Brasil*.

Satarić, B., Slavnić, V., Belić, A., Balaž, A., Muruganandam, P., e Adhikari, S. K. (2016). Hybrid openmp/mpi programs for solving the time-dependent gross–pitaevskii equation in a fully anisotropic trap. *Computer Physics Communications*, 200:411–417.

Tesser, R. K., Pilla, L. L., Dupros, F., Navaux, P. O. A., Méhaut, J.-F., e Mendes, C. (2014). Improving the performance of seismic wave simulations with dynamic load balancing. Em *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, páginas 196–203. IEEE.

Estudo de Implementação com Laços Paralelos em OpenMP 4 com *Offloading* para GPU no Método de Lattice Boltzmann*

Rafael G. Trindade[1], **João V. F. Lima**[1]

[1] Universidade Federal de Santa Maria (UFSM) – Santa Maria – RS – Brasil

{rtrindade,jvlima}@inf.ufsm.br

Abstract. *This work reports the study of parallel implementations aiming better performance, as well as making comparisons with existing ones, for the Lattice Boltzmann Method (LBM). The objective was to use the version 4 of OpenMP library and the distribution of parallel fors in threaded teams on a GPU. Factors such as acceleration and execution time are analyzed, which are presented in graphs, showing the effects of the library use, allowing a discussion about them.*

Resumo. *O presente artigo relata o estudo de implementações paralelas em busca de melhor desempenho, além de efetuar comparações com outras já existentes, para o Método de Lattice Boltzmann (MLB). Objetivou-se o uso da biblioteca OpenMP versão 4 e a distribuição de laços paralelos em times de threads em uma GPU. São analisados fatores como aceleração e tempo de execução, os quais são apresentados em gráficos, mostrando com clareza os efeitos do uso da biblioteca, possibilitando, por fim, uma discussão sobre os mesmos.*

1. Introdução

Como um método de dinâmica de fluídos computacional, o método de Lattice Boltzmann possibilita modelar computacionalmente vários tipos de problemas, permitindo simular fluxos com várias fases em geometrias complexas e condições de contorno diferentes [Schepke et al. 2009]. Como outros métodos de dinâmica de fluídos, o MLB é muito custoso do ponto de vista computacional, e implementações paralelas visando alto desempenho são, portanto, aplicáveis para que o MLB obtenha tempos de execução desejáveis.

Tais implementações fazem uso de conceitos vistos nas áreas de computação paralela e de alto desempenho, conceitos usados para criação de bibliotecas que permitem a exploração de paralelismo em aplicação, como a OpenMP. Com uma API rica, relativamente pouco verbosa e bastante amigável, OpenMP apresenta-se como uma alternativa que possibilita paralelizar aplicações escritas em C, C++ e Fortran, através do uso de diretivas de compilador, os *pragmas*. A biblioteca disponibiliza ao programador a capacidade de criar laços paralelos, seções concorrentes e tarefas assíncronas com poucas linhas de código. A versão 4 expandiu suas capacidades ao prover suporte à paralelismo intrínseco em arquiteturas híbridas, trazendo novas diretivas que possibilitam o uso de vetorização, criação de times de *threads* e *offloading* de dados e instruções para unidades de processamento externas como GPUs, Intel Xeon Phi e dispositivos ARM [Stanzani et al. 2016].

No decorrer deste trabalho será abordado o funcionamento da implementação do MLB, bem como as especificidades das variações de implementação propostas neste artigo que fazem uso dos métodos de *offloading* ofertados pelo OpenMP, comparando seus

*Trabalho desenvolvido recebendo fomento do Edital Nº 07/2016 PROBIC - FAPERGS/UFSM.

resultados com os de demais implementações já existentes. O artigo está organizado da seguinte maneira: a Seção 2 apresenta os trabalhos relacionados com o OpenMP 4 e com o MLB, o qual é melhor detalhado na Seção 3. Essa seção também detalha as implementações paralelas avaliadas do MLB. Os resultados experimentais são mostrados na Seção 4. Finalmente, a Seção 5 e a Seção 6 respectivamente, apresentam a análise dos resultados e conclusão deste trabalho.

2. Trabalhos Relacionados

Muito esforço já foi realizado com o intuito de modelar e implementar soluções paralelas de MLB. Muitas destas implementações são feitas aplicando a equação de Navier-Stokes para calcular o tempo médio da movimentação das partículas após uma colisão [Schepke and Diverio 2006]. Alguns trabalhos visaram dividir os dados em blocos, e redistribuir entre nodos computacionais usando MPI [Schepke and Maillard 2007, Schepke et al. 2009]. Trabalhos mais recentes abordam paralelização usando diversas bibliotecas *multithread*, como pthreads, OpenMP, Intel Cilk e CUDA, possibilitando ganhos consideráveis de desempenho em apenas um processador *multicore* e estendendo para GPU [Serpa et al. 2013, Serpa et al. 2015].

[McIntosh-Smith and Curran 2014] realizou comparações de desempenho em um processador Intel Xeon Phi e uma GPU, utilizando OpenCL. [Gabbana 2015] relatou a implementação de uma versão em OpenACC para uma arquitetura Multi-GPU. Destaca-se também um trabalho que implementara o uso de *offloading* com um protótipo do HOMP (*Heterogeneous OpenMP*), um modelo de acelerador para OpenMP que gera código para GPUs utilizando diretivas OpenMP 4, mas sobre uma implementação da versão 3.0 [Lin et al. 2015].

Com o lançamento da versão 4 em 2013[1], muitos autores exploraram seus novos recursos em seus trabalhos. [Bertolli et al. 2014] descreve o trabalho do suporte da versão para o compilador Clang, utilizada neste trabalho. Com um objetivo similar a este trabalho, [Bercea et al. 2015] mostra os resultados da implementação do LULESH[2] utilizando OpenMP 4 e suas construções de *offloading*.

3. Método de Lattice Boltzmann

O método de Lattice Boltzmann é diferente dos outros métodos numéricos de mecânica contínua e derivado da dinâmica molecular. As variáveis hidrodinâmicas são computadas nos nós como momentos de uma função de distribuição discreta. O MLB vem sendo desenvolvido nos últimos anos como uma ferramenta numérica viável para a dinâmica de fluídos computacionais. Ele possui um algoritmo simples, tratamento fácil de condições de fronteira e adequação ao paralelismo [Liu et al. 2016]. A estrutura de seu algoritmo é composta por um laço principal com um certo número de passos por iteração. Esses passos [Schepke and Maillard 2007] são ilustradas pelo laço principal do algoritmo na Figura 1.

3.1. Versões com MPI, OpenMP e CUDA

O trabalho inclui a execução de três implementações existentes do MLB:

[1] http://www.openmp.org/uncategorized/openmp-40/

[2] LULESH, página oficial: https://codesign.llnl.gov/lulesh.php

```
#pragma omp target data map(/* dados mapeados (malhas e vetor de obstaculos)*/)
for(/* numero de iteracoes definida para o laco principal*/){
    redistribute(...);
    propagate(...);
    bounceback(...);
    relaxation(...);
}
```

Figura 1. Laço principal com mapeamento de dados para GPU via *offloading*

- **MPI** – A versão com MPI destaca-se por conter um método a mais no laço principal, responsável pela sincronização entre os processos. Cada processo possui seu bloco local da malha, trabalha sobre ele durante uma iteração do algoritmo e então realiza comunicação para atualização das bordas. Um detalhe dessa implementação é o uso de *padding* nas bordas da malha, como ferramenta para auxílio nos cálculos parciais.
- **OpenMP** – Com uma quantia menor de código que a versão com MPI, a versão com OpenMP dispensa as preocupações com comunicação. Essa implementação foca em paralelizar os laços existentes dentro dos quatro métodos encontrados no laço principal. O escalonamento escolhido é o estático, com porções dos laços contíguas de tamanhos mais próximos o possível.
- **CUDA** – Nesta implementação, a malha é alocada na GPU e em toda a aplicação só há movimentação de dados antes e depois do laço principal, minimizando o tempo gasto com transferências entre memória principal e GPU. Cada etapa de uma iteração recebe um método em CUDA para execução inteiramente em GPU. O escalonamento das iterações seguem o recomendado para gpus, pulando de blocos em blocos

3.2. Versões com OpenMP 4 e *offloading* de dados e instruções

Tomando por base as versões OpenMP e MPI, duas implementações com OpenMP 4 e *offloading* foram criadas. Assim como na versão em CUDA, um dos desafios era minimizar as transferências de dados necessárias entre memória e GPU. Para tal, criou-se uma região de código com ambiente de dados mapeados para a GPU, com a construção `target data` e a cláusula `map`, como também pode ser conferido na Figura 1. Os dados mapeados foram a matriz da malha, um buffer temporário de mesmo tamanho e o vetor de obstáculos.

Para cada passo do laço principal que possuía laços, reaproveitou-se a diretiva `parallel for` da implementação em OpenMP para que os laços paralelos fossem executados em GPU. Para que isso fosse possível, adicionou-se a diretiva `target` antes. Adicionalmente, mapeou-se dados menores mas importantes para a GPU com a clásula `map`, e especificou-se que os laços fossem distribuídos por times de *threads* na GPU, com a diretiva `teams distribute` juntamente com a `parallel for`, como visto na Figura 2.

4. Resultados Experimentais

As execuções foram conduzidas em uma máquina servidora localizada no Laboratório de Sistemas de Computação da UFSM, denominada **lsc5**. Suas configurações incluem:

```
#pragma omp target map(/* dados menores mapeados */)
#pragma omp teams distribute parallel for
for(/* numero de iteracoes definida para o passo*/){
    /* codigo sensivel ao passo */
}
```

Figura 2. Laços paralelos com distribuição de carga para os times

- CPU Intel Xeon E5620, com 8 núcleos operando a 2.4 GHz, sendo 4 núcleos físicos e 2 núcleos lógicos por núcleo físico, e 12 Mb de *cache* L3;
- GPU Nvidia GeForce GTX Titan X[3], com 3072 núcleos CUDA operando a 1075 MHz, com 12 Gb de memória GDDR5 dedicada;
- 11 Gb de memória RAM DDR3;
- Sistema operacional Debian GNU/Linux Stretch, com kernel Linux 4.9.0, 64 bits;
- CUDA Toolkit versão 8.0.44, OpenMP versão 4.5 com suporte a *offloading* para a GPU supracitada, compiladores Clang versão 4.0 e GCC versão 6.3.0.

A fim de avaliar todas as implementações paralelas do MLB, executou-se diversos casos de teste. Foram feitas simulações de fluxos de fluido através de canais com obstáculos, representados por diferentes configurações. Foram consideradas oito tipos diferentes malhas quadradas, com lados de tamanhos em potência de 2, começando com tamanho 16x16 e indo até tamanho 4096x4096. Essa estrutura quadrada da malha simplifica a distribuição estática de trabalho entre as *threads*.

A Figura 3(a) exibe no formato de gráfico de linhas o tempo resultante das execuções nas diferentes configurações de malha para os diferentes tipos de implementação, enquanto que a Figura 3(b) exibe as acelerações conquistadas pelos tipos diferentes de implementações pros diferentes tamanhos de malha.

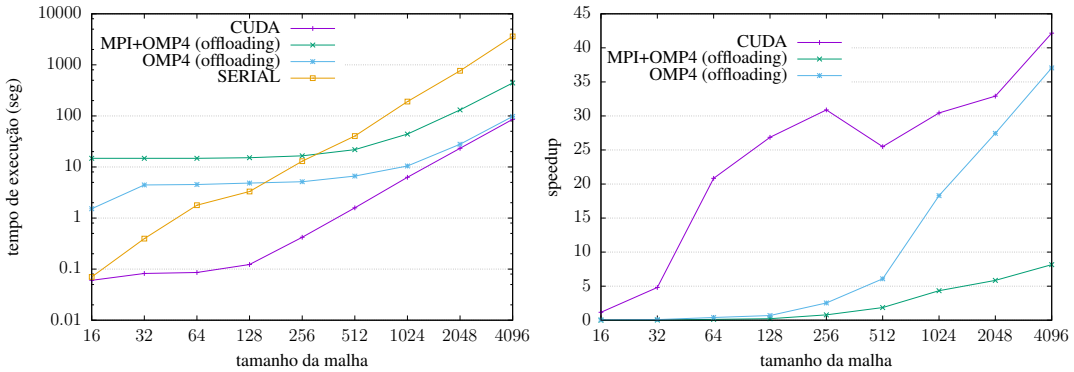


Figura 3. Na esquerda (a) os tempos de execução para cada tamanho de malha. Na direita (b) as acelerações conquistadas pelas implementações para cada tamanho de malha.

5. Análise de Desempenho

Com um olhar atento aos gráficos, denota-se diversos fatores resultantes das execuções. O gráfico de tempos (Figura 3(a)) mostra tempos de execução relativamente altos para

[3]Cedida pela Nvidia através do programa *Hardware Grant Program*

malhas pequenas quando comparadas à versão somente em CUDA e inclusive a versão sequencial. Isso se deve ao fato da diretiva de criação de times do OpenMP criar um número fixo de times por padrão (128), independente do tamanho do trabalho, o qual é dividido estaticamente graças a cláusula `distribute` [Bertolli et al. 2014]. Para malhas pequenas, cada time terá bloco pequeno de iterações, e muitas das *threads* geradas não terão trabalho. Por exemplo, para uma malha de tamanho 16x16 terá 2 iterações por time em laços que explorem toda a área da malha. Esse gargalo começa a ser dissipado no momento em que o tamanho da malha é maior que 362x362, quando cada *thread* de cada time executa pelo menos uma iteração, eliminando a ociosidade, isso pode ser visto na Figura 3(b) quando a aceleração começa a aumentar com tamanhos de malha a partir de 512x512. Isso pode ser evitado, entretanto com as clausulas `num_teams` e `thread_limit`, que definem o número de times e o máximo de threads por time, respectivamente.

A versão com MPI possui tempos em torno de 10 vezes maior em relação a versão com apenas OpenMP. Uma suspeita inicial fora a etapa de sincronização necessária a cada laço, mas a mesma fora descartada após conferência do perfil de execução da aplicação. Constatou-se, entretanto, que passos do laço que utilizam a cláusula `collapse` – cláusula responsável por fundir laços aninhados em apenas um, reduzindo a granularidade de trabalho a ser realizado por *thread* – demandaram aproxidamente um sexto do tempo de execução cada, fato não presente nas demais versões, devido a ausência da cláusula. Investigações futuras sobre o impacto da cláusula devem ser conduzidas.

A implementação com CUDA tende a perder desempenho conforme o tamanho da malha aumenta, já que ele não considera um limite para a criação de blocos em GPU, uma vez que o foco é delegar uma *thread* por iteração. Em tamanhos maiores de malhas, seu tempo de execução pode ser maior se comparado à versão com OpenMP.

Um fato destacável em todas as implementações é que o passo de propagação, apesar de ser relativamente simples quando comparado ao passo de relaxamento, uma vez que só realiza cópias dos valores da malha para a malha temporária, perde desempenho em GPU graças à acessos não-coalescentes à memória global, chegando a ocupar até 51% do tempo total de execução em malhas grandes.

6. Conclusão

Esse trabalho apresentou o uso de diretivas OpenMP para *offloading* de dados e instruções do método de Lattice Boltzman para uma GPU, fazendo proveito também de diretivas de criação de times de *threads* e criação de laços paralelos, distribuídos de forma estática por esses times.

Perante os resultados aqui descritos, conclui-se que o uso dessas diretivas pode trazer um ganho de desempenho muito semelhante à uma implementação puramente escrita em CUDA. Trabalhos futuros podem explorar potenciais ganhos de desempenho ao implementar uma configuração customizada de times e *threads* por time com o uso das cláusulas `num_teams` e `thread_limit`, que não a padrão, usada neste trabalho.

Agradecimentos

Este trabalho foi parcialmente financiado pelo PROBIC - FAPERGS / UFSM. Agradecimentos especiais ao programa NVIDIA Hardware Grant Program que cedeu a GPU

utilizada nos testes realizados nesse trabalho.

Referências

Bercea, G.-T., Bertolli, C., Antao, S. F., Jacob, A. C., Eichenberger, A. E., Chen, T., and Sura (2015). Performance analysis of OpenMP on a GPU using a CORAL proxy application. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, PMBS '15, pages 2:1–2:11, New York, NY, USA. ACM.

Bertolli, C., Antao, S. F., Eichenberger, A. E., O'Brien, K., Sura, Z., Jacob, A. C., Chen, T., and Sallenave, O. (2014). Coordinating GPU threads for OpenMP 4.0 in LLVM. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 12–21, Piscataway, NJ, USA. IEEE Press.

Gabbana, A. (2015). Accelerating the D3Q19 lattice boltzmann model with OpenACC and MPI. Master's thesis, Umeå University, Department of Computing Science.

Lin, P.-H., Liao, C., Quinlan, D. J., and Guzik, S. (2015). *Experiences of Using the OpenMP Accelerator Model to Port DOE Stencil Applications*, pages 45–59. Springer International Publishing, Cham.

Liu, Z., Fang, Y., Song, A., Xu, L., Wang, X., Zhou, L., and Zhang, W. (2016). *Parallel Overlapping Mechanism Between Communication and Computation of the Lattice Boltzmann Method*, pages 196–203. Springer International Publishing, Cham.

McIntosh-Smith, S. and Curran, D. (2014). Evaluation of a performance portable lattice boltzmann code using OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, pages 2:1–2:12, New York, NY, USA. ACM.

Schepke, C. and Diverio, T. A. (2006). Uso do método de lattice boltzmann em aplicações da hidrodinâmica. In *Anais da VI Escola Regional de Alto Desempenho - RS*, pages 111–112.

Schepke, C. and Maillard, N. (2007). Performance improvement of the parallel lattice boltzmann method through blocked data distributions. In *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*, pages 71–78.

Schepke, C., Maillard, N., and Navaux, P. O. A. (2009). Parallel lattice boltzmann method with blocked partitioning. *International Journal of Parallel Programming*, 37(6):593–611.

Serpa, M. S., Schepke, C., and Lima, J. V. F. (2015). Avaliação de desempenho do método de lattice boltzmann em arquiteturas multi-core e many-core. In *Anais do 16º Simpósio em Sistemas Computacionais*.

Serpa, M. S., Severo, E. B., and Schepke, C. (2013). Avaliando diferentes interfaces de programação paralela em simulação de fluxos de fluidos. In *Anais do 16º Simpósio em Sistemas Computacionais*, pages 234–237.

Stanzani, S. L., de Oliveira Cobé, R. M., and Iope, R. L. (2016). Introdução à programação multithreaded: explorando arquiteturas heterogêneas e vetorização OpenMP 4.0. In *Anais da XVI Escola Regional de Alto Desempenho - RS*, pages 89–112.

Proposta de Balanceamento de Carga para Redução do Tempo de Execução de Aplicações em Ambientes Multiprocessados

Vinícius R. S. dos Santos[1], **Ana Karina M. Machado**[1], **Edson L. Padoin**[1,2], **Philippe O. A. Navaux**[2], **Jean-François Méhaut**[3]

[1]Universidade Reg. do Noroeste do Estado do Rio G. do Sul (UNIJUI) - Ijuí - RS - Brasil

[2]Universidade Federal do Rio Grande do Sul (UFRGS) - Porto Alegre - RS - Brasil

[3]Universidade de Grenoble, Grenoble - France

vinirssantos@gmail.com, {ana.morales,padoin}@unijui.edu.br

navaux@inf.ufrgs.br, Jean-Francois.Mehaut@imag.fr

Abstract. *This paper presents a load balancer proposal to reduce the execution time of parallel applications when they run on multiprocessors environments. The load balancer algorithm collects system and application information in real time and uses them to make load balancing decisions dynamically, aiming to reduce the rate of migration and to decrease the execution time.* CHARM++ *was used for implementation due to its compatibility with the desired environments. The results show that our load balancer improves performance by reducing task migration from* 3 *up to* 137 *times and reductions from* 3.5% *to* 19.8% *on total execution time.*

Resumo. *Este artigo apresenta uma proposta de balanceamento de carga para a redução do tempo de execução de aplicações paralelas executadas em ambientes multiprocessados. O algoritmo do balanceador coleta informações do sistema e da aplicação em tempo real e as utiliza na tomada de decisões de balanceamento de carga dinamicamente, visando reduzir o número de migrações de tarefas enquanto reduzindo o tempo total de execução. Para implementação foi utilizado o modelo de programação paralela* CHARM++. *Os resultados preliminares apresentaram reduções de* 3 *à* 137 *vezes na quantidade de migrações de tarefas e reduções de* $3,5\%$ *à* $19,8\%$ *no tempo total de execução.*

1. Introdução

A demanda por sistemas de alto desempenho cresce cada vez mais, à medida que novas aplicações simulam sistemas cada vez mais complexos. As evoluções na forma de concepção dos sistemas e na fabricação dos processadores tem permitido ultrapassar barreiras de desempenho, uma vez que aplicações complexas são divididas em tarefas menores e executadas simultaneamente em nos núcleos das unidades de processamento.

No entanto, tais aplicações geralmente apresentam cargas computacionais diferentes, o que dificulta uma eficiente utilização dos sistemas computacionais. A modelagem deste problema é complexa fazendo com que muitas aplicações sejam executadas com desbalanceamento de carga e excessiva comunicação entre tarefas [Pilla and Meneses 2015, Padoin et al. 2017]. Essa é uma preocupação que surge devido ao seu caráter impeditivo quanto ao alcance de uma boa eficiência na utilização dos recursos dos sistemas paralelos [Padoin et al. 2014].

Nesse contexto, soluções que empregam estratégias para aumentar a eficiência dos recursos paralelos disponíveis vem sendo cada vez mais desenvolvidas e utilizadas. Balanceadores de Carga (BC), almejam detectar e corrigir dinamicamente tais desbalanceamentos, melhorando a utilização dos recursos disponíveis. Deste modo, este trabalho apresenta uma proposta de balanceador de carga denominado SMARTLB que almeja a redução do tempo de execução de aplicações paralelas executadas em ambientes multiprocessados.

O restante do trabalho está assim organizado: A Seção 2 apresenta os trabalhos relacionados. A Seção 3 apresenta a proposta do balanceador. As Seções 4 e 5 descrevem a metodologia que foi empregada na implementação do balanceador de carga SMARTLB e os resultados alcançados. Por fim, são discutidos na Seção 6, algumas considerações finais e perspectivas de trabalhos futuros.

2. Trabalhos Relacionados

Diferentes abordagens tem alcançado resultados positivos quando empregado balanceamento de carga para redução do tempo de execução. Dentre elas destacam-se as estratégias centralizadas e distribuídas, sendo que atualmente novas abordagem hierárquica vem sendo propostas. Nestas novas abordagens, os núcleos de processamento são divididos em grupos independentes e organizados em uma árvore onde cada nível da árvore é composto por grupos de núcleos. Deste modo, quanto mais núcleos são adicionados aos grupos, menor é o uso da memória pelo BC. Usando esta abordagem, Zheng apresenta um BC hierárquica denominado HYBRIDLB e consegue *speedup* de 6 com 2.048 *núcleos* e 145 com 8.192 *núcleos* em relação a versão sequencial [Zheng et al. 2010].

Por outro lado, estratégias centralizadas efetuam decisões de balanceamento de carga em um único processador. Para tanto, os dados de carga e comunicação de todas as tarefas são acumulados em um processador específico, o qual executa um processo de decisão com base nessas informações. Neste tipo de estratégia pode-se citar os balanceadores GREEDYLB e REFINELB. O primeiro, adota uma abordagem de agendamento agressivo, empregando uma heurística gulosa para tomada de decisões. Seu algoritmo objetiva migrar objetos pesados para o núcleo com menor carga, repetindo até que a carga de todos os processadores alcance uma proximidade com a carga média. Já o segundo, toma suas decisões considerando a distribuição de carga atual dos núcleos utilizados. A proposta é mover tarefas dos núcleos mais sobrecarregados para os menos carregados almejando atingir uma média, limitando o número do tarefas migradas [Zheng et al. 2011].

Outras ainda, chamadas de estratégias distribuídas visam melhorar o desempenho de sistemas de grande escala. Nessas estratégias, os processadores trocam informações apenas entre os seus vizinhos, como forma de descentralizar o processo de balanceamento de carga e apresentar menor sobrecarga de balanceamento de carga do que estratégias centralizadas [Kalé and Krishnan 1993]. Neste tipo de balanceador pode-se citar os balanceadores GRAPEVINELB e GRAPEPLUSLB. Este algoritmos realizam em paralelo, o cálculo da carga média de cada processador, sendo este valor médio usado para definir o estado global do sistema [Menon and Kalé 2013].

3. SmartLB

A implementação do balanceador de cargas SMARTLB foi realizada no ambiente de programação CHARM++. Este *framework* de balanceamento de carga foi escolhido uma vez

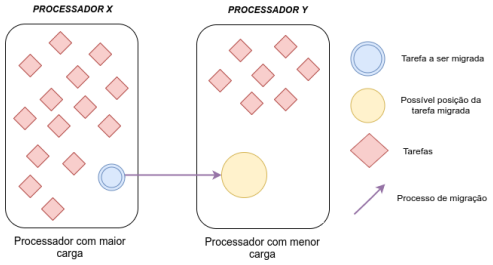
que permite tanto a criação de novos BC quanto a utilização dos BCs disponibilizados pelo ambiente para comparações de resultados.

O CHARM++ adota uma metodologia baseada na medição das cargas das tarefas que executam em cada núcleo. Para isso, o *framework* coleta automaticamente estatísticas da carga computacional e da comunicação destes objetos e armazena tais informações em uma base de dados que pode ser utilizada pelos BC para a tomada de decisões [Jyothi et al. 2004].

A estratégia utilizada para implementação do balanceamento de carga proposto constitui-se de melhorias nas estratégias utilizadas nos algoritmos GREEDYLB e REFINELB. Nossas melhorias buscam equilibrar as cargas entre os processadores reduzindo o número de migrações, adotando um *threshold* para definir o desbalanceamento de carga aceitável.

Na Figura 1 é exemplificado o funcionamento da estratégia de tomada de decisão do SMARTLB. A partir de informações fornecidas pelo CHARM++, o algoritmo busca atingir balanceamento levando em consideração a diferença de carga entre o núcleo mais carregado e o menos carregado. Desta forma, migra tarefas do processador mais carregado para o processador com menor carga, buscando equilibrar a carga total do sistema, reduzir o número de migrações e reduzir o tempo total de execução da aplicação.

Figura 1. Balanceamento de carga executado pelo algoritmo proposto



Na Tabela 1 são apresentados os principais parâmetros utilizados na implementação do algoritmo proposto.

Tabela 1. Principais parâmetros utilizados no algoritmo SMARTLB

Parâmetro	Definição
PM	Processador com maior carga
Pm	Processador com menor carga
D	Desbalanceamento entre PM e Pm
ct	Carga da tarefa P
$nTarefas$	Número de tarefas
$getProcessadorAtual(i)$	Retorna o processador atual da tarefa
$getCargaTarefa()$	Retorna a carga da tarefa
$getCargaMaior()$	Retorna a carga do processador mais carregado
$getCargaMenor()$	Retorna a carga do processador menos carregado
$migrarProcesso(i, PM, Pm)$	Migrar tarefa i de PM para Pm

Assim, quando o balanceador é aplicado, ele primeiro analisa a diferença de carga entre o processador mais e menos carregado. Caso essa diferença for maior que o *threshold* o balanceador busca tarefas do processador mais carregado testando se a carga da tarefa é menor ou igual ao desbalanceamento. Caso seja, ele realiza a migração desta tarefa do processador mais carregado para o menos carregado e encontra o novo processador mais carregado e o novo processador menos carregado, como demonstrado no Algoritmo 1.

Algoritmo 1: Implementação do SMARTLB

```
1  PM = getCargaMaior();
2  Pm = getCargaMenor();
3  if((Pm/PM) > Threshold) {
4      for(i = 1; i <= nTarefas; i++) {
5          if(getProcessadorAtual(i) == PM) {
6              ct = getCargaTarefa(i);
7              D = PM − Pm;
8              if(ct <= D) {
9                  migrarProcesso(i, PM, Pm);
10                 PM = getCargaMaior();
11                 Pm = getCargaMenor();
12             }
13         }
14     }
15 }
```

Desta forma, consegue-se um balanceamento de carga mais preciso, evitando migrações desnecessárias. Após a execução, quando não existem mais processadores a serem mapeados e as cargas de todas as unidades de processamento possuem um valor próximo um do outro, o balanceamento é encerrado.

4. Metodologia

Para validação da proposta, foi utilizado um equipamento com um processador Intel modelo i7-6500U. Este processador possui 4 núcleos com 2 Simultaneous multithreading(SMT)/núcleo, totalizando 8 núcleos. Para os testes, utilizou-se o sistema operacional Linux Manjaro 16.10 com kernel versão $4.4.33 - 1$. A versão do CHARM++ utilizada foi a $6.5.1$ e do compilador g++ a versão $5.4.1$.

Os balanceadores de carga foram submetidos a simulações utilizado o *benchmark* LB_Test. Esse *benchmark* foi escolhido por ser facilmente configurável para apresentar diferentes níveis de desbalanceamento de carga, permitindo que a carga computacional de cada tarefa seja configurada em diferentes padrões de carga, além de ser disponibilizado pelo próprio ambiente de programação.

Testes foram realizados com 50, 100 e 200 tarefas, sendo estas com cargas computacionais que variam entre $1500ms$ e 150000ms. As sincronizações para chamada do balanceador de carga foi definidas a cada 10 iterações.

Os resultados alcançados, tempo total de execução e a quantidade total de objetos migrados, foram comparados com os balanceadores de carga REFINELB e GREEDYLB.

5. Resultados

Na Figura 2 são apresentados os resultantes da simulação realizadas com o benchmark lb_test com 50, 100 e 200 tarefas.

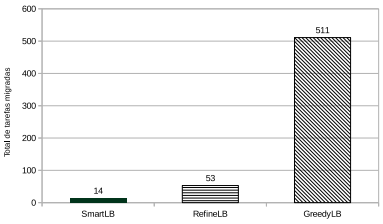
Observando os resultados mensurados, observa-se o tempos total de execução foi reduzido nos três testes executados. Com 50 tarefas, o tempo de execução com SMARTLB foi de $41,3$ segundos. Este tempo é $6,62\%$ menor que o algoritmo REFINELB que levou $44,35$ segundos e $3,57\%$ menor que o algoritmo GREEDYLB.

Esta redução no tempo total de execução é resultado da redução na quantidade total de tarefas migradas. Com 50 tarefas, o algoritmo do SMARTLB migrou apenas 14 tarefas enquanto que os BC REFINELB e GREEDYLB migraram 53 e 511 respectivamente.

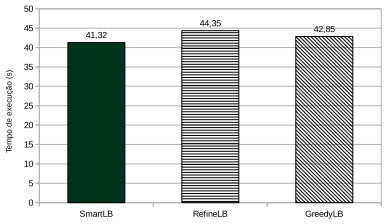
A disparidade na quantidade de tarefas migradas pelos balanceadores SMARTLB com GREEDYLB e REFINELB é consequência das suas estratégias. O primeiro, adota uma estratégia gulosa que migra praticamento todas as tarefas a cada chamada do balanceador de carga, enquanto que o segundo algoritmo, REFINELB faz um cálculo de média aritmética para determinar quais tarefas devem ou não serem migrados. No entando nossa proposta consegue evitar migrações desnecessárias e, consequentemente, diminui a quantidade de tarefas migradas e o tempo total de execução.

Figura 2. Resultados dos testes com 50, 100 e 200 tarefas

50:

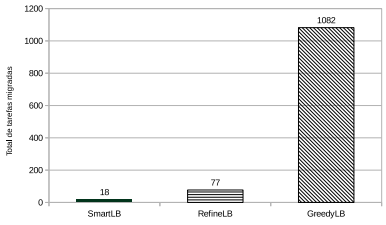


(a) Total de tarefas migradas

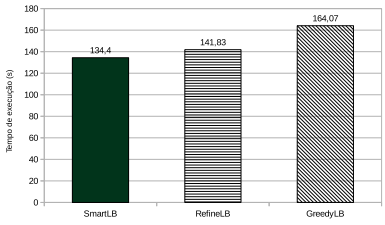


(b) Tempo total de execução (s)

100:

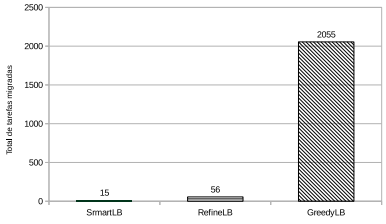


(c) Total de tarefas migradas

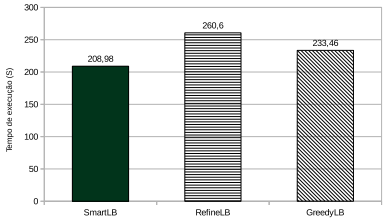


(d) Tempo total de execução (s)

200:



(e) Total de tarefas migradas



(f) Tempo total de execução (s)

Nos testes com 100 e 200 tarefas o SMARTLB também apresentou os menores tempos de execução, bem como também foi o BC que apresentou a menor quantidade de tarefas migradas.

Com 100 tarefas, o algoritmo do SMARTLB obteve um tempo de execução 5.24% menor que algoritmo do REFINELB e 18.08% menor que algoritmo do GREEDYLB. E, com 200 tarefas, o algoritmo do SMARTLB foi 19, 81% menor em relação com o REFINELB e 10.49% menor em relação ao GREEDYLB

Similar aos resultados alcançados com 50 tarefas, a melhora de desempenho em relação aos outros dois balanceadores de carga nas simulações com 100 e 200 tarefas, é justificada pela redução na quantidade de tarefas migradas. O SMARTLB migrou apenas 18 e 15 tarefas nos dois testes enquanto que REFINELB e GREEDYLB migraram 77 e 1082 tarefas nos testes com 100 tarefas e 56 e 2055 tarefas nos testes com 200 tarefas.

6. Conclusões

Este artigo apresentou uma proposta de balanceamento de carga para a redução do tempo de execução de aplicações paralelas executadas em ambientes multiprocessados. Os resultados obtidos foram comparados com outros dois balanceadores de carga do estado da arte.

O balanceador de carga proposto SMARTLB apresentou melhor desempenho nos testes realizados com o benchmark lb_test, tanto na quantidade de tarefas migradas quanto no tempo total de execução quando comparação com os balanceadores de carga REFINELB e GREEDYLB. Estes resultados positivos são alcançados devido ao maior controle na migração das tarefas, evitando assim migrações desnecessárias.

Como futuros trabalhos, pretende-se realizar melhorias no algoritmo SMARTLB almejando gerenciar ainda mais o controle de migrações de tarefas. Pretende-se também realizar testes em sistemas paralelos utilizando problemas reais de computação científica e comparar com outros balanceadores de carga do estado da arte.

Agradecimentos

Trabalho parcialmente apoiado por UNIJUI, CNPq e CAPES. Pesquisa realizada no contexto do Laboratório Internacional Associado LICIA e tem recebido recursos do edital PIBIC/UNIJUI, do programa EU H2020 e do MCTI/RNP-Brasil sob o projeto HPC4E de número 689772.

Referências

Jyothi, R., Lawlor, O. S., and Kalé, L. V. (2004). Debugging support for charm++. In *Parallel and Distributed Processing Symposium, 2004. 18th International*, page 264. IEEE.

Kalé, L. V. and Krishnan, S. (1993). CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings...*, pages 91–108. Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), ACM.

Menon, H. and Kalé, L. (2013). A distributed dynamic load balancer for iterative applications. In *Proceedings...*, page 15. International Conference for High Performance Computing, Networking, Storage and Analysis (SC), ACM.

Padoin, E. L., Castro, M., Pilla, L. L., Navaux, P. O., and Méhaut, J.-F. (2014). Saving energy by exploiting residual imbalances on iterative applications. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–10. IEEE.

Padoin, E. L., Navaux, P. O. A., and Méhaut, J.-F. (2017). Using power demand and residual load imbalance in the load balancing to save energy of parallel systems. In *International Conference on Computational Science (ICCS)*, pages 1–8, Zurich, Switzerland.

Pilla, L. L. and Meneses, E. (2015). Analise de desempenho da paralelização do problema de caixeiro viajante. *XV Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul, Gramado, RS, Brasil*.

Zheng, G., Bhatelé, A., Meneses, E., and Kalé, L. V. (2011). Periodic hierarchical load balancing for large supercomputers. *International Journal of High Performance Computing Applications*, 25(4):371–385.

Zheng, G., Meneses, E., Bhatele, A., and Kale, L. V. (2010). Hierarchical load balancing for charm++ applications on large supercomputers. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 436–444. IEEE.

The Boulitre Project: Implementing MPI-IO and HDF5 APIs Inside the IOR-Extended Benchmark

Rémi Savary[1,2], Eduardo C. Inacio[1], Mario A. R. Dantas[1], Jean-François Méhaut[2]

[1]Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brazil

[2]Polytech Grenoble – Université Grenoble Alpes (UGA)
Grenoble, France

savary.remi@gmail.com, eduardo.camilo@posgrad.ufsc.br,
mario.dantas@ufsc.br, jean-francois.mehaut@univ-grenoble-alpes.fr

Abstract. *The scientific world requires always more computational effort to be able to deal with large and complex problems. This led to the design and development of new and differentiated Parallel Files Systems (PFS). Designed for parallelism and high-performance, those are used to store large amounts of data and to maximize the speed up in terms of access. In order to have a better idea of the performances of these systems in terms of input/output speed up, some benchmarks were created, such as IOR, mpi-tile-io and NPB IO. IOR is one of the most used. Targeting to enhance the support for generation of a fluctuating workload, the present proposal, called as IOR-Extended (IORE), was conceived. This paper will explain the idea behind the implemented two APIs in order to extend IORE benchmarking abilities.*

1. Introduction

High Performance Computing (HPC) is a science approach in constant evolution, but it has always followed the same goal: being able to have always more powerful computational schemes. In the beginning, performances were mostly limited by hardware problems, such as stocking data, improving processors frequencies, having faster networks between machines, etc. Industries made tremendous efforts solving these problems. By focusing on the hardware improvements, new problems appeared, but in the software side, such as: how to optimize the utilization of these systems? Considering the ever-growing size of the data used, how to have a proper management of the memory?

That is why Parallel File Systems (PFS) were conceived following this idea of performance. They allow to manage concurrent access to stocked data, permit to have extremely large datasets, are designed to use high-performance networks and to optimize I/O operations. The most known PFS are Lustre, OrangeFS and GPFS.

Always considering the efficiency problem, and in order to measure the time spent reading/writing data by these systems, PFS benchmarks were implemented. The most used are: IOR (Interleaved Or Random), NPB-IO (NAS Parallel Benchmarks) and mpi-tile-io.

Considering this background, the IOR-Extended benchmark (IORE) [Inacio, 2016], was created, built on the model of the IOR, yet adding more possibilities than the original benchmark. The IORE project was initiated by the Distributed Systems Research Laboratory (LAPESD) of the Federal University of Santa Catarina (UFSC), in Florianópolis, Brazil. In this paper, we will present in further details the IORE, explain the purpose of the Boulitre project which is implementing two new testing interfaces in the IORE, and then show the first achieved results.

This paper is organized as follows: in Section 2, a more complete description of

the concept and the operation of the IORE is provided. The Section 3 contains the proposal, the Boulitre project. In Section 4, experimental environment and results are presented. Finally, Section 5 is dedicated to conclusions.

2. The IOR-Extended Benchmark

The IOR-Extended benchmark [Inacio, 2016] was created for two main reasons: first, it was meant to have the same global features than the IOR, but with in addition the possibility to generate a fluctuating workload. This is important because some application may work with a varying amount of data, so knowing how a system will answer and cope with this is crucial. The second purpose of the IORE is a pedagogic reason. Its implementation is made for being easier to understand than the previous benchmark, making it a good way for students to discover the working of benchmarks.

An experiment with the IORE as the following pattern:

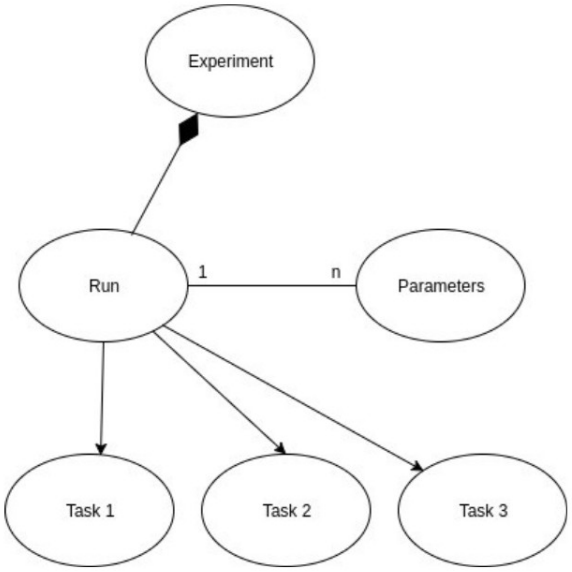


Figure 1. Schema of an IORE experiment.

It is composed of runs, each run taking its parameters from a JSON file given as input of the software. Each run may be executed on 1 or n tasks (process) given its parameters.

To be relevant, the benchmark must allow its user to give various parameters, depending on what characteristics the user wants to test. Some of these parameters are given in the Table 1.

It is possible to give several values of block_sizes for the same run. This way, the benchmark can generate stable or fluctuating workloads, making the IORE a more relevant benchmark than the IOR in terms of stress simulation.

3. The Boulitre Project

Both IOR and IOR-Extended benchmark end use are not only to have an idea of the data speed transmission, but also to help users to choose the best technologies for data management for their systems, thanks to the measurements provided. To do so, the

benchmarks have to be able to use those different technologies. It means that they have to provide different ways to perform experiments, more precisely they have to provide different ways to write/read into files.

IOR allows to do these operations by supporting 4 APIs: POSIX, Parallel NetCDF, MPI-IO and HDF5. Since IORE is still under development, it may only perform I/O using the POSIX API. The purpose of the Boulitre project is to implement two more APIs in IORE: MPI-IO and HDF5.

Table 1. Some parameters available in IORE

Parameter Name	**Meaning**
num_tasks	Number of tasks per run
api	API used for I/O operations
block_sizes	Total amount of data written/read
transfer_sizes	Size the data written/read per operation
write_test	The test performs a writing operation
read_test	The test performs a reading operation

3.1. Characteristics of the Libraries

MPI-IO are MPI (Message Passing Interface) routines that allow to perform read and write operations in parallel. They are part of the MPI-2 standard. They enable one or multiple processes to read/write data from/to one or multiple files. This is very useful while using PFS. Since MPI-IO is well-used and developed to work on PFS, it was an obvious choice to take.

HDF5 (Hierarchical Data Format 5) is an open-source technology which provides: a file format for storing data, a data model for organizing and accessing data from applications, and the libraries, language interfaces and tools for working with this format. It is made to store huge amounts of data, and is often used in scientific research applications, such as meteorology or molecular dynamics simulation.

3.2. Implementation of the APIs

To add new interfaces in the IORE benchmark, there is five functions mandatory to implement. The first is the file creation function. Given the parameters of the experiment (which contain the name of the file to be created), this function has to create a file in the file system using the right flags and to make sure that the file is accessible for writing and reading operations. The second function is an opening file function. Similar to the create file function, given the parameters of the experiment, this function opens the file required by the task. It also returns its file pointer. Then comes the closing function, which is necessary to close the file and everything else opened during the test, in order to end properly the program. The 4th function is a deleting file function. Thanks to the parameters, users can decide to suppress the file used for the experiment at the end of it. That is why a delete function is necessary. The last and the most important function performs writing and reading operations. Its role is not only to perform the good I/O operation, but also to make sure that the data is written/read in the right place inside the file, managing the position indicator for all the accessing processes.

3.3. Error Management

An important part of software development is error management and program robustness. In order to manage the different errors that might happen during the various operations, the result of every system call is checked, using a try/catch system adapted to each API. If the result is not what is expected, exceptions are thrown with an adapted error message.

4. Tests and First Results

The very first tests were done on a single machine using the following environment: an Ubuntu 16.04 LTS, running on 64 bits. The processor is a 2,4 x 4 GHz Intel® Core TM i7-5500U. The computer's RAM is 6 GB. The hard drive storage allocated for the OS is 145 GB.

Since the HDF5 API is still under development, the reading operation is not yet available. So, the following results will only show writing performances while using HDF5.

```
{
    "runs" : [
        {
            "params" : {
                "num_tasks" : 1 ,
                "api" : "MPIIO",
                "block_sizes" : [ "1G" ],
                "transfer_sizes" : [ "1G" ],
                "write_test" : true,
                "read_test" : false,
                "ref_num" : 1000,
                "root_file_name" : "test",
                "num_repetitions" : 1,
                "inter_test_delay" : 1,
                "intra_test_barrier" : true,
                "run_time_limit" : 1,
                "keep_file" : true,
                "single_io_attempt" : true
            }
        },

    ],

    "verbosity" : "DEBUG"
}
```

Figure 2. Example of parameters for an experiment.

The Figure 2 gives an overview of the JSON file used to give parameters to the experiment. Figure 3 shows the results given by the benchmark.

The result shows that the system spent 14,56 seconds to write 1 GB of data on the disk. The throughput of the data writing was 70,31 MiB/s.

Although for practical purposes, the results presented may not provide helpful insight on the performance of the compared APIs, they are important to assess the correctness of the extensions carried out in this project. The results here do not show a big difference between the 3 different interfaces in terms of performance, the only thing that appears here is that POSIX API has better results for small amounts of data.

```
Participating tasks: 1

Task 0 on Remi-PC
access    open(s)         io(s)    close(s)      total(s)   tput(MiB/s)  iter
Task 0 writing to file test
Delaying for 1 seconds...
Starting write performance test: Mon Aug  7 01:00:55 2017
Task 0 writing to offset 0
Run 0: Iter=0, Task=0, Time=1502060455.465194, write open start
Run 0: Iter=0, Task=0, Time=1502060455.465365, write open stop
Run 0: Iter=0, Task=0, Time=1502060455.465383, write start
Run 0: Iter=0, Task=0, Time=1502060470.028774, write stop
Run 0: Iter=0, Task=0, Time=1502060470.028776, write close start
Run 0: Iter=0, Task=0, Time=1502060470.028806, write close stop
write     0.0002       14.5634      0.0000       14.5636       70.3122     0
```

Figure 3. Results of an experiment.

Table 2. Performance comparisons. Data format: Time spent / Throughput.

Operation	**POSIX API**	**MPI-IO API**	**HDF5 API**
Write 1M	0.0005 s / 2003 MiB/s	0.0007 s / 1376.5 MiB/s	0.0029 s / 339.59 MiB/s
Write 1G	12.28 s / 83.3 MiB/s	13.967 s / 73.31 MiB/s	14.02 s / 73.03 MiB/s
Write 2G	30.28 s / 67.61 MiB/s	30.23 s / 67.73 MiB/s	30.87 s / 66.30 MiB/s
Read 1M	0.001 s / 1011.40 MiB/s	0.009 s / 105.43 MiB/s	
Read 1G	15.47 s / 66.17 MiB/s	14.17 s / 72.26 MiB/s	
Read 2G	19.24 s / 106.4 MiB/s	19.25 s / 106.38 MiB/s	

5. Conclusions

In this paper we introduce the Boulitre project, in which capabilities of the IOR-Extended benchmark are considerably extended by adding support for two parallel I/O APIs. IORE is designed to test the performances of a parallel file system using various parameters and is able to help users to know what may be the best technologies concerning data writing/reading.

The Boulitre project is not over yet, the HDF5 API has to be finished and tests need to be performed in a deeper way, using bigger systems working with PFS. Nevertheless, the addition of two new interfaces in the IORE makes it even more relevant, because it can now have a more complete approach and simulation of the problems it is supposed to help solving.

This project has been a great challenge, since we had 3 months to achieve it, and because it required to get familiar with HPC world, to learn the working of PFSs and benchmarks, and to get used to both MPI-IO and HDF5 libraries. We believe that this project is a big improvement which will help the IORE to get more relevant and making it a serious alternative to other known benchmarks.

References

Inacio E. C., (2016) “IOR-Extended Benchmark”, https://github.com/lapesd/iore

Lawrence Livermore National Laboratory, “IOR”, https://github.com/LLNL/ior

Prabhat and Koziol, Q. (2014) “High Performance Parallel I/O”, Chapman & Hall/CRC, Computational Science Series.

Inacio E. C., Barbetta P. A., Dantas M. A. R. (2017), “A Statistical Analysis of the Performance Variability of Read/Write Operations on Parallel File Systems”, ICCS 2017 – International Conference on Computational Science (Procedia Computer Science – Special Issue), Volume 108, Pages 2393-2397.

Inacio E. C., Pilla L. L., Dantas M. A. R., (2015) “Understanding the Effect of Multiple Factors on a Parallel File System’s Performance”, 2015 IEEE 24th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises.

The HDF Group (2017), “HDF5 Tutorial”, https://support.hdfgroup.org/HDF5/Tutor/.

Um Modelo de Reconhecimento de Atividades Humanas Baseado no Uso de Acelerômetro com QoC

Wagner D. do Amaral[1], **Mario A. R. Dantas**[1]

[1]Departamento de Informática e Estatística (INE) – Universidade Federal de Santa Catarina (UFSC)

wagner.amaral@grad.ufsc.com, mario.dantas@ufsc.br

Abstract. *This work presents a model proposal, entitled HARA (Human Activity Recognition with Accelerometer), which utilizes data related to the person location, movement and time. The model uses an accelerometer as its main sensor due to the low power consumption and small dimensions of this sensor. Identifying activities of an individual is particularly important because, in this way, it is possible to understand their individual routine, and based on that, one can identify abrupt changes in such routine, which may be related to the early stage of a disease. Aiming the efficiency of the model, the use of a filter for quality of context (QoC) is also proposed, in a way that data with low QoC value are dropped.*

Resumo. *Este trabalho apresenta o modelo HARA (Human Activity Recognition with Accelerometer), para reconhecimento de atividades humanas a partir de dados de localização, movimentação e tempo. O modelo utiliza um acelerômetro como principal sensor devido ao baixo consumo energético e as pequenas dimensões desse sensor. Identificar atividades de um indivíduo é importante pois, desta forma, é possível compreender sua rotina individual, e baseado nisso, identificar mudanças bruscas, que podem estar relacionadas a estágios iniciais de uma doença. Visando a eficiência do modelo, também é proposto o uso de um filtro de qualidade de contexto, onde dados com baixo valor na qualidade de contexto (QoC) são descartados.*

1. Introdução

Segundo projeções do IBGE, o número de idosos - pessoas com mais de 60 anos - no Brasil em 2060, representará quase 34% da população do Brasil. Em números absolutos a estimativa é que o número de idosos praticamente triplique até 2060 [IBGE 2013].

Com esse grande crescimento da população de idosos, vê-se a oportunidade e a necessidade da criação de condições que os garanta uma vida com qualidade e independência. Como solução, estão surgindo tecnologias que monitoram indivíduos em suas casas, os chamados ambientes domiciliares assistidos.

Um ambiente domiciliar assistido é composto por um conjunto heterogêneo de sensores corporais e de ambiente, que geram grandes volumes de dados. A qualidade dos dados, portanto, é a chave para a manutenção desses ambientes [Forkan et al. 2015]. Para lidar com esse grande volume de dados e garantir que o modelo seja acessível, é necessário processamento de alto desempenho com arquiteturas de baixo custo, além da

implementação de um analisador de contexto que filtre os dados, descartando informações com pouco ou nenhum valor semântico.

Ambientes domiciliares assistidos tradicionais usam diversas técnicas para detecção de anormalidades, porém, de uma forma geral, detectam anormalidades somente quando elas ocorrem, o que muitas vezes está ligado ao estado avançado de uma doença [Forkan et al. 2015]. Neste tipo de sistema, não é possível prever anomalias com tempo hábil para que ações preventivas sejam tomadas, o que evitaria que a situação se torne crítica.

Para preencher essa lacuna deixada pelos ambientes domiciliares assistidos tradicionais, este trabalho propõe o modelo HARA (*Human Activity Recognition with Accelerometer*). Ao armazenar dados do posicionamento na residência, movimentação e tempo, é possível inferir as atividades diárias de um indivíduo, e assim entender sua rotina individual. Mudanças bruscas na rotina podem estar relacionadas ao estágio inicial de alguma doença, e a identificação de uma doença em sua etapa inicial geralmente representa maiores chances de um tratamento bem sucedido.

O modelo HARA utiliza um acelerômetro como principal sensor para o reconhecimento de atividades humanas devido ao baixo consumo energético e às pequenas dimensões desse sensor, favorecendo a não intrusividade, e garantindo assim uma maior qualidade de vida para o usuário monitorado. A adição da informação da localização *indoor* garante uma maior precisão para a identificação das atividades, uma vez que atividades humanas e localização estão fortemente correlacionadas em ambientes internos [Zhu and Sheng 2011].

Este trabalho de pesquisa está dividido da seguinte forma: na seção 2 é apresentada uma revisão da literatura e alguns trabalhos correlatos, a proposta está descrita na seção 3. Os dados preliminares do modelo são apresentados na seção 4, e as conclusões e recomendações para trabalhos futuros estão descritas na seção 5

2. Revisão Bibliográfica e Trabalhos Correlatos

Recentemente, muitos artigos passaram a abordar o tema de ambientes assistidos. Por se tratar de um conceito relativamente novo, poucos padrões são encontrados na literatura. Alguns trabalhos, como [Mannini et al. 2013], utilizam apenas um único acelerômetro para o reconhecimento das atividades. Outros trabalhos, como [Dwiyantoro et al. 2016], [Murao and Terada 2016] e [Kim et al. 2015], utilizam diversos sensores para a mesma finalidade.

O uso de sensores corporais é amplamente utilizado nessa área, porém outras abordagens também são encontradas na literatura, como a utilização de imagens para o reconhecimento das atividades. Esse é o caso de trabalhos como [Maurer et al. 2006].

Para os sensores corporais, a posição do corpo onde o dispositivo será acoplado possui grande importância. Apesar de também não existir um padrão sobre onde os sensores devem ser posicionados, alguns dos lugares mais comuns são: peito, pulso, tornozelo e cintura. Em [Atallah et al. 2011] é realizado um estudo para determinar o melhor posicionamento para sensores corporais.

Para melhor compreender o estado atual da pesquisa nessa área, realizou-se uma revisão da literatura. Foram realizadas buscas pelo termo "*activity recognition* AND *acce-*

lerometer" no título de documentos de 4 bases de dados: *IEEE Xplorer*, *ProQuest*, *Scopus* e *Web of Science*. Todas as buscas foram realizadas em janeiro de 2017, sendo excluídos documentos redigidos em idioma diferente do português, inglês ou espanhol, documentos sem o texto completo disponível e documentos repetidos. Como resultado, foi encontrado um total de 253 artigos entre as 4 bases de dados, desses, 76 foram selecionados para uma análise mais detalhada.

2.1. Comparativo das Abordagens

A Tabela 1 apresenta uma comparação entre as abordagens utilizadas em alguns dos trabalhos selecionados durante a revisão da literatura.

Tabela 1. Comparativo de Trabalhos de Pesquisa

Referência	Sensor	Localização do sensor	Método de classificação
[Mannini et al. 2013]	Acelerômetro	Pulso ou Tornozelo	Support Vector Machine
[Dwiyantoro et al. 2016]	Acelerômetro e Sensor de gravidade (smartphone)	Bolso frontal da calça	Dynamic Time Warping e K-Nearest Neighbors
[Murao and Terada 2016]	Múltiplos acelerômetros	Ambos pulsos e tornozelos e quadril	Dynamic Time Warping e Support Vector Machine
[Kim et al. 2015]	Acelerômetro e Giroscópio (smartphone)	Cintura	Hidden Markov Model

3. Proposta

este trabalho propõe o modelo HARA, um modelo de análise de dados com a finalidade de inferir as atividades realizadas por um indivíduo monitorado dentro de sua casa. O sistema ainda considera a abordagem de QoC, visando uma avaliação da qualidade de contexto dos dados recebidos e que possa levar a uma decisão do que deve ser descartado e o que deve dar origem a alertas. Uma visão geral do modelo pode ser observada na Figura 1.

3.1. Obtenção dos Dados

Como citado anteriormente, serão obtidos dados de posicionamento, movimentação e tempo. Para a obtenção desses dados, um dispositivo deverá ser montado a partir de uma placa de processamento *Tinyduino*, acoplando a ela os *shields* de acelerômetro e BLE (*Bluetooth Low Energy*). Esse dispositivo deverá ser posicionado no corpo do indivíduo a ser monitorado. Segundo [Atallah et al. 2011], para a identificação de atividades como andar e afazeres domésticos, os melhores lugares para posicionar o acelerômetro seriam o pulso ou o peito. Levando em consideração a comodidade para o indivíduo monitorado, decidiu-se acoplar o dispositivo, de dimensões 20mm x 20mm x 16.8mm, em seu pulso.

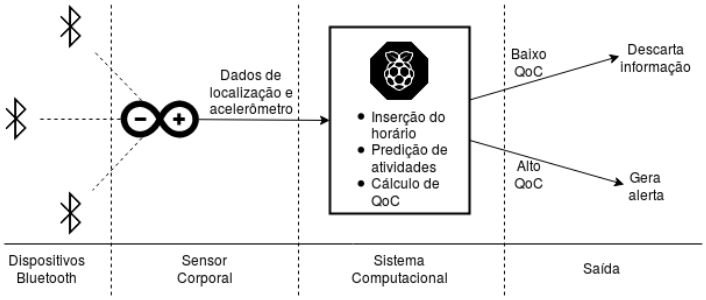


Figura 1. HARA - Visão Geral.

A escolha dos *shields* de acelerômetro e BLE para incorporar o sensor corporal levou em consideração o baixo consumo energético desses *shields*. O acelerômetro também apresenta-se como uma alternativa ideal para o modelo proposto devido a suas pequenas dimensões, favorecendo a não intrusividade.

Para os dados de movimentação, serão utilizadas as informações do acelerômetro. Para os dados de localização, o módulo BLE ficará responsável por detectar a intensidade de sinal dos dispositivos disponíveis para pareamento. Haverá um dispositivo BLE para cada cômodo da casa, e as intensidades serão comparadas para determinar a posição do indivíduo na casa. A cada 200ms esses dados serão enviados, através do BLE, do *Tinyduino* para um *Rasperry Pi 3*, dispositivo responsável pelo processamento desses dados, e inclusão da informação de tempo.

3.2. Predição das Atividades

Após a obtenção e transmissão dos dados de movimentação e localização para o *Raspberry Pi 3*, com a adição da informação de tempo, será realizado um processo de predição para inferência de qual atividade o indivíduo monitorado está realizando. Para a implementação dessa predição, a técnica utilizada será o Modelo Oculto de Markov (HMM) [Blunsom 2004].

A escolha pelo HMM se deve a esta técnica ser capaz de representar uma quantidade infinita de sequência possíveis, através de uma quantidade finita de estados. Assim, a técnica se mostra ideal para a necessidade do modelo proposto, representar uma sequência qualquer de ações executadas por um indivíduo, por meio de uma quantidade limitada de estados.

3.3. Qualidade de Contexto (QoC)

É proposto também, que o modelo avalie a qualidade de contexto dos dados recebidos e decida o que deve ser descartado e o que deve dar origem a alertas. Para o cálculo da QoC, será utilizada a abordagem proposta por [Nazário et al. 2014], na qual cinco parâmetros são considerados para determinar o resultado desse cálculo. São ele: *Completeness*, *Coverage*, *Precision*, *Up-to-dateness* e *Significance*. Cada um dos parâmetros possui valor entre 0 e 1, e é realizada uma média entre eles, excluindo o parâmetro *Significance*. Esse último parâmetro possuirá valor 1 se os valores medidos são válidos, mas não esperados, indicando um caso onde a criação de um alerta é necessária, ou 0, nos demais casos, indicando que os dados podem ser descartado.

4. Ambiente e Resultados Experimentais

Nesta seção, serão apresentados os dados preliminares do modelo HARA. Inicialmente é realizado um experimento para a leitura dos dados do acelerômetro, que em seguida passam pelo filtro QoC. Para esta etapa, foram monitorados os movimentos de 5 indivíduos, durante uma hora cada.

A Figura 2 apresenta os valores lidos durante 35 segundos, pelo acelerômetro ($\pm$ 2g) acoplado ao pulso de um dos indivíduos, enquanto o mesmo realizava atividades em sua própria casa.

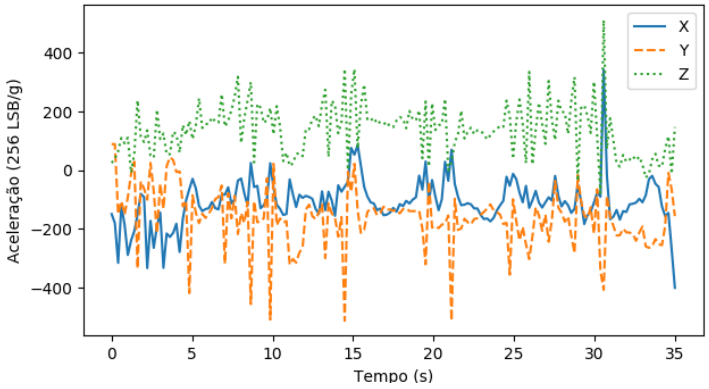


Figura 2. Aceleração Referente a Sequência de Atividades.

Através de uma análise do gráfico, é possível observar que durante a sequência de atividades realizadas, os valores lidos pelo acelerômetro vão se alterando, porém, fica claro o quão difícil e ineficaz seria uma identificação manual dessas atividades. Com isso, temos a importância do modelo HARA, que se propõe a identificar instantaneamente a atividade realizada pelo indivíduo.

A Tabela 2 expõe dados referentes a aplicação do filtro QoC nos dados recebidos durante os 35 segundos de monitoramento. Esse filtro é responsável por garantir a qualidade dos dados. O parâmetro *Significance* não foi considerado para este experimento preliminar devido a ausência de valores de referência.

Tabela 2. Aplicação de filtro QoC

Situação	Quantidade
Sucesso	172
Filtrado	3
Total	175

5. Conclusões e Trabalhos Futuros

Este trabalho apresentou um modelo de reconhecimento de atividades humanas, através do uso de sensor corporal para a captação de dados de movimentação e localização, além do tempo. O modelo também considera o uso da abordagem de QoC, visando uma avaliação da qualidade dos dados recebidos, levando a decisão do que deve ser descartado e o que deve dar origem a alertas.

O uso do acelerômetro como sensor principal para a identificação de atividades, se deu principalmente por seu baixo consumo energético e suas pequenas dimensões, o que favorece a não intrusividade.

Para trabalhos futuros é sugerido a comparação da eficiência entre diferentes tecnologias de comunicação, como o *bluetooth* 4.0 clássico, ou *ZigBee*. Outra comparação possível diz respeito a técnica de classificação. Com base na revisão da literatura, visto na seção 2, observou-se que quase 50% dos trabalhos selecionados utilizam algoritmos de árvore de decisão ou *Support Vector Machine*. Desta forma, a comparação entre a utilização do Modelo Oculto de Markov com os algoritmos citados, também seria interessante.

Referências

Atallah, L., Lo, B., King, R., and Yang, G. Z. (2011). Sensor positioning for activity recognition using wearable accelerometers. *IEEE Transactions on Biomedical Circuits and Systems*, 5(4):320–329.

Blunsom, P. (2004). Hidden markov models. *Lecture notes, August*, 15:18–19.

Dwiyantoro, A. P. J., Nugraha, I. G. D., and Choi, D. (2016). A simple hierarchical activity recognition system using a gravity sensor and accelerometer on a smartphone. *International Journal of Technology*, 7(5).

Forkan, A. R. M., Khalil, I., Tari, Z., Foufou, S., and Bouras, A. (2015). A context-aware approach for long-term behavioural change detection and abnormality prediction in ambient assisted living. *Pattern Recognition*, 48(3):628–641.

IBGE (2013). Projeção da população do brasil por sexo e idade 2000-2060. Revisão 2013.

Kim, Y. J., Kang, B. N., and Kim, D. (2015). Hidden markov model ensemble for activity recognition using tri-axis accelerometer. In *2015 IEEE International Conference on Systems, Man, and Cybernetics*, pages 3036–3041.

Mannini, A., Intille, S. S., Rosenberger, M., Sabatini, A. M., and Haskell, W. (2013). Activity recognition using a single accelerometer placed at the wrist or ankle. *Medicine and Science in Sports and Exercise*, 45(11):2193–2203.

Maurer, U., Smailagic, A., Siewiorek, D. P., and Deisher, M. (2006). Activity recognition and monitoring using multiple sensors on different body positions. In *Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks*, BSN '06, pages 113–116, Washington, DC, EUA. IEEE Computer Society.

Murao, K. and Terada, T. (2016). A combined-activity recognition method with accelerometers. *Journal of Information Processing*, 24(3):512–521.

Nazário, D. C., Tromel, I. V. B., Dantas, M. A. R., and Todesco, J. L. (2014). Toward assessing quality of context parameters in a ubiquitous assisted environment. In *2014 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, Madeira, Portugal. IEEE Computer Society.

Zhu, C. and Sheng, W. (2011). Motion- and location-based online human daily activity recognition. *Pervasive Mob. Comput.*, 7(2):256–269.

Optimizing a Boundary Elements Method for Stationary Elastodynamic Problems implementation with GPUs

Giuliano A. F. Belinassi[1], Rodrigo Siqueira[1], Ronaldo Carrion[2] , Alfredo Goldman[1], Marco D. Gubitoso[1]

[1]Instituto de Matemática e Estatística (IME) – Universidade de São Paulo (USP)
Rua do Matão, 1010 – São Paulo – SP – Brazil

[2]Escola Politécnica (EP) – Universidade de São Paulo (USP)
Avenida Professor Mello Moraes, 2603 – São Paulo – SP – Brazil

Abstract. *The Boundary Element Method requires a geometry discretization to execute simulations, and it can be used to analyze the 3D stationary behavior of wave propagation in the soil. Such discretization involves generating two high computational power demanding matrices, and this article demonstrates how Graphical Processing Units (GPU) were used to accelerate this process. In an experiment with 4000 Mesh elements and 1600 Boundary elements, a speedup of 107× was obtained with a GeForce GTX980.*

1. Introduction

Differential equations governing problems of Mathematical Physics have analytical solutions only in cases in which the domain geometry, boundary and initial conditions are reasonably simple. Problems with arbitrary domains and fairly general boundary conditions can only be solved approximately, for example, by using numerical techniques. These techniques were strongly developed due to the presence of increasingly powerful computers, enabling the solution of complex mathematical problems.

The Boundary Element Method (BEM) is a very efficient alternative for modeling unlimited domains since it satisfies the Sommerfeld radiation condition, also known as geometric damping [Katsikadelis 2016]. This method can be used for numerically modeling the stationary behavior of 3D wave propagation in the soil and it is useful as a computational tool to aid in the analysis of soil vibration [Dominguez 1993]. A BEM based tool can be used for analyzing the vibration created by heavy machines, railway lines, earthquakes, or even to aid the design of offshore oil platforms.

With the advent of GPUs, several mathematical and engineering simulation problems were redesigned to be implemented into these massively parallel devices. However, first GPUs were designed to render graphics in real time, as a consequence, all the available libraries, such as OpenGL, were graphical oriented. These redesigns involved converting the original problem to the graphics domain and required expert knowledge of the selected graphical library.

NVIDIA noticed a new demand for their products and created an API called CUDA to enable the use of GPUs for general purpose programming. CUDA uses the concept of kernels, which are functions called from the host to be executed by GPU threads. Kernels are organized into a set of blocks composed of a set of threads that cooperate with each other [Patterson and Hennessy 2007].

The memory of a NVIDIA GPU is divided in global memory, local memory, and shared memory. Global memory is accessible by all threads, local memory is private to a thread and shared memory is low-latency and accessible by all threads in a block[Patterson and Hennessy 2007]. CUDA provides mechanisms to access all of them.

Regarding this work, this parallelization approach is useful because an analysis of a large domain requires a proportionally large number of mesh elements, and processing a single element have a high time cost. Doing such analysis in parallel reduces the computational time requied for the entire program because multiple elements are processed at the same time. This advantage was provided by this research.

Before discussing any parallelization technique or results, Section 2 presents a very brief mathematical description of BEM for Stationary Elastodynamic Problems and the meaning of some functions presented in this paper. Section 3 shows how the most computational intensive routine was optimized using GPUs. Section 4 discusses how the results were obtained. Section 5 presents and discusses the results. Finally, Section 6 provides an overview of our future work.

2. Boundary Elements Method Background

Without addressing details on BEM formulation, the Boundary Integral Equation for Stationary Elastodynamic Problems can be written as:

$$c_{ij}u_j(\xi,\omega) + \int_S t^*_{ij}(\xi,x,\omega)u_j(x,\omega)\mathrm{d}S(x) = \int_S u^*_{ij}(\xi,x,\omega)t_j(x,\omega)\mathrm{d}S(x) \tag{1}$$

After performing the geometry discretization, Equation (1) can be represented in matrix form as:

$$Hu = Gt \tag{2}$$

Functions $u^*_{ij}(\xi,x,\omega)$ and $t^*_{ij}(\xi,x,\omega)$ (called fundamental solutions) present a singular behavior when $\xi = x$ ordely $O(1/r)$, called weak singularity, and $O(1/r^2)$, called strong singularity, respectively. The r value represents the distance between x and ξ points. The integral of these functions, as seen in Eq. (1), will generate the G and H matrices respectively, as is shown in Eq. (2). For computing these integrals numerically, the Gaussian quadrature can be deployed. Briefly, it is an algorithm that approximates integrals by sums as shown in equation (3)[Ascher and Greif 2011], where g is the number of Gauss quadrature points.

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{i=1}^{g} w_i f(x_i) \tag{3}$$

To overcome the mentioned problem in the strong singularity, one can use the artifice known as Regularization of the Singular Integral, expressed as follows:

$$\begin{aligned} c_{ij}(\xi)u_j(\xi,\omega) + \int_S \left[t^*_{ij}(\xi,x,\omega)_{\text{DYN}} - t^*_{ij}(\xi,x)_{\text{STA}}\right] u_j(x,\omega)\mathrm{d}S(x) + \\ + \int_S t_{ij}(\xi,x)_{\text{STA}}u_j(x)\mathrm{d}S(x) = \int_S u^*_{ij}(\xi,x,\omega)_{\text{DYN}}t_j(x,\omega)\mathrm{d}S(x) \end{aligned} \tag{4}$$

Where DYN = Dynamic, STA = Static. The integral of the difference between the dynamic and static nuclei, the first term in Equation (4), does not present singularity when executed concomitantly as expressed because they have the same order in the both problems.

Algorithmically, equation (1) is implemented into a routine named `Nonsingd`, computing the integral using the Gaussian Quadrature without addressing problems related to singularity. To overcome singularity problems, there is a special routine called `Sing_de` that uses the artifice described in equation (4). Lastly, `Ghmatecd` is a routine developed to create both the H and G matrices described in equation (2). Both `Nonsingd` and `Sing_de` are called from `Ghmatecd` routine.

3. Parallelization Strategies

A parallel implementation of BEM began by analyzing and modifying a sequential code provided by [Carrion 2002]. Gprof, a profiling tool by [GNU], revealed the two most time-consuming routines: `Ghmatecd` and `Nonsingd`, with 60.9% and 58.3% of the program total elapsed time, respectively. Since most calls to `Nonsingd` were performed inside `Ghmatecd`, most of the parallelization effort was focused on that last routine.

3.1. `Ghmatecd` Parallelization

Algorithm 1 shows pseudocode for the `Ghmatecd` subroutine. Let n be the number of mesh elements and m the number of boundary elements. `Ghmatecd` builds matrices H and G by computing smaller 3×3 matrices returned by `Nonsingd` and `Sing_de`.

Algorithm 1 Creates $H, G \in \mathbb{C}^{(3m)\times(3n)}$

```
1:  procedure GHMATECD
2:      for j := 1, n do
3:          for i := 1, m do
4:              ii := 3(i − 1) + 1; jj := 3(j − 1) + 1
5:              if i == j then
6:                  Gelement, Helement ← Sing_de(i)    ▷ two 3 × 3 complex matrices
7:              else
8:                  Gelement, Helement ← Nonsingd(i, j)
9:              G[ii : ii + 2][jj : jj + 2] ← Gelement
10:             H[ii : ii + 2][jj : jj + 2] ← Helement
```

There is no interdependency between all iterations of the loops in lines 2 and 3, so all iterations can be computed in parallel. Since typically a modern high-end CPU have 8 cores, even a small number of mesh elements generate enough workload to use all CPUs resources if this strategy alone is used. On the other hand, a typical GPU contain thousands of processors, hence even a considerable large amount of elements may not generate a workload that consumes all the device's resources. Since `Nonsingd` is the cause of the performance bottleneck of `Ghmatecd`, the main effort was implementing an optimized version of `Ghmatecd`, called `Ghmatecd_Nonsingd`, that only computes the `Nonsingd` case in the GPU, and leave `Sing_de` to be computed in the CPU after the computation of `Ghmatecd_Nonsingd` is completed. The pseudocode in Algorithm

2 pictures a new strategy where `Nonsingd` is also computed in parallel. Let g be the number of Gauss quadrature points.

Algorithm 2 Creates $H, G \in \mathbb{C}^{(3m)\times(3n)}$

```
1:  procedure GHMATECD_NONSINGD
2:      for j := 1, n do
3:          for i := 1, m do
4:              ii := 3(i − 1) + 1; jj := 3(j − 1) + 1
5:              Allocate Hbuffer and Gbuffer, buffer of matrices 3 × 3 of size g^2
6:              if i ≠ j then
7:                  for y := 1, g do
8:                      for x := 1, g do
9:                          Hbuffer(x, y) ← GenerateMatrixH(i, j, x, y)
10:                         Gbuffer(x, y) ← GenerateMatrixG(i, j, x, y)
11:             Gelement ← SumAllMatricesInBuffer(Gbuffer)
12:             Helement ← SumAllMatricesInBuffer(Hbuffer)
13:             G[ii : ii + 2][jj : jj + 2] ← Gelement
14:             H[ii : ii + 2][jj : jj + 2] ← Helement
15: procedure GHMATECD_SING_DE
16:     for i := 1, m do
17:         ii := 3(i − 1) + 1
18:         Gelement, Helement ← Sing_de(i)
19:         G[ii : ii + 2][ii : ii + 2] ← Gelement
20:         H[ii : ii + 2][ii : ii + 2] ← Helement
21: procedure GHMATECD
22:     Ghmatecd_Nonsingd()
23:     Ghmatecd_Sing_de()
```

The `Ghmatecd_Nonsingd` routine can be implemented as a CUDA kernel. In a CUDA block, $g \times g$ threads are created to compute in parallel the two nested loops in lines 2 and 3, allocating spaces in the shared memory to keep the matrix buffers `Hbuffer` and `Gbuffer`. Since these buffers contain matrices of size 3×3, nine of these $g \times g$ threads can be used to sum all matrices, because one thread can be assigned to each matrix entry, unless $g < 3$. Note that g is also upper-bounded by the amount of shared memory available in the GPU. Launching $m \times n$ blocks to cover the two nested loops in lines 2 to 3 will generate the entire H and G without the `Sing_de` part. The `Ghmatecd_Sing_de` routine can be parallelized with a simple OpenMP `Parallel for` clause, and it will compute the remaining H and G.

4. Methods

Matrix norms were used to assert the correctness of our results. Let $A \in \mathbb{C}^{m\times n}$. [Watkins 2004] defines matrix 1-norm as:

$$\|A\|_1 = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}| \tag{5}$$

Table 1. Data experiment set

Number of Mesh elements	240	960	2160	4000
Number of Boundary elements	100	400	900	1600

All norms have the property that $\|A\| = 0$ if and only if $A = 0$. Let u and v be two numerical algorithms that solve the same problem, but in a different way. Now let y_u be the result computed by u and y_v be the result computed by v. The *error* between these two values can be measured computing $\|y_u - y_v\|$. The error between CPU and GPU versions of H and G matrices was computed by calculating $\|H_{cpu} - H_{gpu}\|_1$ and $\|G_{cpu} - G_{gpu}\|_1$. An automated test check if this value is bellow 10^{-4}.

Gfortran 5.4.0 and CUDA 8.0 were used to compile the application. The main flags used in Gfortran were `-Ofast -march=native -funroll-loops -flto`. The flags used in CUDA nvcc compiler were: `-use_fast_math -O3 -Xptxas --opt-level=3 -maxrregcount=32 -Xptxas --allow-expensive-optimizations=true`.

For experimenting, there were four data samples as shown in Table 1. The application was executed for each sample using the original code (serial implementation), the OpenMP version and the CUDA and OpenMP together. All tests but the sequential set the number of OpenMP threads to 4. The machine used in all experiments had an AMD A10-7700K processor paired with a GeForce GTX980[1].

Before any data collection, a warm up procedure is executed, which consists of running the application with the sample three times without getting any result. Afterward, all experiments were executed 30 times per sample. Each execution produced a file with total time elapsed, where a script computed averages and standard deviations for all experiments.

GPU total time was computed by the sum of 5 elements: (1) total time to move data to GPU, (2) launch and execute the kernel, (3) elapsed time to compute the result, (4) time to move data back to main memory, (5) time to compute the remaining H and G parts in the CPU. The elapsed time was computed in seconds with the OpenMP library function `OMP_GET_WTIME`. This function calculates the elapsed wall clock time in seconds with double precision. All experiments set the Gauss Quadrature Points to 8.

5. Results

The logarithmic scale graphic at Figure 1 illustrates the results. All points are the mean of the time in seconds of 30 executions as described in Methodology. The average is meaningful as the maximum standard deviation obtained was 2.6% of the mean value.

The speedup acquired in the 4000 mesh elements sample with OpenMP and CUDA+OpenMP with respect to the sequential algorithm are 2.7 and 107 respectively. As a conclusion, the presented strategy paired with GPUs can be used to accelerate the overall performance of the simulation for a large number of mesh elements. This is a consequence of parallelizing the construction of both matrices H and G, and the calculations in the `Nonsingd` routine. Notice that there was a performance loss in the 260

[1]Thanks to NVIDIA for donating this GPU.

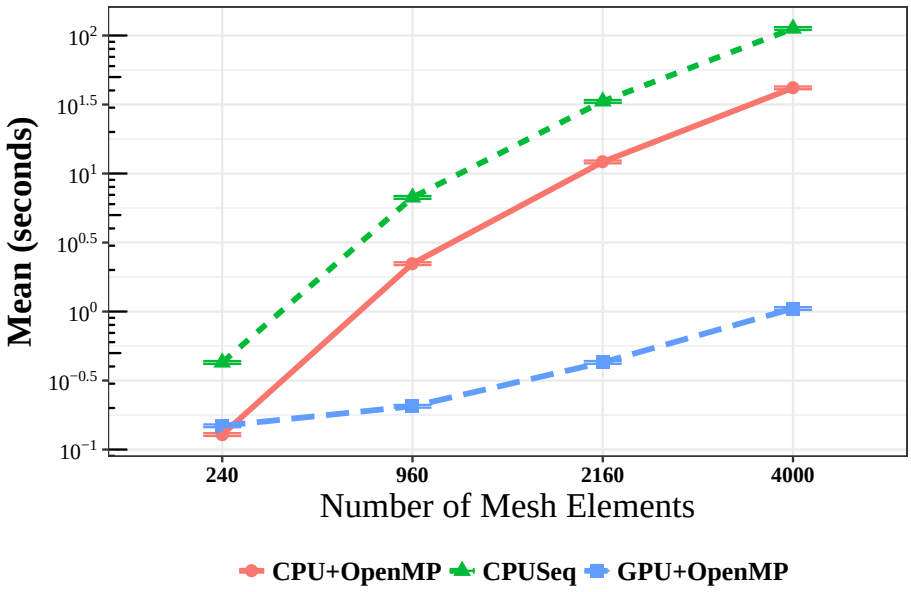


Figure 1. Time elapsed by each implementation in logarithm scale

sample between OpenMP and CUDA+OpenMP, this was caused by the high latency between CPU-GPU communication, thus the usage of GPUs may not be attractive for small meshes.

6. Future Work

There are issues related to the g described in Algorithm 2. Detailed studies are required to determine the exact value of g that provides a good relation between precision and performance. Also, better ways to compute the sum in lines 11-12 of Algorithm 2 may increase performance. The usage of GPUs for the singular case can also be analyzed.

References

Ascher, U. M. and Greif, C. (2011). *A first course on numerical methods*. SIAM.

Carrion, R. (2002). *Uma Implementação do Método dos Elementos de Contorno para problemas Viscoelastodinâmicos Estacionários Tridimensionais em Domínios Abertos e Fechados*. PhD thesis, Universidade Estadual de Campinas.

Dominguez, J. (1993). *Boundary elements in dynamics*. Wit Press.

GNU. Gnu binutils. `https://www.gnu.org/software/binutils/`. Accessed: 2017-05-08.

Katsikadelis, J. T. (2016). *The Boundary Element Method for Engineers and Scientists: Theory and Applications*. Academic Press.

Patterson, D. A. and Hennessy, J. L. (2007). Computer organization and design. *Morgan Kaufmann*.

Watkins, D. S. (2004). *Fundamentals of matrix computations*, volume 64. John Wiley & Sons.

Avaliação da Migração Vertical na Amazon Web Services

Matheus Filipe F. L. da Costa[1], **Luan Teylo**[1], **Lúcia M. A. Drummond**[1]

[1]Instituto de Computação – Universidade Federal Fluminense (UFF)
Caixa Postal – 24210-346 – Niterói – RJ – Brasil

{matheusfilipe, luanteylo, lucia}@ic.uff.br

Abstract. *Elasticity is one of the key concepts in cloud computing. It allows a dynamic resource dimensioning to serve different application demands. Vertical migration is a tool that can change the characteristics (processing power, memory, etc.) of a previously allocated virtual machine to better serve a new demand. Although this tool is available in the majority of public cloud providers, to the best of our knowledge, there are no related works in literature that analyze its use in high performance computing. This work presents initial efforts to evaluate the use of this tool in the context of these applications.*

Resumo. *Elasticidade é um dos conceitos chaves na computação em nuvem, pois permite a adaptação dinâmica de recursos para atender aos diferentes requisitos das aplicações. A migração vertical é uma ferramenta que possibilita a alteração das características (poder de processamento, memória, etc.) de uma máquina virtual previamente alocada para melhor atender uma nova demanda. Embora esse mecanismo esteja presente na maioria dos provedores de nuvem pública, não há trabalhos relacionados na literatura que avaliam o uso desse recurso aplicado a computação de alto desempenho. Este trabalho apresenta os esforços iniciais de avaliação do uso deste recurso no contexto destas aplicações.*

1. Introdução

A elasticidade é um dos conceitos chave da computação em nuvem. Em um cenário no qual os recursos alocados não são suficientes para atender a demanda da aplicação, seja por variações de carga ou por estimativas incorretas, o mecanismo de elasticidade é capaz de adaptar o ambiente alocando mais recursos. Além disso, o uso de elasticidade descarta a necessidade de requisitar previamente uma quantidade maior de poder computacional do que a aplicação realmente necessita (*over-provisioning*), para atender a possíveis variações de carga ao longo da execução. Dessa forma, o cliente minimiza gastos, pois só paga pelo que foi efetivamente utilizado, e diminui o tempo de execução de suas aplicações.

Duas abordagens distintas de elasticidade em nuvens computacionais são encontradas na literatura: a elasticidade horizontal e a elasticidade vertical. Na elasticidade horizontal a quantidade de máquinas virtuais (MV) pode ser alterada em qualquer ponto da execução, ou seja, as MVs são adicionadas (ou removidas) conforme a demanda da aplicação. Neste modelo, uma MV é vista como uma unidade de recurso computacional e suas características (processamento, memória, rede, etc.) são previamente definidas pelo provedor, que atribui um tipo e um preço a cada uma dessas máquinas. Por exemplo, na

Amazon AWS[1] é possível encontrar desde máquinas de uso geral, como a MV de tipo 't2.large' que possui 2 VCPUs[2], 16 GBs de memória e custa $0,2 a hora, até máquinas otimizadas para computação, como é o caso da 'c4.8xlarge' que tem 36 VCPUs, 60 GBs de memória, custando $1,591/hora.

Já na elasticidade vertical, o redimensionamento é direcionado às MVs já alocadas. Nessa abordagem, os recursos atribuídos a uma MV, como tamanho da memória, poder de processamento e espaço de armazenamento, são alterados em tempo de execução, sem a necessidade de interrupção da máquina ou das aplicações. As vantagens e as limitações dessas abordagens em relação a execução de aplicações HPC são discutidas em [Galante et al. 2016]. Segundo os autores, aplicações científicas podem se beneficiar da capacidade de redimensionamento vertical das nuvens computacionais, melhorando o desempenho e minimizando os custos de execução.

Embora vários trabalhos na literatura abordem a utilização de ambientes de nuvem na execução de aplicações HPC [Gupta e Milojicic 2011, Mauch et al. 2013, Gupta et al. 2013, Lorido-Botran et al. 2014, Galante e Bona 2014, Galante et al. 2016], o foco principal da maioria dos estudos tem sido a elasticidade horizontal e pouco se discute sobre o uso da elasticidade vertical. Isso provavelmente é um reflexo direto da falta de suporte dado pelos grandes provedores de nuvem da atualidade a esses mecanismos, conforme evidenciado no levantamento realizado em [Galante et al. 2016], onde os autores apontam que dos 11 provedores avaliados, apenas 3 apresentaram soluções verticais (Profitbricks, CloudSigma e ElasticHosts). Assim, embora os benefícios da elasticidade vertical existam, como mostrado em [Galante e Bona 2014], na prática as aplicações HPC não fazem uso desse recurso por falta de ferramentas e suporte.

Existe, porém, uma forma de verticalização de recursos que é oferecida pela maioria dos provedores, chamada aqui de migração vertical. A migração vertical tem esse nome, pois permite que o tipo atribuído a uma MV já alocada seja alterado a qualquer momento, ou seja, é possível migrar uma MV para um outro tipo de máquina, com mais ou com menos recursos. A migração vertical não é semelhante a elasticidade vertical, pois não permite que recursos específicos, como quantidade de memória e número de núcleos, sejam adicionados (ou removidos) e requerem a interrupção da execução das MVs e, consequentemente, das aplicações. O processo de migração é dividido em três etapas: (i) pausa, que interrompe a execução da máquina, colocando-a em um estado de espera; (ii) migração, que altera o tipo associado a máquina; e (iii) *resume*, que retoma a execução da máquina com a nova configuração de recursos. Essas três etapas são executadas em sequência e, como será apresentado na Seção 2, demandam diferentes tempos para serem concluídas.

Na literatura relacionada não há proposta de avaliação do mecanismo de migração vertical em relação às aplicações HPC. Assim, esse trabalho representa os esforços iniciais para avaliar a viabilidade do uso deste mecanismo em relação a essa classe de aplicações. O objetivo é verificar se há um ganho de desempenho e/ou custo ao utilizar esse mecanismo. Além disso, também serão avaliados os *overheads* decorrentes deste processo. Para isso, foi utilizado o modelo de execução de aplicações de memória compartilhada

[1]https://aws.amazon.com/pt/ec2/pricing/on-demand/

[2]VCPU é uma unidade definida pela AWS que determina o número de cores virtuais que um determinado tipo de máquina tem.

apresentado em [Galante e Bona 2014], no qual a requisição de recursos é feita dentro da própria aplicação. Esse modelo permite que a migração vertical seja facilmente realizada, uma vez que o desenvolvedor pode determinar pontos de execução no qual o processamento parcial seja salvo e a migração seja realizada. Os testes foram executados em um dos principais provedores de nuvem, o Amazon AWS, que é largamente utilizada tanto no meio acadêmico quanto empresarial.

2. Metodologia

No modelo de execução utilizado [Galante e Bona 2014], a aplicação é responsável por requisitar os recursos durante sua execução, ou seja, fica a cargo do programador definir a quantidade de recursos necessários para executar determinadas partes do código. Assim, diferente do modelo tradicional de aplicações elásticas, no qual um parâmetro é utilizado como gatilho do redimensionamento (como por exemplo o número de acessos simultâneos a um servidor web), neste a aplicação deve apresentar uma variação de carga previamente conhecida.

Neste trabalho, o ambiente de execução é composto por duas entidades principais: (i) o *Controlador*, responsável pelo gerenciamento das MVs e da aplicação; e (ii) a *Aplicação*, que utiliza os recursos computacionais da nuvem. O *Controlador* foi implementado utilizando a linguagem de programação Python 2.7, e o gerenciamento do ambiente foi feito através da API Boto3 [3] disponibilizada e mantida pela própria AWS. A API permite que uma aplicação externa inicie e controle a execução das MVs alocadas na AWS. Além de coordenar a execução das máquinas, o *Controlador* também é responsável por enviar o comando de execução das aplicações e realizar o processo de migração vertical quando requisitado.

A Figura 1 representa uma típica execução desse modelo. Como pode ser visto, após alocar e iniciar uma MV, o *Controlador* inicia a *Aplicação* e, em seguida, aguarda o recebimento de requisições de recursos. O processo de migração é iniciado assim que essa requisição é recebida. Para isso, o *Controlador* interrompe a execução da máquina, modifica o seu tipo e, por fim, retoma a execução da máquina e da *Aplicação*.

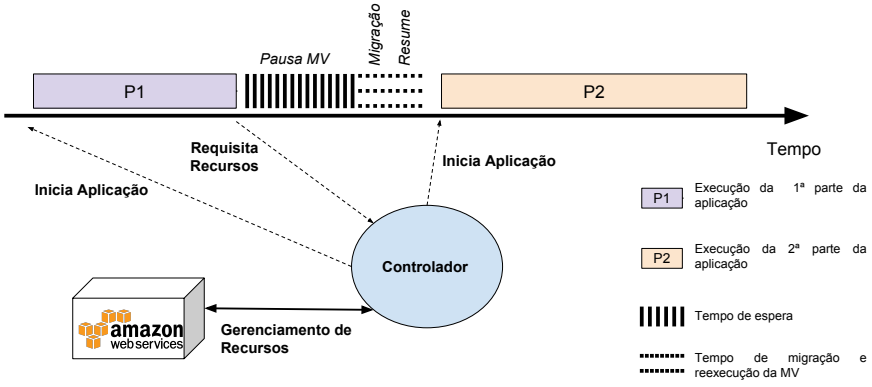


Figura 1. Modelo de execução de uma aplicação auto-escalável.

[3]https://boto3.readthedocs.io

Para avaliar as execuções foi utilizado um algoritmo de multiplicação de matrizes quadradas, implementado em C++ e paralelizado com OpenMP [Dagum e Menon 1998]. Essa aplicação foi escolhida pois é facilmente adaptável para o modelo de execução descrito anteriormente, já que uma variação de carga pode ser simulada através da alteração no tamanho da matrizes. Para simular a demanda de recursos o algoritmo foi dividido em duas etapas. Na primeira, duas matrizes de tamanho n são multiplicadas. Já na segunda, o tamanho das matrizes é modificado para $2n$ e uma nova multiplicação é realizada. Como o algoritmo tem complexidade assintótica de $O(N^3)$, essa variação no tamanho do problema reflete diretamente no tempo total de execução. Dessa forma, é possível simular um aumento na demanda por poder de processamento e memória.

3. Resultados Experimentais

Para avaliar a migração vertical oferecida pela AWS, foram realizados vários testes práticos, que consistiam basicamente em alocar um determinado número de máquinas e executar o processo de migração. O objetivo foi mensurar os tempos decorrentes deste processo para determinados tipos de máquinas. A Tabela 1 apresenta os resultados deste experimento, com a seguinte divisão nos tempos: tempo de criação, que é o período que uma nova máquina leva para ficar disponível; tempo de parada, que é o intervalo de tempo necessário para que a máquina seja alterada do estado de execução (*running*) para o estado de parada (*stopped*); e o tempo de migração/*resume*, que diz respeito a período de modificar o tipo e retomar a execução da MV.

Além dos tempos relacionados ao processo de migração, também são apresentadas os tipos de MV utilizados na avaliação, onde a primeira coluna apresenta a máquina inicial e a coluna seguinte a máquina pós migração. A última coluna da Tabela 1 apresenta o tempo total do processo de migração (tempo de parada mais o tempo de migração/*resume*) e os tempos médios com desvio padrão são apresentados na última linha. Os resultados apresentados são valores médios provenientes de uma bateria de 5 execuções realizadas em intervalos de 2 horas cada. Como pode ser visto, o tempo médio de todo o processo de migração é de 95,75 segundos, onde a maior parte do processo é gasta na operação de tempo de parada (70,76 segundos, em média), que é obrigatório para realização da migração. As máquinas das famílias 't2' e 'm4' foram escolhidas para essa avaliação pois possuem poder de processamento e memória suficientes para o algoritmo executado. As especificações de todas as MVs utilizadas neste trabalho são encontradas em *https://aws.amazon.com/pt/ec2/instance-types/*.

Tabela 1. Avaliação dos tempos associados ao processo de migração vertical.

Máquina Inicial	Máquina Final	Processo de Migração			Tempo Total(s)
		Tempo Criação (s)	Tempo Parada (s)	Tempo Migração/*resume* (s)	
t2.micro	t2.medium	39,93	69,89	24,48	94,37
t2.micro	t2.large	36,67	70,64	24,1	94,74
t2.micro	t2.xlarge	36,67	69,46	24,78	94,24
t2.micro	m4.large	35,90	71,02	24,98	96,00
m4.large	m4.xlarge	36,30	72,38	24,44	96,82
m4.large	m4.2xlarge	36,83	71,21	27,15	98,36
Média		$37,05 \pm 1,45$	$70,76 \pm 1,03$	$24,98 \pm 1,10$	$95,75 \pm 1,62$

Nota-se que o processo de migração/*resume* leva menos tempo que o processo de criação de uma MV: 37,05 e 24,98 segundos, respectivamente. Assim, um possível cenário no qual a migração vertical poderia ser explorada, é na execução de aplicações com longos períodos de execução, nas quais a demanda por recursos varia periodicamente. Um exemplo desse tipo de aplicação são os chamados *workflows* científicos [Deelman et al. 2009]. Nestas aplicações as máquinas poderiam ser colocadas no estado de suspensão ao invés de terminadas, como é tipicamente feito [Coutinho et al. 2014], e o mecanismo de migração poderia ser acionado sempre que novos recursos fossem necessários. Vale ressaltar que isso é válido do ponto de vista financeiro, pois o cliente não é cobrado pelas máquinas que estão em estado de suspensão.

Na Tabela 2, os resultados referentes às execuções do algoritmo de multiplicação de matrizes são apresentados. Essas execuções foram realizadas variando o tamanho da entrada e o tipo das MVs. Para avaliar o mecanismo de migração, foram definidos três cenários. O primeiro, utiliza uma máquina com um menor poder de processamento para executar toda a aplicação (multiplicação da matriz de tamanho n seguida da multiplicação da matriz de tamanho $2n$), e a migração vertical não é executada. Já no segundo cenário, uma máquina com maior capacidade de processamento é usada durante toda a execução sem o uso de migração. Por fim, a migração vertical é utilizada.

Como pode ser visto, ao utilizar a migração vertical os tempos de execução são consideravelmente menores em relação ao uso exclusivo das máquinas de menor poder de processamento, tendo no entanto um aumento no custo financeiro. Esse resultado é esperado, já que a parte da aplicação que demanda mais tempo (multiplicação da maior matriz) é realizada na MV com mais recursos computacionais, o que claramente minimiza o tempo total de execução, mesmo considerando o tempo gasto no processo de migração. Porém, ao analisar o tempo de execução e o custo financeiro decorrentes da execução realizada exclusivamente na máquina com maior poder de processamento, vemos que, para os testes realizados, o processo de migração não foi capaz de minimizar os custos de execução. Isso ocorreu neste exemplo, porque, com a migração, o custo para usar a máquina de maior porte foi equivalente ao uso sem migração com a mesma máquina. A AWS aloca máquinas por unidade de hora e nos testes apresentados as máquinas de maior custo foram usadas pelo mesmo número de horas (com e sem migração). Ou seja, o tempo gasto na primeira parte da aplicação, utilizando a máquina menor, não permitiu a redução de horas na execução da segunda parte.

Tabela 2. Execuções do algoritmo de multiplicação de matrizes com, e sem, a operação de migração vertical.

Tamanho Entrada	Sem Migração			Com Migração				
	VM	Tempo Total Execução (min)	Custo Total ($)	Tempo Migração (min)	Tempo Execução (min)	Custo MV1 ($)	Custo MV2 ($)	Custo Total ($)
2000 - 4000	t2.medium	12,41	0,05	0.78	7.84	0,05	0,19	0,24
2000 - 4000	t2.xlarge	6,85	0,19					
5000-10000	m4.2xlarge	76,69	0,80	0.53	45.91	0,40	0,80	1,20
5000-10000	m4.4xlarge	39,79	0,80					
6000-12000	m4.4xlarge	68,57	1,60	0.78	29.74	0,80	3,20	4,00
6000-12000	m4.16xlarge	23,91	3,20					
7000-14000	m4.4xlarge	107,48	1,60	0.54	45.97	0,80	3,20	4,00
7000-14000	m4.16xlarge	37,15	3,20					
14000-28000	m4.4xlarge	1312,41	17,60	0.52	582.14	1,60	28,80	30,40
14000-28000	m4.16xlarge	517,23	28,80					

4. Conclusão

Neste trabalho o mecanismo de migração vertical presente na Amazon Web Services foi avaliado. Os resultados mostraram que a viabilidade do uso deste mecanismo, em relação as aplicações HPC, depende da divisão correta da carga do algoritmo executado. Portanto, em um trabalho futuro deverá ser criado um modelo de apoio a decisão ao programador, que define as etapas da execução nas quais a migração vertical deve ser realizada, e quais os respectivos recursos devem ser requisitados a fim de minimizar o custo e/ou o tempo de execução.

Referências

Coutinho, R., Drummond, L., Frota, Y., de Oliveira, D., and Ocaña, K. (2014). Evaluating grasp-based cloud dimensioning for comparative genomics: A practical approach. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 371–379.

Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55.

Deelman, E., Gannon, D., Shields, M., and Taylor, I. (2009). Workflows and e-science: An overview of workflow system features and capabilities. *Future Gener. Comput. Syst.*, 25(5):528–540.

Galante, G. and Bona, L. C. E. (2014). Supporting elasticity in openmp applications. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 188–195.

Galante, G., Erpen De Bona, L. C., Mury, A. R., Schulze, B., and da Rosa Righi, R. (2016). An analysis of public clouds elasticity in the execution of scientific applications: a survey. *Journal of Grid Computing*, 14(2):193–216.

Gupta, A., Kale, L. V., Gioachin, F., March, V., Suen, C. H., Lee, B. S., Faraboschi, P., Kaufmann, R., and Milojicic, D. (2013). The who, what, why, and how of high performance computing in the cloud. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 1, pages 306–314.

Gupta, A. and Milojicic, D. (2011). Evaluation of hpc applications on cloud. In *2011 Sixth Open Cirrus Summit*, pages 22–26.

Lorido-Botran, T., Miguel-Alonso, J., and Lozano, J. A. (2014). A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592.

Mauch, V., Kunze, M., and Hillenbrand, M. (2013). High performance cloud computing. *Future Gener. Comput. Syst.*, 29(6):1408–1416.

Implementação e avaliação de co-processadores para Ray-Tracing em FPGA usando HLS

Adrianno A. Sampaio[1], **Alexandre S. Nery**[1]

[1]Departamento de Informática e Ciência da Computação
Instituto de Matemática e Estatística
Universidade do Estado do Rio de Janeiro (UERJ), Brasil

adriannosampaio@gmail.com,anery@ime.uerj.br

Abstract. *The creation of photorealistic images is the main objective of Computer Graphics. However, fast solutions, such as Rasterization, do not model the illumination precisely and, because of that, do not produce high-fidelity images. On the other hand, realistic solutions such as Ray-Tracing possesses high computational complexity, taking, in general, a long execution time. In this paper, the computational complexity problem of the Ray-Tracing is analized through the implementation of accelerators in FPGA, analysing possible code parts that could be executed in specialized hardware, as well as the communication method between that and the host processor.*

Resumo. *A produção de imagens foto-realistas é o principal objetivo da computação gráfica. Porém, as soluções mais rápidas, como a rasterização, não modelam a iluminação de forma precisa e, por isso, não geram imagens com alta fidelidade. Soluções mais realistas, por outro lado, como o algoritmo de Ray-Tracing, possuem uma alta complexidade de computação, levando, em geral, a um longo tempo de execução. Neste artigo, o problema da complexidade do algoritmo de Ray-Tracing é abordado através da implementação e avaliação de aceleradores em FPGA, analisando possíveis trechos do código que poderiam ser executados em uma arquitetura especializada, bem como a forma de comunicação entre a mesma e o processador hospedeiro.*

1. Introdução

Ao contrário dos tradicionais algoritmos de rasterização, que são baseados em técnicas de iluminação local, o *Ray-Tracing* é um algoritmo de iluminação global, que simula a forma como raios de luz viajam pelo ambiente e interagem com os objetos nele, podendo ser refletidos pela superfície de um objeto, absorvidos por ele ou transmitidos por dentro dele [Glassner 1989, Shirley and Marschner 2002, Hall and Greenberg 1983]. Por ser um algoritmo baseado em modelos físicos de transporte de luz é capaz de gerar imagens com maior grau de realismo a partir de um mundo virtual tridimensional [Pharr and Humphreys 2010]. Porém, trata-se de um algoritmo de alto custo computacional, que exige um cálculo exaustivo de interseção de todas as combinações raio-objeto da cena. Por isso, o uso do algoritmo de *Ray-Tracing* geralmente limita-se à renderização de animações 3-D [Christensen et al. 2006], por se tratar de um processo de renderização sem necessidade de interatividade, de forma que os quadros da animação podem ser gerados e agrupados posteriormente.

Com os avanços recentes na tecnologia de miniaturização de semicondutores e o crescimento de poder de processamento de arquiteturas multi-núcleo, como as que se encontram em unidades de processamento gráfico modernas, já é possível sintetizar imagens em tempo real usando o algoritmo de Ray-Tracing [Budge et al. 2008]. No entanto, tais arquiteturas são ineficientes no tocante ao consumo de energia [Betkaoui et al. 2010], pois são otimizadas para computação em fluxo de dados, ideal para processar algoritmos de iluminação local e aplicações de paralelismo de dados. As arquiteturas reconfiguráveis por outro lado, têm mostrado uma relação de eficiência melhor que a de muitos sistemas de computação [Blott 2016]. As FPGAs (*Field-Programmable Gate Arrays*) são uma classe de circuitos reconfiguráveis que podem ser usadas para projetar arquiteturas e sistemas dedicados à execução eficiente de uma aplicação qualquer.

Neste artigo foi analisado como solução para aprimorar o desempenho do algoritmo de *Ray-Tracing* o desenvolvimento de co-processadores para cálculo de interseções raio-objeto em arquiteturas reconfiguráveis (FPGAs) da família Zynq. Este trabalho é um estudo inicial da aplicação na arquitetura Zynq, que possui uma parte programável e um processador ARM interno, que coordena a execução do coprocessador em questão, diferentemente de trabalhos anteriores nos quais a aplicação é implementada completamente na FPGA [Park et al. 2008, Nery et al. 2010], o que impossibilita uma comparação justa no atual estágio de desenvolvimento. Tradicionalmente, a programação das FPGAs é feita através de uma linguagem de definição de hardware (HDL - *Hardware Definition Language*) e o processo de compilação/mapeamento é chamado de Síntese de Circuitos [Wilson 2011]. Porém, neste projeto, foi utilizado um compilador de síntese de alto nível (*High-Level Synthesis - HLS*) da Xilinx, chamado *Vivado HLS*, que traduz um programa descrito em C/C++ para uma arquitetura RTL (*Register Transfer Level*) descrita em VHDL ou Verilog [Xilinx 2017]. O objetivo é avaliar o desempenho, o consumo de área e o consumo de energia do co-processador gerado por HLS.

O trabalho está organizado em três seções. A Seção 2 introduz brevemente o algoritmo de *Ray-Tracing*, aborda os co-processadores propostos e as opções de interface de comunicação entre o processador e a lógica programável da FPGA. Na Seção 3 serão descritos os resultados experimentais de desempenho computacional, custo de área e dissipação de potência das soluções propostas. Por fim, a Seção 4 conclui este trabalho e apresenta algumas idéias para trabalhos futuros.

2. Co-Processadores para Ray-Tracing

O algoritmo básico de *Ray-Tracing* funciona de forma simples: para cada pixel de uma câmera virtual é lançado um raio e, em seguida, é calculado se existe uma interseção entre este e algum objeto da cena virtual. Se houver interseção e ela for mais próxima da câmera do que a última interseção calculada, então o pixel assume a cor do ponto atingido no objeto.

Como as funções mais computacionalmente custosas do algoritmo são as que calculam as interseções raio-objeto [Kay and Kajiya 1986], ou seja, as que determinam qual objeto é visível para um raio, a primeira função a ser considerada para implementação em *hardware* foi a interseção Raio-Esfera. O algoritmo pode se resumir ao cálculo das raízes de uma equação de segundo grau, o que o torna um ótimo exemplo por possuir diversas operações de ponto flutuante, incluindo uma operação de raiz quadrada, que é mais lenta do que outras operações mais simples. A função de interseção Raio-Esfera implementada no *Vivado HLS* pode ser vista na Figura 1.

```
void ray_sphere(float* sx, float* sy, float* sz, float* r,  // Esfera
                float* ox, float* oy, float* oz,            // Origem Raio
                float* dx, float* dy, float* dz,            // Direção Raio
                float* res0, float* res1){                  // Resultado

    float t0, t1;
    float tmp_x = *ox - *sx,
          tmp_y = *oy - *sy,
          tmp_z = *oz - *sz;

    float a = ((dx)(dx)) + ((*dy)(dy)) + ((*dz)(*dz));
    float b = 2.0f * ((tmp_x * (*dx)) + (tmp_y * (*dy)) + (tmp_z * (*dz)));
    float c = ((tmp_x*tmp_x) + (tmp_y*tmp_y) + (tmp_z*tmp_z)) - ((r)(*r));
    float disc = (b*b) - (4.0f*a*c);

    if( disc < 0.0f ){
        t0 = t1 = -1.0f;
    }
    else {
        float e = sqrt(disc);
        float denom = 2.0f * a;
        t0 = ((-b) - e) / denom;
        t1 = ((-b) + e) / denom;
    }
    *res0 = t0;
    *res1 = t1;
}
```

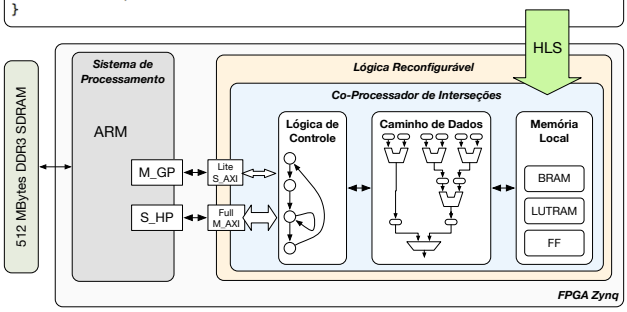


Figura 1. Função de interseção Raio-Esfera em HLS e a Arquitetura do Co-Processador gerado na FPGA.

As opções de aceleradores desenvolvidos implementam o mesmo algoritmo. A diferença está na interface de comunicação, que se dá através do protocolo AXI de comunicação [Xilinx 2012] (*Advanced eXtensible Interface*), que faz parte do padrão AMBA (*Advanced Microcontroller Bus Architecture*) da ARM. Esse protocolo de comunicação permite que circuitos mapeados em FPGA sejam acessíveis ao processador. A FPGA Zynq foi escolhida pela possibilidade de executar o algoritmo de *Ray-Tracing* no processador ARM embarcado enquanto os trechos mais críticos do programa executam em hardware dedicado na parte programável da arquitetura. O compilador HLS traduz a hierarquia de funções C/C++ para uma descrição RTL em VHDL (ou Verilog). Os parâmetros da função raiz na hierarquia de funções definirão as portas de E/S da interface de comunicação, enquanto as operações encontradas na função serão escalonadas para executar em unidades funcionais dedicadas, em paralelo. Operações de controle serão traduzidas em uma lógica de controle por máquina de estados, permitindo que laços de repetição e desvios sejam implementados. Variáveis de memória como vetores e matrizes são geralmente mapeados em blocos de memória embarcados na própria FPGA.

Os três principais tipos de interface do protocolo AXI são: AXI-Lite, AXI (ou *AXI-Full*) e AXI-Stream. A interface AXI-Lite é a mais simples, normalmente utilizada para sinais de controle e baixa vazão de dados, seguida pela AXI-Full utilizada para transporte de rajadas de 256 dados de 32-bits. Por último, a interface AXI-Stream é usada para transporte de um alto volume de dados em fluxos contínuos, sem limite de rajadas curtas. Neste trabalho serão avaliados os co-processadores com interfaces AXI-Lite e AXI-Full, como mostra a Figura 2.

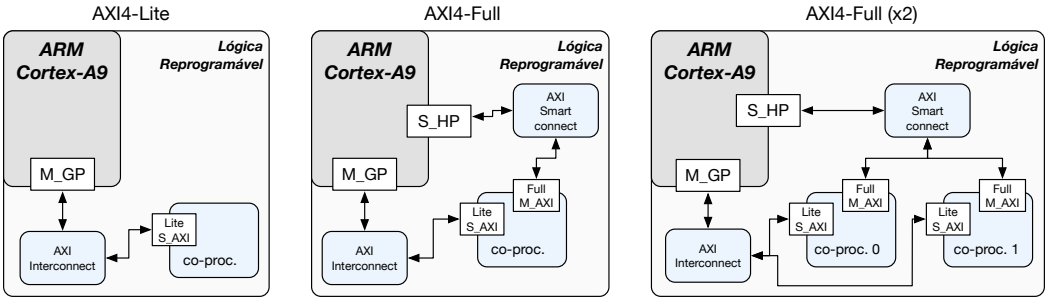


Figura 2. Diferentes arquitetura de comunicação entre o co-processador (acelerador) e o processador ARM, usando interfaces AXI-Lite e AXI-Full.

2.1. Co-processador com interface AXI-Lite

A primeira arquitetura concebida se utiliza totalmente da interface AXI-Lite, e, por ter menos capacidade no envio de dados, realiza apenas o cálculo de interseção entre um raio e uma esfera. Como esta interface é usada em geral para sinais de controle, ela não dispõe da possibilidade de enviar dados em sequência, sendo necessário o uso de múltiplos ciclos de relógio para verificar se o barramento está livre para envio de novos dados. Ela se conecta à porta de propósito geral (*Master General Purpose*) do ARM.

2.2. Co-processador com interface AXI-Full

A principal característica da interface AXI-Full é a possibilidade de envio de rajadas de dados, permitindo que parte do custo de comunicação se sobreponha ao custo de computação. A interface AXI-Lite continua sendo usada, mas apenas para um número consideravelmente menor de dados, como os próprios sinais de controle e os dados do raio. A interface AXI-Full é usada para transferência dos dados de um conjunto de esferas que compõem a cena 3-D. Esta interface é, também, mais rápida do que a AXI-Lite por utilizar uma porta de comunicação de alto desempenho (*Slave High-Performance*) do ARM, que possui um caminho de dados mais rápido entre o processador e a FPGA.

A interface AXI-Full foi utilizada na criação de duas arquiteturas, uma versão com um co-processador (*single-core*, AXI-Full) e uma versão com dois co-processadores (*dual-core*, AXI-Full x2). Na primeira versão, o acelerador recebe através da interface AXI-Full uma rajada de dados de 25 esferas (totalizando 100 valores em ponto flutuante), e retorna, no mesmo endereço base de memória, os dois pontos de interseção de cada esfera (devolvendo 50 valores em ponto flutuante). A segunda versão usa duas instâncias do co-processador raio-esfera, possibilitando o processamento de dois raios em paralelo.

3. Resultados experimentais

Para analisar o desempenho das arquiteturas utilizadas é necessário primeiro estabelecer um parâmetro de comparação, que, neste caso, será a execução do mesmo algoritmo no processador ARM embarcado na FPGA Zynq. Um teste foi feito realizando o cálculo de interseção entre um raio e 25 esferas. A medida de desempenho utilizada se dará em ciclos de relógio do processador ARM por execução. Com esse teste foi possível calcular que o processador ARM precisou de 20.086 ciclos de relógio para a execução destes cálculos de interseção, o que corresponde a aproximadamente 803 ciclos por esfera. Na Figura 4 é

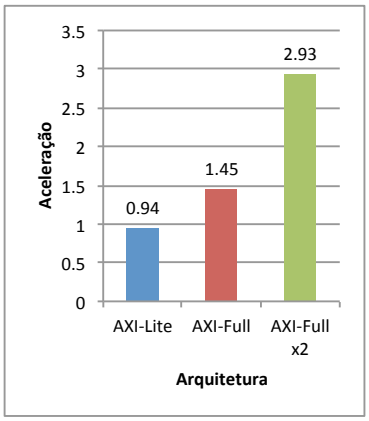


Figura 3. Aceleração.

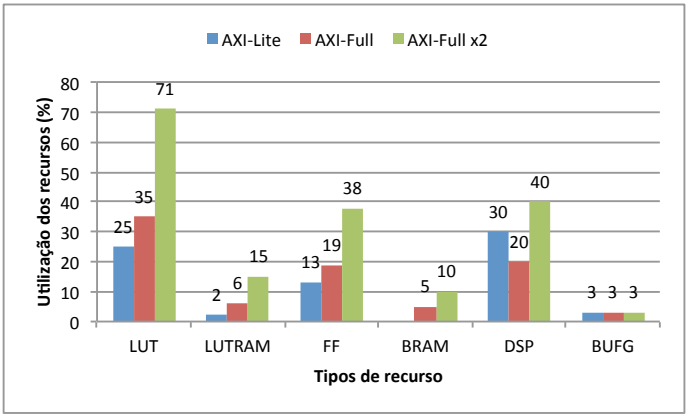


Figura 4. Recursos utilizados da FPGA.

possível observar uma comparação do uso de recursos da FPGA. A medida LUT se refere à porcentagem de uso das Lookup Tables da FPGA e é a medida que mais interfere na possibilidade de escalabilidade do número de núcleos deste projeto na FPGA escolhida, visto que, não é possível adicionar um terceiro núcleo neste caso.

3.1. Co-processador com interface AXI-Lite

Esta arquitetura demonstrou um desempenho inferior ao processador ARM, necessitando de 846 ciclos (contados a partir do ARM) para envio e cálculo da interseção de um raio e uma esfera. A interface AXI-Lite não é capaz de transmitir dados em rajadas. Por isso, dificilmente o tempo de execução sofrerá melhora significativa para um número maior de esferas. Logo, para este algoritmo a interface AXI-Lite não possui um desempenho desejável. Além disso, a arquitetura ocupou 25% das LUTs da *FPGA*.

3.2. Um núcleo co-processador com interface AXI-Full

Seu desempenho foi consideravelmente melhor do que as duas soluções anteriores, utilizando apenas 13.794 ciclos de relógio (contados a partir do ARM) para a operação completa, ou aproximadamente 551 ciclos por esfera, ocupando 35% de LUTs da FPGA e obtendo assim uma aceleração de $1,45$.

3.3. Dois núcleos co-processadores com interface AXI-Full

Foi possível observar nesta arquitetura que não houve grande mudança no número de ciclos consumidos individualmente por cada núcleo em relação à versão com apenas um núcleo, com uma média de 13700 ciclos para a operação completa de interseção com as 50 esferas, ou seja, 274 ciclos por esfera, mostrando assim que replicar a arquitetura possibilita o processamento de 2 raios em paralelo, porém a um custo de área significativamente maior da FPGA, como mostra a Figura 4.

4. Conclusão e Trabalhos Futuros

Com os resultados obtidos é possível perceber que o uso de um co-processador para a operação de interseção raio-esfera implementado em FPGA conseguiu atingir uma aceleração de 1.45 para a arquitetura com interface AXI-Full e de quase 3 na arquitetura AXI-Full com dois núcleos de processamento. Assim, é possível inferir que a adição de novos núcleos no acelerador aumenta o desempenho de forma quase linearmente.

Para trabalhos futuros temos, além da função de cálculo de interseção Raio-Esfera, outras funções que podem ser implementadas na FPGA, como as interseções Raio-Caixa e Raio-ObjetoCSG (outros objetos implementados no *Ray-Tracer*). Outra questão inexplorada no projeto foi o uso da interface de comunicação AXI-Stream, que permite um fluxo constante e ilimitado de dados entre o processador e o acelerador, o que poderia levar a uma melhora ainda maior no tempo de execução, principalmente com um maior volume de dados. Também é possível explorar a possibilidade de uso de mais núcleos de processamento em uma FPGA com maior capacidade.

Referências

Betkaoui, B., Thomas, D. B., and Luk, W. (2010). Comparing performance and energy efficiency of fpgas and gpus for high productivity computing. In *2010 International Conference on Field-Programmable Technology*, pages 94–101.

Blott, M. (2016). Reconfigurable future for hpc. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 130–131.

Budge, B. C., Anderson, J. C., Garth, C., and Joy, K. I. (2008). A straightforward cuda implementation for interactive ray-tracing. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 178–178.

Christensen, P. H., Fong, J., Laur, D. M., and Batali, D. (2006). Ray tracing for the movie 'cars'. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 1–6.

Glassner, A. S. (1989). *An Introduction to Ray Tracing*. Academic Press, Ltd.

Hall, R. and Greenberg, D. (1983). A testbed for realistic image synthesis. *IEEE Computer Graphics and Applications*, 3:10–20.

Kay, T. L. and Kajiya, J. T. (1986). Ray tracing complex scenes. *SIGGRAPH Comput. Graph.*, 20(4):269–278.

Nery, A. S., Nedjah, N., and França, F. M. G. (2010). A parallel architecture for ray-tracing. In *2010 First IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, pages 77–80.

Park, W. C., ho Nah, J., Park, J. S., Lee, K.-H., Kim, D.-S., Kim, S.-D., Park, J. H., Kim, C.-G., Kang, Y.-S., Yang, S.-B., and Han, T.-D. (2008). An fpga implementation of whitted-style ray tracing accelerator. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 187–187.

Pharr, M. and Humphreys, G. (2010). *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition.

Shirley, P. and Marschner, S. (2002). *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., 4th edition.

Wilson, P. (2011). *Design Recipes for FPGAs: Using Verilog and VHDL*. Elsevier Science.

Xilinx (2012). *AXI Reference Guide*. Xilinx.

Xilinx (2017). *Vivado Design Suite User Guide: High Level Syntesis*. Xilinx.

Proposta e Avaliação de uma Rede-em-Chip Programável

João Paulo P. Novais, Matheus A. Souza, Henrique C. Freitas

Grupo de Arquitetura de Computadores e Processamento Paralelo (CArT)
Departamento de Ciência da Computação
Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
Belo Horizonte, Brasil

{joao.novais,matheus.alcantara}@sga.pucminas.br, cota@pucminas.br

Abstract. *Communication between the cores in multi and many-core architectures might be impaired if they do not employ appropriate interconnection strategies. The use of Networks-on-chip (NoCs) appears as an alternative, mitigating the limitations inherent in the size of traditional buses and crossbar switches. However, they must be prepared to handle different communication patterns from parallel applications, without harming the architecture scalability. In this work, we present a programmable NoC hardware design, based on the integration between the System-on-Chip Interconnection Network (SoCIN) and the Network Processor-on-Chip (NPoC). The proposal is to change the SoCIN routing from fixed to dynamic, adding flexibility to the NoC, allowing the use of a wide range of routing protocols.*

Resumo. *A comunicação entre os núcleos de arquiteturas multi e many-core pode ser prejudicada se elas não adotarem estratégias apropriadas para interconexão. O uso das Networks-on-chip (NoCs) surge como alternativa, mitigando as limitações inerentes ao tamanho de barramentos e chaves crossbar tradicionais. Porém, devem estar preparadas para lidar com diversos padrões de comunicação de aplicações paralelas, sem prejudicar a escalabilidade da arquitetura. Neste trabalho, é apresentado um projeto de hardware de uma NoC programável, fundamentada na integração da System-on-Chip Interconnection Network (SoCIN) com o Network Processor-on-Chip (NPoC). A proposta é modificar o roteamento da SoCIN de fixo para dinâmico, acrescentando flexibilidade à NoC e permitindo o uso de um leque maior de protocolos de roteamento.*

1. Introdução

Um dos principais problemas presentes na área da computação diz respeito à demanda por desempenho de aplicações diversas. O limite encontrado nas arquiteturas *single-core*, por restrições no paralelismo em nível de instrução ou elevado consumo de potência, estimulou o desenvolvimento de arquiteturas *multi-core*. Nessas arquiteturas, o processador possui vários núcleos, sendo possível a execução de diferentes *threads* em paralelo. Os núcleos também podem ter características heterogêneas, permitindo que as aplicações paralelas explorem melhor a arquitetura, obtendo mais desempenho. Em um processamento paralelo, geralmente é necessário o compartilhamento dos dados processados, através de memórias compartilhadas, ou a troca de mensagens entre as tarefas, por meio de protocolos de comunicação.

A comunicação dos núcleos de uma arquitetura multi-core pode utilizar barramentos e/ou chaves *crossbar* [Kumar et al. 2005]. Essa abordagem, para arquiteturas *many-core* (elevada quantidade de núcleos), torna-se inviável, pois aumenta-se a quantidade de barramentos, bem como a concorrência no uso. O aumento da resistência no fio e da latência de comunicação também são problemas das abordagens tradicionais, além da maior complexidade de controle [Benini and De Micheli 2002].

No projeto de uma arquitetura *many-core*, as *Networks-on-chip (NoCs)* são propostas como solução para integrar o alto número de núcleos. Uma *NoC* é um sistema mais complexo de interconexão entre núcleos de um único *chip*, que propõe o uso de conexões curtas entre roteadores para formar uma rede de comunicação na qual pacotes de dados são encaminhados. Nesta abordagem, os problemas de latência na comunicação e resistência do fio são resolvidos.

O roteador é o elemento da *NoC* responsável pelas operações de rede e por garantir a comunicação entre os núcleos. Portanto, é relevante propor uma arquitetura de roteador adequada para que um processador *many-core* baseado em *NoC* seja concebido. *NoCs* convencionais são compostas por roteadores dedicados para cada núcleo ou em formato de *multi-clusters* [Freitas et al. 2008]. Sendo assim, para novas topologias, é fundamental a preocupação com os possíveis padrões de comunicação. Há então a necessidade de criar novas arquiteturas de roteador especializadas para cada topologia.

Diante desse contexto, e com a necessidade de se explorar cada vez mais as possíveis melhorias nas *NoCs*, o objetivo deste trabalho é desenvolver uma *NoC* programável utilizando como base o processador *Network Processor-on-Chip (NPoC)*, e a *System-on-Chip Interconnection Network (SoCIN)* [Zeferino and Susin 2003]. Essa arquitetura pretende tornar flexível o roteamento de um processador, permitindo aos desenvolvedores explorar melhor diversos algoritmos de roteamento sem a necessidade de mudar os componentes, apenas programando novos protocolos de roteamento.

O artigo está organizado da seguinte forma. A Seção 2 apresenta os trabalhos correlatos. Na Seção 3 explica-se o projeto proposto, enquanto na Seção 4 são apresentados os resultados. Por fim, na Seção 5 aborda-se as conclusões e trabalhos futuros.

2. Trabalhos Correlatos

Inicialmente, em [Freitas et al. 2006] é apresentado o projeto de um processador de rede *intra-chip* responsável por gerenciar e controlar a comunicação entre os múltiplos núcleos. Esse trabalho é continuado em [Freitas et al. 2008], no qual é proposta uma arquitetura de *NoC* que suporta múltiplos *clusters* de núcleos de processamento através de roteadores programáveis e de topologias reconfiguráveis.

Em [Djemal et al. 2012], os autores propõem um roteador programável para melhorar a eficiência de escalonamento e roteamento de pacotes em uma *NoC*. Os resultados alcançados foram a redução nas contenções em rede, possibilitando um melhor desempenho das aplicações. Já em [Chatrath et al. 2016], foi desenvolvido um roteador programável de alta performance utilizando *Verilog HDL*. A proposta dita por eles para aumentar a eficiência da rede em grandes cargas, é de alocar dinamicamente os *buffers*, tendo um custo de potência de 5mW e aumentando a eficiência do roteador programável.

A implementação de *DRSs (Dynamic Reconfigurable Systems)* possui um custo

alto se tratando de área no *chip* e tempo para configuração de roteamento. Sendo assim, em [Castillo et al. 2015] os autores propuseram uma abordagem de DRS baseada em *NoC* e na lógica do algoritmo *Flexible Direction Order Routing (FDOR)* em diversos cenários, mostrando que é uma solução adequada para o alto custo.

Na Seção 3 é apresentada a arquitetura de uma *NoC* programável baseada em uma topologia *mesh* e um processador de roteamento para gerenciar os pacotes armazenados nos *buffers*. O propósito é o desenvolvimento do *hardware*, em linguagem VHDL da arquitetura de *NoC* flexível, habilitando o uso de diversos protocolos de roteamento definidos pelo desenvolvedor. Há também a possibilidade dos pacotes de rede serem processados, permitindo o gerenciamento dos dados durante o encaminhamento pela *NoC*.

3. Projeto da Arquitetura de NoC Programável

A arquitetura da *NoC* programável proposta é baseada na topologia *mesh* da *NoC SoCIN* [Zeferino and Susin 2003], mantendo as entradas e saídas originais dos roteadores. Utiliza-se o NPoC para gerenciar os pacotes que trafegam na *NoC*.

O *NPoC* compreende uma arquitetura de processador de roteador programável para uma *NoC*, usada para controle de comunicação de múltiplos núcleos. Essa comunicação é feita entre roteadores vizinhos que são interconectadas por uma chave *crossbar* simplificada. O *NPoC* substitui grande parte do roteador da *NoC*, mantendo o controle de fluxo de dados, transmitindo informações de um *buffer* a outro através dos protocolos programados. Os protocolos são desenvolvidos pelo programador e enviados para uma memória RAM de instruções. Essas instruções fazem o controle de entrada e saída dos pacotes de um *buffer*, e o gerenciamento da chave *crossbar*, mapeando os caminhos do remetente ao destino dos pacotes enviados.

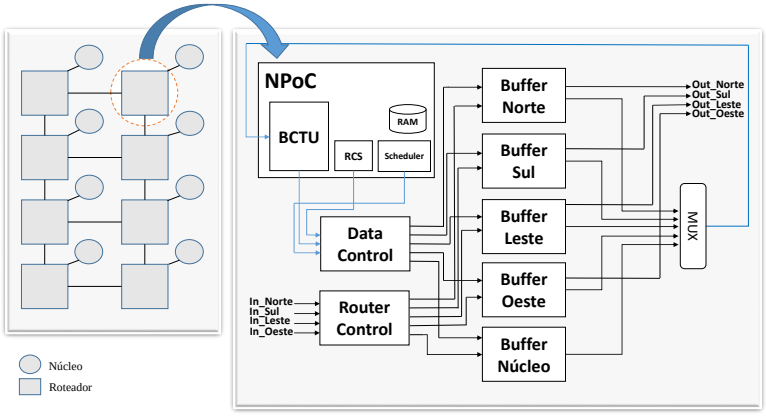


Figura 1. Proposta de Hardware

A *NoC* programável proposta tem topologia *mesh* e um novo bloco de roteador, integrando o *NPoC*. A Figura 1 apresenta uma visão geral da *NoC* proposta, e do seu roteador. Um dos componentes do roteador é o *NPoC*, que altera basicamente duas partes do roteador: a Tabela de Roteamento e o controle dos *Buffers*.

No NPoC, a tabela de roteamento é responsável por mapear as conexões entre os roteadores, controlando o chaveamento dos barramentos. O *NPoC* irá atuar como um

controlador desta tabela, de forma dinâmica e programável. A saída RCS configura a tabela de roteamento, podendo ser alteradas através de instruções descritas pelo programador. Os *Buffers* são responsáveis por transmitir um pacote de um roteador a outro. Para isso é necessário controlar quais pacotes serão enviados, e em qual sequência, evitando problemas como *deadlock* ou sobreposição de instruções.

O Roteador da *NoC* possui um controlador que indica para qual *buffer* será encaminhado o pacote, e posteriormente seu destino. Considerando a topologia *Mesh*, os destinos são: Norte, Sul, Leste, Oeste ou o Núcleo. Contudo, para controle dos pacotes que serão armazenados ou enviados dos *Buffers*, o *NPoC* gerencia de forma dinâmica e programável através do bloco *Buffers and Crossbar Transfer Unit (BCTU)* ilustrado na Figura 1. Este bloco atua como intermediário interconectando os pinos de entrada e saída do *NPoC* com os *buffers*.

A arquitetura compõe vários roteadores interconectados, ligados aos núcleos. Cada roteador possui 5 *buffers*, que são usados para receber os pacotes vindo de outro roteador. O roteador da *NoC* programável é composto por: Controlador de dados da *NoC*, sendo responsável por controlar a sequência de envio dos pacotes ou instruções do *NPoC* para os *Buffers*; Controlador de Pacotes do Roteador, para direcionar os pacotes ao *Buffer* destino; *Buffers*, armazenando ou enviar os pacotes que chegam de outro roteador ou o *NPoC*; e o *NPoC*, que é um processador capaz de controlar entrada e saída dos pacotes do *buffer* e a tabela de roteamento, por meio de um algoritmo pré estabelecido.

A transição dos pacotes entre os roteadores passa por uma interface de rede que empacota os dados na origem e desempacota no destino. Por fim, essa arquitetura é desenvolvida de forma a interconectar todos os núcleos por intermédio dos roteadores, que por sua vez é controlado dinamicamente pelo processador de roteamento (*NPoC*).

4. Resultados

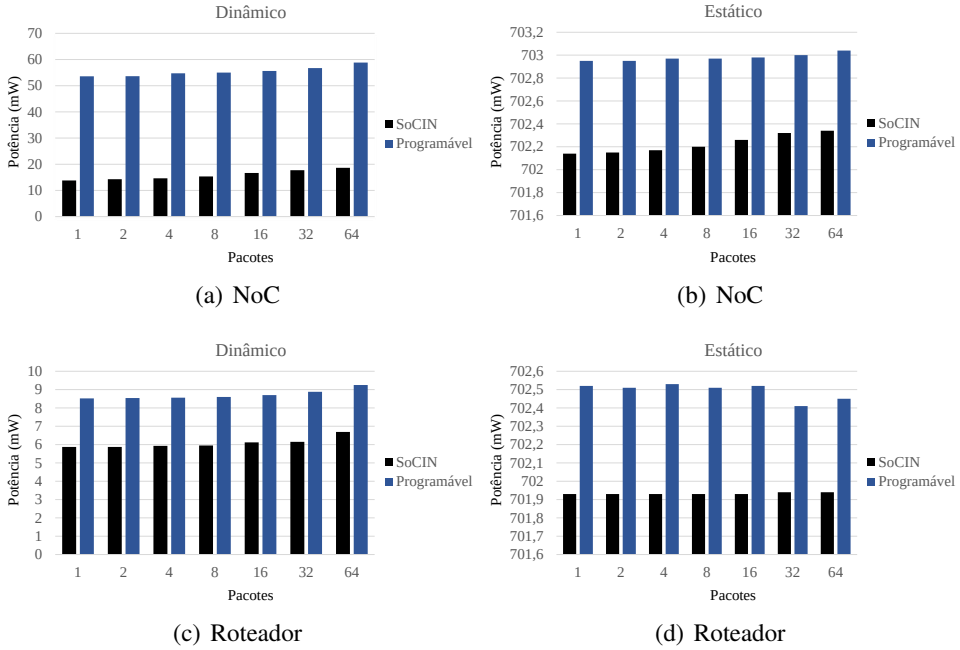
Para a *NoC* proposta, foram coletados o consumo de potência e a quantidade de elementos lógicos após a adição do NPoC no roteador, com o propósito de verificar a viabilidade do uso dessa *NoC*. Utilizou-se o FPGA Cyclone V: 5CGXFC9E6F35C7, comparando o roteador da *NoC SoCIN* com o roteador programável.

Tabela 1. Elementos Lógicos

		Roteador		NoC		
Elementos	Qtde. do FPGA	Programável	SoCIN	Programável	SoCIN	NPoC
CLBs	113560	1281	434	7653	2153	2505
LUTs	113560	536	277	3917	1030	1967
FFs	113560	578	148	2730	675	1460
Pinos	616	362	362	544	544	226

O *NPoC* ocupa boa parte do roteador proposto. Uma análise dos componentes lógicos desse processador, em comparação com uma arquitetura MIPS, é apresentada em [Novais et al. 2016]. O *NPoC* provoca um pequeno aumento no uso de componentes lógicos, como é possível verificar na última coluna da Tabela 1.

A compilação de ambas arquiteturas, *NoC SoCIN* e Programável, foi obtido um *clock* máximo respectivamente de 131,66 MHz e 123,49 MHz (utilizando o mesmo



(a) NoC (b) NoC

(c) Roteador (d) Roteador

Figura 2. Potência consumida pela NoC e pelos Roteadores

FPGA). Após a descrição do *hardware* da *NoC* programável, foi realizada a comparação dos elementos lógicos com a *NoC SoCIN*, cujos valores também são apresentados na Tabela 1. É observado um aumento de 3 vezes mais nos componentes lógicos da *NoC* programável em comparação a *SoCIN*, impactando no consumo de potência.

A análise de potência foi feita através da ferramenta do *PowerPlay Power Analyzer Tool*. Foi utilizado um algoritmo de roteamento simples em uma topologia *Mesh*, que tem o objetivo de enviar os dados de um roteador X para outro Y. Na simulação, diversos pacotes de dados foram enviados para o *buffer* de um roteador, analisando o consumo de potência durante o processamento, ou quando há o envio de um roteador a outro. Na Figura 2 é apresentada a relação desse consumo com a quantidade de pacotes introduzidos na *NoC* (eixo horizontal). A simulação é realizada com a mesma carga de pacotes para o projeto da *NoC* proposta e para a *NoC SoCIN*, sendo analisado separadamente os roteadores, e por conseguinte a estrutura completa da *NoC*.

A Figura 2.a mostra o consumo de potência dinâmico para processar os pacotes das *NoCs*, já a Figura 2.b apresenta o consumo de potência estático das *NoCs*. Na Figura 2.c é apresentado o consumo de potência dinâmico para processar os pacotes nos roteadores, e a Figura 2.d mostra o consumo de potência estático dos roteadores.

Verifica-se um aumento do consumo de potência da *NoC* proposta, devido a adição do roteador que contém o *NPoC*. Entretanto, é mantido a mesma quantidade de ciclos para direcionar o pacote ao *buffer* de destino, além de possibilitar a flexibilidade de configuração dos protocolos de roteamento. Para novas políticas de roteamento, basta programar um novo algoritmo, evitando o desenvolvimento de novas arquiteturas.

5. Conclusão

Neste trabalho foi projetado o *hardware* de uma *NoC* com roteador programável, baseado na junção da *NoC SoCIN* e do processador de rede *NPoC*. O projeto reconfigurável proposto permite maior escalabilidade e flexibilidade em processadores *many-core*, graças a possibilidade de desenvolvimento de algoritmos de roteamento para controlar o fluxo de dados no processador. A principal contribuição deste trabalho é o desenvolvimento do *hardware* proposto. A *NoC* com o roteador programável desenvolvida poderá ser usada em novos projetos de arquiteturas *many-core*, que são tendências para obtenção de alto desempenho. Como trabalhos futuros, propõe-se a redução do consumo de potência estudando posicionamentos dos roteadores programáveis (uma arquitetura híbrida), e o estudo dos algoritmos de roteamento e de qualidade de serviço (QoS), com objetivo de alcançar maior desempenho na comunicação dos núcleos da *NoC* com baixo consumo. Também é sugerida a avaliação de estratégias de paralelismo com diferentes cargas de trabalho no *hardware* proposto.

Agradecimentos

Ao CNPq, FAPEMIG e CAPES pelo suporte no desenvolvimento do trabalho.

Referências

Benini, L. and De Micheli, G. (2002). Networks on chip: a new paradigm for systems on chip design. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 418–419. IEEE.

Castillo, E. V. et al. (2015). Dynamically reconfigurable NoC using a deadlock-free flexible routing algorithm with a low hardware implementation cost. In *Latin American Symposium on Circuits Systems (LASCAS)*, pages 1–4.

Chatrath, A. K., Gupta, A., and Pandey, S. (2016). Design and implementation of high speed reconfigurable NoC router. In *International Conference on Inventive Computation Technologies (ICICT)*, volume 3, pages 1–5.

Djemal, M. et al. (2012). Programmable routers for efficient mapping of applications onto NoC-based MPSoCs. In *Conference on Design and Architectures for Signal and Image Processing*, pages 1–8. IEEE.

Freitas, H., Santos, T., and Navaux, P. (2008). Design of programmable noc router architecture on fpga for multi-cluster nocs. *Electronics Letters*, 44(16):969–971.

Freitas, H. C. et al. (2006). Projeto de um processador de rede intra-chip para controle de comunicação entre múltiplos cores. In *Workshop em Sistemas Computacionais de Alto Desempenho*, pages 3–10. Sociedade Brasileira de Computação.

Kumar, R., Zyuban, V., and Tullsen, D. M. (2005). Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 408–419. IEEE Computer Society.

Novais, J. P. P., Souza, M. A., and Freitas, H. C. (2016). Projeto em VHDL de um processador de rede intra-chip. In *Simpósio em Sistemas Computacionais de Alto Desempenho - Workshop de Iniciação Científica (WSCAD-WIC)*, page 19.

Zeferino, C. A. and Susin, A. A. (2003). Socin: a parametric and scalable network-on-chip. In *Symposium on Integrated Circuits and Systems Design*, pages 169–174.

Avaliação do Consumo Energético da Miniaplicação LULESH em OpenMP com a Arquitetura big.LITTLE

Pedro L. Lima[1], **João V. F. Lima**[1]

[1]Universidade Federal de Santa Maria (UFSM) – Santa Maria – RS – Brasil

{plima,jvlima}@inf.ufsm.br

Resumo. *Este artigo propõe estudar o desempenho da miniaplicação LULESH paralelizada de duas maneiras - versões estática e dinâmica - e avaliar seu consumo energético com diferentes governos de processadores utilizando a arquitetura big.LITTLE. O governo performance mostrou-se mais eficiente nas execuções das duas versões, enquanto o governo powersafe mostrou-se mais eficiente que o governo on_demand na versão estática.*

Abstract. *This article proposes to study the performance of the miniapplication LULESH parallelized in two ways - static and dynamic versions - and evaluate its energy consumption in different governors using the big.LITTLE architecture. The performance governor proved to be more efficient in the execution of both versions, while the powersafe governor showed up to be more efficient than the on_demand governor in the static version.*

1. Introdução

Muitas simulações computacionais requerem modelagem hidrodinâmicas para serem resolvidas. Essas modelagens descrevem o movimento dos materiais relativos uns aos outros quando submetidos a forças. Quantidades massivas de dados devem ser analisadas e processadas com velocidade razoável, e para isso são necessários computadores com *hardwares* suficientemente poderosos para manter a performance concebível. Este processamento demanda um grande custo energético, sendo preocupação fundamental para pesquisas da área.

Uma estratégia estudada para diminuir o custo energético da execução de aplicações é o uso de arquiteturas de baixo consumo, onde os processadores comuns - os processadores x86 - são substituídos por processadores ARM [Nikola Rajovic 2014]. A tecnologia *big.LITTLE* emprega o uso tanto dos processadores convencionais - denominados *big*, projetados para prover máxima performance computacional - quanto os processadores *LITTLE*, projetados para ter máxima eficiência energética [Arm Limited].

Para melhorar o desempenho dessas aplicações, a programação paralela pode ser utilizada. A programação paralela se baseia no uso simultâneo de múltiplos recursos computacionais para resolver um determinado problema, que é dividido em pequenas partes que podem ser resolvidas concorrentemente. Cada parte é repartida novamente em uma série de instruções executadas simultaneamente em diferentes processadores [Blaise Barney 2015].

LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) é uma miniaplicação, versão simplificada de uma aplicação científica real, que pode ser

facilmente adaptada para qualquer modelo de programação paralela. Seu comportamento semelhante a de uma aplicação proporciona uma ferramenta simples e prática para experimentos.

Neste artigo, o consumo de energia e o desempenho da miniapliçação LULESH, na linguagem C++, paralelizada com a biblioteca OpenMP, são avaliados na arquitetura big.LITTLE em diferentes modelos de performance dos processadores [OpenMP Architecture Review Board 2015].

2. Trabalhos Relacionados

Recentemente, foi implementado e avaliado diferentes versões paralelas em OpenMP da miniaplicação LULESH [Pedro Langbecker Lima 2016]. Os resultados foram analisados a partir dos rastros de execução *post mortem* obtidos com a ferramenta Score-P. De todas versões analisadas, percebeu-se um ganho maior de desempenho utilizando diretivas estáticas, e uma semelhança com o tempo sequencial utilizando diretivas dinâmicas [Knüpfer et al. 2012].

Outro artigo avalia o consumo energético de uma aplicação de *High Performance Computing* de E/S com processadores ARM e compara o desempenho de servidores de baixa potência. Foi possível reduzir o consumo de energia por 85% em cargas de trabalho de leitura, e em 82% em cargas de trabalho com escrita intensiva [Vinícius Rodrigues Machado 2017].

[Padoin et al. 2017] combina balanceamento de carga dinâmico com voltagem dinâmica e técnicas de escalonamento de frequências para reduzir a frequência de *clock* dos *cores* de computação descarregados. Estas estratégia é aplicada em duas aplicações, Ondes3D e LULESH.

3. LULESH

LULESH é uma miniaplicação implementada em C++ que aproxima equações hidrodinâmicas discretamente, particionando o domínio espacial do problema em uma coleção de elementos volumétricos definidos em um malha. Um nodo da malha é um ponto onde as linhas da malha se interseccionam. LULESH é construído no conceito de uma malha hexágona não estruturada. Ao contrário, são utilizadas matrizes não diretivas que definem os relacionamentos de malha [Karlin et al. 2013]. Apesar de ser uma versão simplificada que resolve um problema de explosão Sedov, os algoritmos numéricos e o fluxo de dados do LULESH são semelhantes ao de aplicações científicas.

4. Resultados Experimentais

4.1. Ambiente

O LULESH possui versões paralelas tanto com OpenMP (API de programação paralela para memória compartilhada), OpenMP+MPI (uso de memória compartilhada e cada nodo, e memória distribuída na comunicação entre os nodos), e CUDA (API de programação paralela usando GPU's da NVIDIA). Os experimentos utilizaram a versão LULESH 2.0 em OpenMP.

A máquina utilizada é a ODROID-XU3 Lite, executando com Linux Ubuntu 16.04 LTS. A ODROID possui a arquitetura big.LITTLE, com um processador Samsung Exynos5422 Cortex™-A15 1.8Ghz *quad core*, e um processador Cortex™-A7 *quad core*. A ODROID-XU3 Lite possui 2GB de memória RAM LPDDR3.

A arquitetura big.LITTLE que a ODROID possui utiliza o modelo *heterogeneous multi-processing* (HMP), que disponibiliza o uso de todos *cores* físicos ao mesmo tempo. Com isso, as *threads* com maior intensidade computacional podem ser alocadas nos *cores big*, enquanto as *threads* menos intensas podem ser alocadas nos *cores LITTLE*, poupando assim o consumo energético[Mittal 2016].

4.2. Metodologia

A miniaplicação LULESH2.0 em OpenMP aplica a paralelização em diferentes laços de repetições do programa. A diretiva *static* é utilizada constantemente ao longo das paralelizações, tornando o escalonamento de *threads* dividido em porções semelhantes de *chunks* (blocos). Outra versão da aplicação foi experimentada, onde a diretiva *dynamic* substituiu o escalonamento estático. Neste escalonamento, é usado uma fila interna que decide os tamanhos dos blocos nas iterações dos laços para cada *thread*. Ambas diretivas modificam o escalonamento das *threads* nos diversos laços de repetições da miniaplicação, onde a diretiva *static* é recomendada em iterações com processamento semelhante, e a diretiva *dynamic* é aconselhada para desbalanceamento de processo entre as iterações.

A placa ODROID foi avaliada com três diferentes tipos de governadores: *on_demand*, *performance* e *powersafe*. No governo *performance*, o governador de desempenho força a CPU a usar a frequência de relógio mais alta possível, que mantém-se estática e não muda. No governo *powersafe*, ao contrário, a CPU utiliza a menor frequência de relógio possível. No governo *on_demand*, a CPU alcança a frequência máxima de relógio quando a carga do sistema for alta, mas também uma frequência mínima quando o sistema estiver ocioso.

Cada uma das duas versões paralelas (estática e dinâmica) da miniaplicação LULESH foram executadas 5 vezes em cada um dos três governadores apresentados. A partir da média aritmética das execuções, obteve-se os valores de tempo e potência da miniaplicação para cada uma das seis versões avaliadas. O consumo energético foi obtido pela multiplicação do tempo médio de execução multiplicado pela potência média encontrada, em Joules.

4.3. Resultados

Pela Figura 1, nota-se que a diferença de potência média, medida em *watts*, é quase cinco vezes menor na versão *powersafe* comparada com as outras versões. A diferença das implementações paralelas estáticas e dinâmicas do LULESH não foi significativa, variando no máximo em cerca de 10% nas versões com o governo *performance*. A potência média da *on_demand* é ligeiramente inferior a versão rodando com o *performance* na versão estática, mas é praticamente idêntica na versão dinâmica.

Nos seis experimentos realizados, a versão dinâmica manteve-se muito inferior à versão estática no tempo de execução, como pode-se observar na Figura 2. O tempo de execução mais rápido foi nas versões com o governo *performance*, onde o tempo de

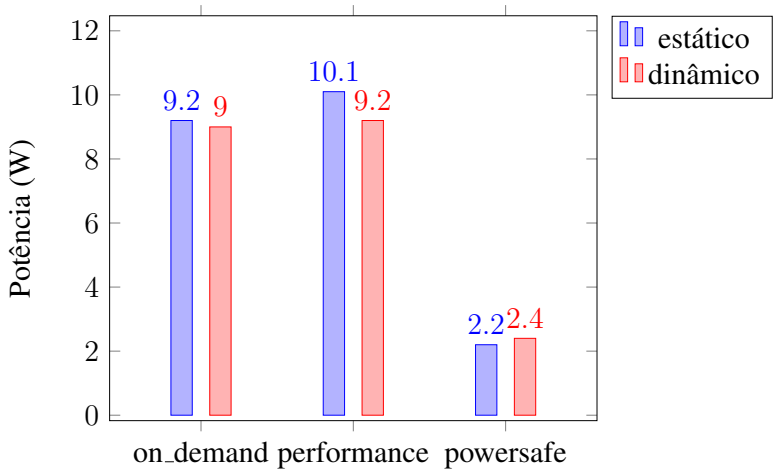


Figura 1. Potência média (em Watts)

execução médio foi de 131,5 segundos na versão *static*, e 596,5 na versão *dynamic*. A versão *on_demand* teve o tempo de execução cerca de 50% maior na versão estática, e 10% maior na versão dinâmica.

Quanto a diferença de performance entre a versão estática e dinâmica, o tempo desperdiçado das *threads* nas barreiras implícitas pode explicar parte da redução de performance da versão dinâmica, pelo maior tempo de espera da sincronização de suas *threads*.

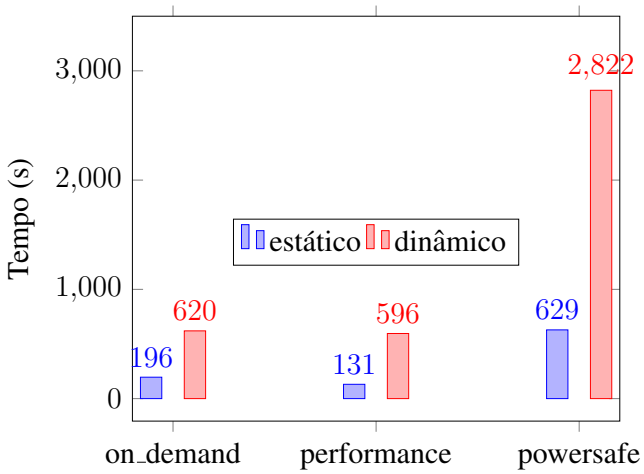


Figura 2. Tempo de execução (em segundos)

Por conta do maior tempo de execução, percebe-se que o consumo energético das versões dinâmicas foi muito superior às versões estáticas, como pode ser observado na Tabela 1. Seguindo a ordem de menor tempo de execução, o menor consumo energético utiliza o governo *performance*, seguido pelo governo *on_demand*, e por último *powersafe*.

Apesar da baixa potência do governo *powersafe*, o *tradeoff* pelo baixo CPU acabou aumentando o consumo energético da execução da miniaplicação LULESH nas duas

Tabela 1. Consumo de Energia (em Joules)

Governo	Versão	Energia (J)
powersafe	estática	1444,2
powersafe	dinâmica	6791
on_demand	estática	1807,8
on_demand	dinâmica	5638,9
performance	estática	1333,4
performance	dinâmica	5494,1

versões, pelo aumento considerável do tempo de execução. Entretanto, percebe-se que na versão estática, a energia total gasta neste governo é inferior ao governo *on_demand*, enquanto na versão dinâmica, o contrário ocorre. A partir desta comparação, pode-se deduzir que o aumento do tempo de execução médio do LULESH influencia negativamente na eficiência energética do governo *powersafe*.

Já no governo *on_demand*, a diferença resultada do modelo *performance* pode ser explicada pelo aumento da frequência de CPU, que começa baixa, por estar em um estado inicial ocioso, e cresce conforme a necessidade de processamento. Como a potência é menor nos segundos iniciais, a média de potência é reduzida e o tempo de execução aumenta. Este *tradeoff* de CPU por potência se mostra menos efetivo que os outros dois governos na versão estática, mas superior ao governo *powersafe* na versão dinâmica.

5. Conclusão

A partir dos experimentos avaliados, foi possível concluir que a utilização do governador *performance* é mais indicado para a execução da miniaplicação LULESH, tanto pelo menor tempo de execução quanto pelo menor consumo energético. O governador *on_demand* teve resultados inferiores, mas por conta de sua flexibilidade de frequência de CPU, pode ser mais adequado para cenários onde ocorram computações extensivas seguidas por períodos de ociosidade. O governo *powersafe* não mostrou-se econômico em aplicações com alta demanda de processamento.

Referências

Arm Limited. big.little. Disponível em `https://developer.arm.com/technologies/big-little`.

Blaise Barney, L. L. N. L. (2015). Introduction to parallel computing. Disponível em: `https://computing.llnl.gov/tutorials/parallel\_comp/`.

Karlin, I., Keasler, J., and Neely, R. (2013). Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973.

Knüpfer, A., Rössel, C., Mey, D. a., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W. E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., and Wolf, F. (2012). *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir*, pages 79–91. Springer Berlin Heidelberg, Berlin, Heidelberg.

Mittal, S. (2016). A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Comput. Surv.*, 48(3):45:1–45:38.

Nikola Rajovic, Alejandro Rico Carro, N. P. A. R. (2014). Tibi-dabo1: Making the case for an arm-based hpc system. *Future Generation Computer Systems*, 36:322–334.

OpenMP Architecture Review Board (1997-2015). Disponível em: `http://www.openmp.org`.

Padoin, E. L., Pilla, L. L., Castro, M., Navaux, P. O. A., and Méhaut, J.-F. (2017). *Exploration of Load Balancing Thresholds to Save Energy on Iterative Applications*, pages 76–88. Springer International Publishing, Cham.

Pedro Langbecker Lima, J. V. F. L. (2016). Rastros de execução paralela da miniaplicação lulesh em openmp. XXXI Jornada Acadêmica Integrada - JAI. Universidade Federal de Santa Maria.

Vinícius Rodrigues Machado, Amanda Binotto Braga, N. G. R. F. Z. B. J. L. B. E. L. P. P. O. N. (2017). Avaliacão do consumo energético de processadores arm em sistemas de arquivos paralelos. *Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul*.

Algoritmo de ajuste de consumo de energia utilizando XenServer

Douglas S. Wang[1], **Vinicius Miana** [1], **Calebe de P. Bianchini**[1]

[1]Faculdade de Computação e Informática – Universidade Presbiteriana Mackenzie
R. da Consolação, 930 – São Paulo, SP – CEP 01.302-907

douglaswang3@gmail.com, vinicius@miana.com.br, calebe.bianchini@mackenzie.br

Abstract. *Reduce the energy consumption represents one great challenge of our century. The data-centers are no exception. In this context, this work presents an algorithm that aims to reduce the energy consumption of data-centers by adjusting the frequency of CPU of its computers. The algorithm uses CPU usage historical data to calculate the optimal frequency. To evaluate the algorithm, a historic series with three virtual machines with different behavior was used. At the end of the experiment, the algorithm showed that savings can reach from 3% to 10% depending on the use case and configuration.*

Resumo. *Reduzir o consumo enérgetico representa um grande desafio do nosso século. Os data-centers não são uma exceção. Dentro deste contexto, este trabalho apresenta um algoritmo que visa a diminuição do consumo de energia através do ajuste da frequência da CPU. O algoritmo usa dados históricos de uso desta CPU para o cálculo da frequência ótima. Para avaliar o algoritmo, foi utilizada uma série histórica composta por três máquinas virtuais com comportamentos distintos ao longo de um período de tempo. Ao final do experimento, o algoritmo mostrou uma redução de 3% a 10% dependo do cenário e das configurações definidas.*

1. Introdução

A computação em nuvem traz novos desafios na constituição de data centers. É estimado que o número de data centers alcance a quantidade de 8.600.000 [Framingham 2014], contra 500.000 em 2011 [Miller 2011]. O rápido aumento no número de data centers dedicados para computação em nuvem acabou por agravar um grande problema da atualidade que é o consumo de energia. Em particular, isso ocorre pela grande quantidade de energia necessária para resfriar esses data centers e mantê-los em funcionamento.

Várias técnicas e tecnologias foram rapidamente desenvolvidas para tentar reduzir o consumo dos data centers. A virtualização é um dos métodos para obter essa melhora [Berl et al. 2010]. Para gerenciar a virtualização é necessário a utilização de sistemas específicos que melhor os recursos físicos. Uma das técnicas utilizadas para fazer essa decisão é o *Dynamic Voltage/Frequency Scaling* (DVFS) [Zaliwski et al. 2016]. Ela tem como fundamento usar o controle de frequência dos processadores para gerir o uso de energia.

Esse trabalho apresenta um algoritmo para o tratamento de dados históricos de uso de uma CPU para o cálculo da frequência a ser ajustada nessa mesma CPU na intenção

de melhorar o consumo de energia. Para avaliar o algoritmo, foi utilizada uma série história composta por três máquinas virtuais com comportamentos distintos ao longo de um período de tempo. Ao final do experimento, o algoritmo mostrou uma redução de 3% a 10% dependo do cenário e das configurações definidas.

Este artigo está organizado em 7 seções. A seção 2 apresenta trabalhos relacionados a *Energy-Efficient*. A seção 3 discute sobre a metodologia utilizada no trabalho. Na seção 4 o algoritmo e os experimentos realizados. A seção 5 discute os resultados. E por fim, na seção 6 são apresentadas as conclusões e os trabalhos futuros.

2. Trabalhos Relacionados

Diversos estudos foram propostos nos últimos anos para tentar resolver o problema de excessivo consumo de energia dos data centers, com as mais diversos focos e soluções. [Berl et al. 2010] faz um breve resumo de várias técnicas utilizadas em sistemas eficientes em energia, sendo que a virtualização é uma tecnologia chave alcançar esse objetivo.

Alguns pesquisadores se focam em criar soluções diretamente no servidor ou na infraestrutura em seu redor como, por exemplo, no tráfego de rede. [Carrega et al. 2012] propôs a agregação do tráfego de rede reduzindo o consumo de energia.

Outros autores se concentram no controle de temperatura dos servidores, pois o seu resfriamento representa uma boa parte do custo de energia. [Rodero et al. 2010] explora as técnicas de monitoramento de máquinas virtuais, atingindo um equilíbrio entre desempenho e temperatura. [Bash and Forman 2007] propôs um modelo que reconfigura automaticamente o sistema de controle térmico, melhorando a eficiência do resfriamento e o consumo de energia.

Há paralelamente esforços para produzir algoritmos de escalonamento, ou novas técnicas, que alinham os interesses de desempenho e gasto de energia. [Tesfatsion et al. 2014] propôs uma solução de escalonamento por meio de diversas técnicas utilizadas no dia a dia para minimizar o custo de energia, preservando o desempenho desejado. [Salehi et al. 2012] realiza um estudo sobre como os data centers que usam os esquemas *advance-reservation* e *best-effort* podem trabalhar sem que o último esquema sofra problemas de falta de recursos. [Song et al. 2007] propõem um esquema adaptativo e dinâmico para gerenciar os recursos de um único servidor de forma eficiente. [von Laszewski et al. 2009] apresenta um algoritmo para alocar de forma eficiente VMs (Virtual Machines, ou, máquinas virtuais) em um cluster DVFS, dinamicamente escalonando sua tensão.

3. Metodologia

Esse trabalho propõe um algoritmo que realiza o tratamento de dados históricos de Uso de CPU para realizar o cálculo da Frequência de CPU. Para que o algoritmo seja validado é necessário definir 3 pontos: (1) a plataforma; (2) os cenários; (3) e a métrica de avaliação.

Para conduzir os experimentos foram utilizados os dados de uma plataforma composta por processadores ”Intel Xeon E5-2670 v2”. Este é um processador de 10-core, frequência máxima de 3.3GHz, com um *Thermal Design Power* (TDP) de 150w, e tensão entre 0.65~1.3v. Por ser uma simulação, foi assumido que a capacidade de escalonamento da frequência do processador permite variações de 100MHz entre 0.0~3.3Ghz.

A intenção da criação de cenários como sendo situações hipotéticas é permitir um melhor controle do ambiente para simular situações especificas. Para execução da análise do algoritmo, foram definidos três cenários diferentes. O primeiro cenário especifica uma VM (máquina virtual) que funciona a 60~90% da sua capacidade de CPU de forma ininterrupta. O segundo cenário especifica uma VM que funciona intensamente nos horários comerciais a 50~100%, e nos demais horários a 0~10%. O último cenário especifica uma VM que possui picos de uso em horários específicos (80~100%), e o resto do tempo ela permanece com um uso de CPU de 40~60%.

A métrica utilizada nesse trabalho, o TDP, foi escolhida pois está diretamente relacionada a dois fatores: energia necessária para resfriamento e o consumo de energia da CPU [Hennessy and Patterson 2011]. De acordo com [Intel®2004], o cálculo do TDP é dado pela Equação 1, sendo que *P* representa a potência (watts); *C*, a capacitância (farads); *V*, a tensão (volts); e *F*, a frequência (mhz).

$$P = CV^2F \tag{1}$$

A partir da métrica coletada da plataforma de gerenciamento, uma medida de qualidade é calculada baseada nos ambientes simulados.

4. Algoritmo Proposto

Esse trabalho é parte de um conjunto de módulos que representam uma situação no mundo real. Um destes módulos possui um algoritmo, conforme apresentado no Algoritmo 1, que é a principal contribuição apresentada neste artigo.

4.1. Experimento

Um dos pontos chave do experimento é a massa de dados coletadas de servidores. Em relação a isto, existem três considerações a serem feitas.

A primeira consideração é que não é possível armazenar a massa de dados com o tamanho desejado (e.g um ano), por diversas situações como (1) falta de espaço físico em disco; (2) tempo de coleta; (3) contrato de confidencialidades; dentre outras situações que inviabilizam esse armazenamento. A segunda, a própria xAPI do XenServer não fornece a granularidade e período de dados desejado, por limitações da própria xAPI. A terceira, o processador escolhido possui tecnologia de ajuste automático de frequência, o que causaria distorções não previstas nos dados coletados.

Para contornar a primeira e a segunda situações, decidiu-se coletar os dados em um período de uma semana com intervalos de um minuto, o que geraria uma quantidade de dados suficiente para uma análise do comportamento do algoritmo. A terceira situação foi resolvida desativando a tecnologia de ajuste de frequência automática da CPU.

Para melhor avaliar os resultados obtidos pelo algoritmo, alguns outros parâmetros foram definidos. A CPU “Intel Xeon Processor E5-2670 v2” possui frequência máxima de 3.3Ghz, utilizando a *boost-technology* e não é possível redefinir frequências que não sejam múltiplas de 100 Mhz [Intel®2004]. Baseado nessas informações, foi decidido utilizar variações de frequência de 100, 300 e 500 Mhz.

Algoritmo 1: Algoritmo de cálculo de nova frequência

Entrada: XML, Data, Hora, Frequência Máxima, Espaço de Tempo, Corte, Variação e Porcentagem
Saída: Valor da Nova Frequência de CPU

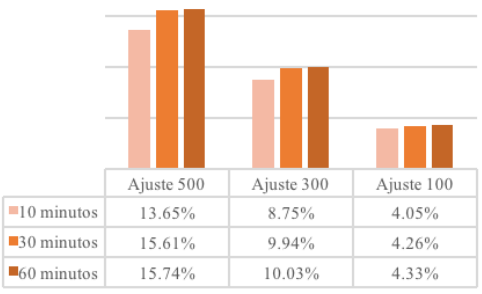
```
1  início
2      Periodo ← (Data + Hora + Espaço de Tempo);
3      para cada item ∈ XML faça
4          se item ⊂ Período então
5              soma ← soma + item;
6              contador++;
7          fim
8      fim
9      se contador > 0 então
10         media ← soma ÷ contador;
11     fim
12     frequência ← CONVERTE(media, Frequência Máxima);
13     se frequência > Frequência Máxima × Corte então
14         se Porcentagem então
15             frequência ← |Variação − (Frequência Máxima × Corte)|;
16         senão
17             frequência ← frequência - Porcentagem;
18         fim
19     senão
20         se Porcentagem então
21             frequência ← Variacao + (Frequência Máxima × Corte);
22         senão
23             frequência ← frequência + Porcentagem;
24         fim
25     fim
26     frequência ← SENSIBILIDADE(frequência);
27     ATUALIZAR(XML, frequência);
28 fim
29 retorna frequência
```

Foram também definidos intervalos de tempo que seriam utilizados no algoritmo: 10, 30 e 60 minutos foram adotado por serem números utilizados em outras pesquisas (seção 2).

5. Resultados

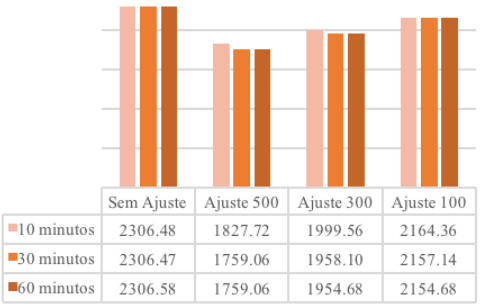
No Cenário 1 (servidor com 60~90% de capacidade de CPU), os ajustes de 100Mhz, 300Mhz e 500Mhz obtiveram uma melhora entre 4~4.3%, 8.7~10% e 13.6~15.7% respectivamente no consumo de energia, conforme mostra a Figura 1. Essa redução no consumo de energia está condizente com a equação de TDP, ou seja, ela é linear e proporcional ao ajuste realizado. Ao mesmo tempo, as janelas de tempo de 10, 30 e 60 minutos mostraram um comportamento não linear, conforme mostra a Figura 2. A comparação

entre as janelas de 10 minutos e a janela de 30 minutos permitiu detectar uma melhora de 5~14.3%. Entretanto, entre a janela de 30 minutos e a janela de 60 minutos ocorreu uma melhora menor, de 0.8~1.65%.



	Ajuste 500	Ajuste 300	Ajuste 100
10 minutos	13.65%	8.75%	4.05%
30 minutos	15.61%	9.94%	4.26%
60 minutos	15.74%	10.03%	4.33%

Figura 1. Ganho porcentual por configuração no Cenário 1



	Sem Ajuste	Ajuste 500	Ajuste 300	Ajuste 100
10 minutos	2306.48	1827.72	1999.56	2164.36
30 minutos	2306.47	1759.06	1958.10	2157.14
60 minutos	2306.58	1759.06	1954.68	2154.68

Figura 2. Frequência média por configuração no Cenário 1

Os demais Cenários (2 e 3) tiveram um efeito oposto. Grande parte das configurações definidas para estes cenários obtiveram resultados negativos. Isso ocorreu devido a frequência média dos cenários possuir valores baixos ao longo do período medido. Ao final, isso resultou em um aumento desnecessário na frequência da CPU, causando um maior gasto de energia em relação ao que seria consumido caso nenhuma política fosse utilizada como pode ser visto nas Tabelas 1 e 2.

Tabela 1. Resultados do cenário 2

Cenário 2	
Configuração	*Variação Consumo*
Ajuste de 100	-1.2∼0.7%
Ajuste de 300	3.7∼7.2%
Ajuste de 500	8.7∼9.3%

Tabela 2. Resultados do cenário 3

Cenário 3	
Configuração	*Variação Consumo*
Ajuste de 100	0.6∼1.4%
Ajuste de 300	9.2∼9.4%
Ajuste de 500	8.7∼14%

6. Conclusão e Trabalhos Futuros

Esse trabalho apresentou um algoritmo que permite analisar dados históricos coletados de um controlador de VMs (máquinas virtuais) e descobrir uma frequência ideal de uso de CPU. Com esse valor gerado pelo algoritmo, é possível definir uma série de configurações para um data center, ou utilizá-lo para fins de análise de desempenho. Os experimentos demonstraram que a aplicação do algoritmo pode gerar resultados positivos para um data center que possui a "Frequência de CPU" alta. Além disso, os dados gerados pelo algoritmo podem ser utilizados para várias outras aplicações, como por exemplo, estatísticas e previsões de consumo de energia de um servidor. Entretanto, foi detectado que o algoritmo analisa e propõe uma alteração na frequência de forma a deteriorar o consumo de energia em cenários de baixo consumo de CPU.

Uma série de outros modelos e técnicas pode ser utilizada de forma conjunta ou complementar ao algoritmo, como por exemplo o *Dynamic Voltage Scaling* (DVS) apresentado por [Changtian and Jiong 2012].

Referências

Bash, C. and Forman, G. (2007). Cool job allocation: Measuring the power savings of placing jobs at cooling-efficient locations in the data center. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 29:1–29:6, Berkeley, CA, USA. USENIX Association.

Berl, A., Gelenbe, E., Di Girolamo, M., Giuliani, G., De Meer, H., Dang, M. Q., and Pentikousis, K. (2010). Energy-efficient cloud computing. *The Computer Journal*, 53(7):1045.

Carrega, A., Singh, S., Bruschi, R., and Bolla, R. (2012). Traffic merging for energy-efficient datacenter networks. In *2012 International Symposium on Performance Evaluation of Computer Telecommunication Systems (SPECTS)*, pages 1–5.

Changtian, Y. and Jiong, Y. (2012). Energy-aware genetic algorithms for task scheduling in cloud computing. In *2012 Seventh ChinaGrid Annual Conference*, pages 43–48.

Framingham, M. (2014). Idc finds growth, consolidation, and changing ownership patterns in worldwide datacenter forecast. Acessado em 16 Março 2014.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

Intel®(2004). Enhanced intel®speedstep®technology for the intel®pentium®m processor. Acessado em 13 Março 2015.

Miller, R. (2011). How many data centers? emerson says 500,000. Acessado em 16 Março 2014.

Rodero, I., Lee, E. K., Pompili, D., Parashar, M., Gamell, M., and Figueiredo, R. J. (2010). Towards energy-efficient reactive thermal management in instrumented datacenters. In *2010 11th IEEE/ACM International Conference on Grid Computing*, pages 321–328.

Salehi, M. A., Krishna, P. R., Deepak, K. S., and Buyya, R. (2012). Preemption-aware energy management in virtualized data centers. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 844–851.

Song, Y., Sun, Y., Wang, H., and Song, X. (2007). An adaptive resource flowing scheme amongst vms in a vm-based utility computing. In *7th IEEE International Conference on Computer and Information Technology (CIT 2007)*, pages 1053–1058.

Tesfatsion, S., Wadbro, E., and Tordsson, J. (2014). A combined frequency scaling and application elasticity approach for energy-efficient cloud computing. *Sustainable Computing: Informatics and Systems*, 4(4):205 – 214. Special Issue on Energy Aware Resource Management and Scheduling (EARMS).

von Laszewski, G., Wang, L., Younge, A. J., and He, X. (2009). Power-aware scheduling of virtual machines in dvfs-enabled clusters. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10.

Zaliwski, A., Lankes, S., and Sinnen, O. (2016). Evaluating dvfs scheduling algorithms on real hardware. In *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, pages 273–280.

Alocação de máquinas virtuais no CloudSim e OpenStack Symphony

Guilherme B. Schneider[1], **Renata H. S. Reiser**[1], **Mauricio L. Pilla**[1], **Vítor A. Ataides**[1]

[1]Universidade Federal de Pelotas (UFPEL)
Computação - CDTec

gbschneider,reiser,pilla,vataides@inf.ufpel.edu.br

Abstract. *Currently on the market there are several opensource tools to implement a cloud, both for simulation and for physical installation, and expect that both obtained values be the same. In this paper are compared one simulation platform titled CloudSim and a virtual machine allocation framework titled Symphony developed for OpenStack. The objective of the work is to compare the results obtained through two allocation algorithms executed in both environments, reinforcing the authenticity between the results of both the simulated environment and the physical environment.*

Resumo. *Atualmente no mercado estão presentes diversas ferramentas opensource para implementar uma nuvem, tanto para simulação quanto para a instalação física, e espera-se que em ambas os valores obtidos sejam os mesmos. Neste trabalho são comparados uma plataforma de simulação intitulada CloudSim e uma framework de alocação de máquinas virtuais intitulada Symphony desenvolvida para OpenStack. O objetivo do trabalho é comparar os resultados obtidos através de dois algoritmos de alocação executados em ambos ambientes, reforçando a autenticidade entre os resultados tanto do ambiente simulado quanto do ambiente físico.*

1. Introdução

A Computação em Nuvem (CN) possibilita acessar recursos computacionais(por exemplo, servidores, redes, serviços e aplicações) de maneira prática e sob demanda, e que podem ser liberados para o usuário sem qualquer envolvimento gerencial [Mell et al. 2011], e ainda, possibilita que desenvolvedores possam implantar suas aplicações em qualquer lugar do mundo, efetuando cobranças específicas dependendo das necessidades de qualidade de serviço de cada cliente. Essas características são capazes devido à criação de máquinas virtuais de acordo com as necessidades da aplicação e alocando-as em máquinas físicas.

Tanto as máquinas virtuais (VMs) quanto as máquinas físicas são heterogêneas. Um dos principais desafios é a alocação dessas VMs, para isso alocadores são propostos com o intuito de otimizar o uso das máquinas físicas, procurando alocar a maior quantidade de recursos na menor quantidade de máquinas físicas, garantindo desempenho e procurando maior eficiência energética.

A motivação deste trabalho é comparar os valores obtidos na alocação de máquinas virtuais através de um ambiente simulado e um ambiente real, pois tanto empresas quanto pesquisadores esperam que os resultados obtidos sejam semelhantes, porém muitas vezes essa realidade pode não ser satisfeita.

Este trabalho é dividido em 6 Seções. Na Seção 2 são apresentados trabalhos que citam os principais desafios na administração de um ambiente de Computação em Nuvem e algumas heurísticas para a solução desses problemas. Na Seção 3 é apresentado um estudo sobre um simulador de Computação em Nuvem, intitulado CloudSim. Na Seção 4 é apresentado um estudo sobre um ambiente real de Computação em Nuvem, intitulado OpenStack Symphony. Na Seção 5 são apresentados os resultados. E por fim, na Seção 6 são apresentadas as conclusões deste trabalho.

2. Trabalhos Relacionados

Em, [Endo et al. 2011] , o autor fala sobre desafios da Computação em Nuvem onde que diversos requisitos do usuário e a natureza da heterogeneidade, dinâmica e descentralização tornam muito desafiador permitir alocação de recursos otimizada

Em, [Majumdar 2011] , o autor descreve os principais conceitos de incerteza presentes nas administração dos recursos de um ambiente de Computação em Nuvem. Essas incertezas referem-se à parâmetros e a políticas.

Em, [Calheiros et al. 2009] , o autor descreve estratégias para lidar com os desafios relacionados a configuração da nuvem de acordo com o uso do usuário, executando seus testes e comparações com o CloudSim, demonstrando suas funcionalidades e flexibilidade. O artigo também apresenta uma série de experimentos demonstrando a eficiência e o baixo consumo de memória do CloudSim na modelagem de grandes ambientes de Computação em Núvem.

Em, [I. Foster and Kaufmann 1999] , o autor propõe um algoritmo de alocação de máquinas virtuais para minimizar o custo que o consumidor tem que pagar ao provedor de nuvem.

3. CloudSim

CloudSim [Calheiros et al. 2011] é um framework para modelar e simular a infraestrutura e os serviços de um ambiente de Computação em Nuvem. Seu principal objetivo é prover um ambiente completo de simulação que possibilita a modelagem e simulação de grandes ambientes de Computação em Nuvem, permitindo com que provedores testem seus serviços repetidamente sob um ambiente controlado, tudo isso com custo zero.

A Figura 1 mostra o design multi-camadas da estrutura do software CloudSim e seus componentes arquitetônicos. Neste trabalho o enfoque está na camada de código do usuário (topo), sua função é expor as configurações básicas do ambiente, sendo essas: *hosts* (número de máquinas e suas especificações), aplicações (número de tarefas e suas necessidades), VMs, número de usuário e seus tipos de aplicações, e ainda as políticas de balanceamento.

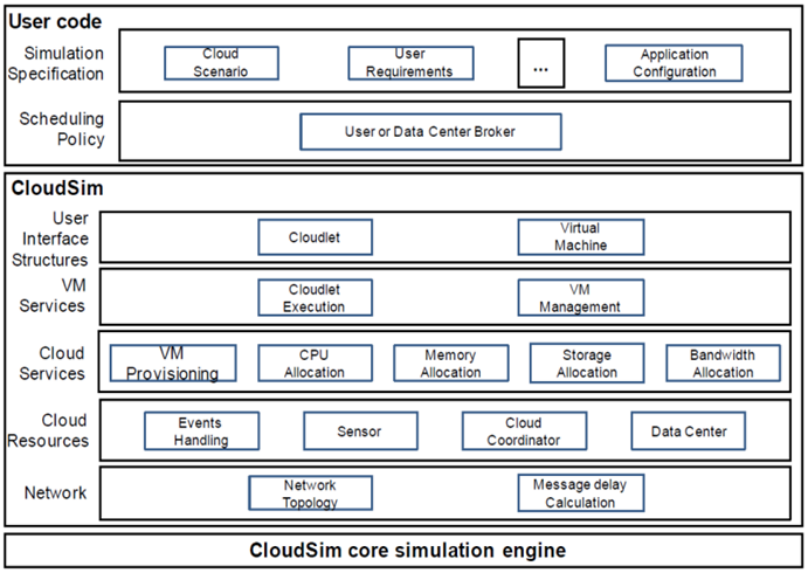


Figura 1. As camadas do CloudSim [Calheiros et al. 2009]

4. OpenStack Orchestra - Symphony

O Symphony é o módulo de escalonamento de VMs do OpenStack Orchestra [Vítor A. Ataides 2017]. Seu objetivo é definir em qual nó de computação uma nova VM será alocada. Para isso o Symphony utiliza uma política de escalonamento de VMs. Como pode ser observado na Figura 2, a arquitetura do Symphony é dividida em 5 partes:

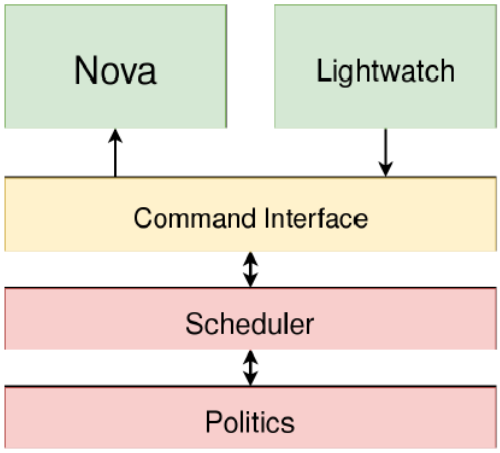


Figura 2. As camadas do Symphony [Vítor A. Ataides 2017]

- **Command Interface** é parte responsável por receber comandos como: requisições de alocações de novas VMs e remoção de VMs.
- **Lightwatch** é a parte que se comunica com o serviço Lightwatch-API para receber *logs* da Nuvem.
- **Nova** é a parte que se comunica com o módulo Nova do OpenStack.
- **Politics** é onde estão implementadas as políticas de escalonamento de VMs.
- **Scheduler** é responsável por aplicar a política selecionada e enviar o resultado ao Nova.

O Symphony foi desenvolvido com o intuito de tornar a adição de novas políticas de alocação de VMs algo simples. Uma política é uma função que tem como parâmetro uma lista de *logs*. Cada *log* contém uma lista de nós de computação e a data do *log*. Cada nó de computação contém uma lista de VMs, seu nome, *IP*, consumo de CPU e consumo de memória. Cada VM contém seu nome, consumo de CPU, consumo de memória. O retorno deve ser o nome do nó de computação escolhido para receber a VM.

5. Resultados

Os testes foram executados em uma nuvem homogênea que contém 9 nós, onde um nó é o *controller* e os outros 8 nós são nós de computação. A configuração é apresentada abaixo:

- **Processador:** Intel Core i5 2310 2.9GHz (80000 MIPS, ou milhões de instruções por segundo)
- **Memória:** 16324MB, DDR3 1333 MHz
- **Disco:** ATA MB1000CBZQE, 1TB, 7200RPM
- **Sistema Operacional:** Ubuntu 14.04.3 LTS
- **Kernel:** Linux 3.19.0-37-generic

As VMs a serem alocadas são de características homogêneas, onde seus requisitos tem como objetivo restringir que uma máquina física contenha no máximo 5 VMs, com configuração apresentada a baixo:

- **Processador:** 15000 MIPS
- **Memória:** 3000MB
- **Disco:** 150GB

Foram realizados testes com duas políticas diferentes, com 50 execuções, que foram divididas em: execuções com 8 VMs, 16 VMs, 24 VMs, 32 VMs e 40 VMs.

Tanto no CloudSim quanto no Symphony os resultados para todas as execuções com as políticas de alocação Round-Robin e Compacta foram os mesmos, nas Figuras 3 e 4 são apresentados os valores obtidos na alocação de 8 VMs, o eixo horizontal representa o tempo necessário para a inserção de uma VM, e o eixo vertical representa cada nó de computação. A mudança de cores representa alteração no número de VMs de um nó de computação. Esses valores mantiveram constância nos demais testes em ambas políticas.

5.1. Round-Robin

Round Robin é uma política que consiste em tentar manter uma distribuição de VMs homogênea.Neste caso como as máquinas iniciavam com nenhuma VM alocada ele primeiramente aloca uma VM em cada nó e continua o processo procurando deixar todos os nós com o mesmo número de VMs.

Na Figura 3 são apresentados os resultados obtidos com esse algoritmo, importante observar após o tempo 1 até o tempo 8, que é o momento que o poder computacional não está mais homogêneo o algoritmo procura retomar essa homogeneidade alocando cada VM subsequente no próximo nó ainda sem nenhuma VM.

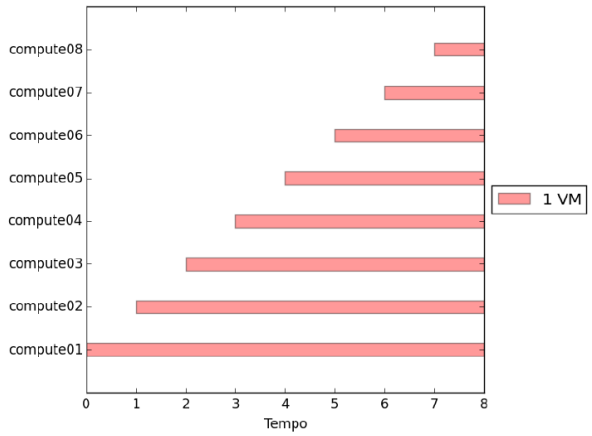


Figura 3. Inserção de 8 VMs com a política RoundRobin

5.2. Compacta

Compacta é uma política que consiste em manter o máximo número de nós de computação desocupados. Para isso ele inicialmente tenta manter o máximo de VMs num mesmo nó de computação, quando este nó não suportar mais VMs ele começa a alocar as novas VMs em outro.

Na Figura 4 são apresentados os resultados obtidos com esse algoritmo, importante observar que até o tempo 5 somente o nó compute01 está com VMs. Já no tempo 6 o algoritmo procura o próximo nó a ser preenchido com VMs devido à limitação proposta nas configurações de VM onde que cada nó suporta no máximo 5 VMs.

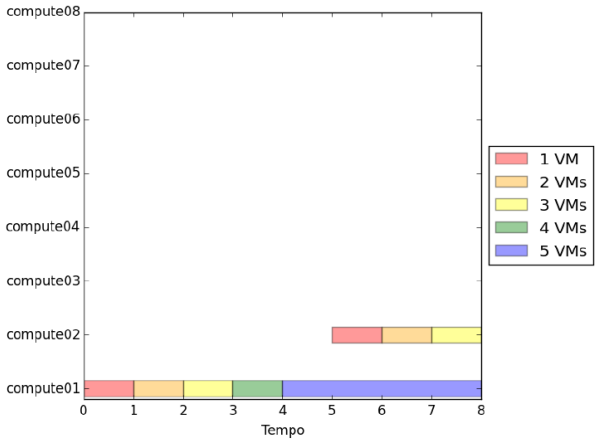


Figura 4. Inserção de 8 VMs com a política Compacta

6. Conclusão

Neste trabalho, validou-se o escalonamento do OpenStack Symphony em relação com uma versão simulada no CloudSim. Os resultados obtidos mostraram-se consistentes e mostram que a modelagem proposta captura as características da versão executada em hardware real, permitindo a exploração futura de outros algoritmos de escalonamento de forma mais dinâmica que com a execução em hardware. Em trabalhos futuros, pretende-se avaliar outros algoritmos de escalonamento para nuvens computacionais, incluindo Lógica Fuzzy.

7. Agradecimentos

Os autores agradecem o financiamento parcial do projeto via editais 448766/2014-0 (MCTI/CNPQ/Universal 14/2014 - B), 310106/2016-8 (CNPq/PQ 12/2016), pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior e pelo projeto Green Cloud FAPERGS/PRONEX.

Referências

Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A., and Buyya, R. (2011). Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50.

Calheiros, R. N., Ranjan, R., De Rose, C. A., and Buyya, R. (2009). Cloudsim: A novel framework for modeling and simulation of cloud computing infrastructures and services. *arXiv preprint arXiv:0903.2525*.

Endo, P. T., de Almeida Palhares, A. V., Pereira, N. N., Goncalves, G. E., Sadok, D., Kelner, J., Melander, B., and Mangs, J.-E. (2011). Resource allocation for distributed cloud: concepts and research challenges. *IEEE network*, 25(4).

I. Foster, C. K. and Kaufmann, M. (1999). The grid: Blueprint for a new computing infrastructure.

Majumdar, S. (2011). Resource management on cloud: handling uncertainties in parameters and policies. *CSI communicatons*, 22:16–19.

Mell, P., Grance, T., et al. (2011). The nist definition of cloud computing. page 7.

Vítor A. Ataides, Laercio L. Pilla, M. L. P. (2017). Openstack orchestra, um escalonador de maquinas virtuais é balanceador de carga para nuvens openstack.

Análise de desempenho de sistemas de gerenciamento de dados em triplas com base no benchmark WatDiv

Felipe Luzzardi da Rosa[1], Roger S. Machado[1], Gerson Geraldo H. Cavalheiro[1], Adenauer Corrêa Yamin[1], Ana Marilza Pernas[1]

[1]Universidade Federal de Pelotas (UFPel), Pelotas – RS – Brasil

{fldrosa,rdsmachado,gerson.cavalheiro,adenauer,marilza}@inf.ufpel.edu.br

Abstract. *Various are the applications, in different areas, that manipulate data in triple format, as is the case of ontologies. One of the reasons for the increasing interest in the use of such structures is the possibility of representation and the inference of knowledge. However, little attention is paid in how to persist and manage such data, with researches being more focused in its modeling. Thus, the present paper has as its motivation the expansion of research related to the persistent of such data, from an analysis of the main existing triplestore systems. For this, a performance evaluation with the intent of comparing the AllegroGraph, GraphDB, MarkLogic, Stardog, and Virtuoso tools was made, utilizing the WatDiv benchmark. In the performed tests Virtuoso showed the best performance in the vast majority of the queries.*

Resumo. *Muitas são as aplicações, em diferentes áreas, que manipulam dados modelados em formato de triplas, como é o caso das ontologias. Um dos motivos para o crescente interesse no uso dessas estruturas é a possibilidade de representação e inferência de conhecimento. Entretanto, observa-se pouca atenção relacionada à forma de se persistir e gerenciar esses dados, sendo as pesquisas mais focadas em sua modelagem. Desta forma, o presente artigo tem como motivação ampliar a pesquisa relacionada a persistência destes dados, a partir de uma análise dos principais sistemas gerenciadores de triplas existentes. Para isso, foram realizados testes de desempenho visando comparar as ferramentas AllegroGraph, GraphDB, MarkLogic, Stardog e Virtuoso utilizando o benchmark WatDiv. Nos testes realizados a ferramenta Virtuoso apresentou o melhor desempenho na grande maioria das consultas.*

1. Introdução

Aplicações cientes de contexto desenvolvidas em áreas como Big Data, Internet das Coisas (IoT) e Web Semântica, utilizam fontes de informação para tomadas de decisões, sendo estas informações disponibilizadas por ontologias. Embora diversas propostas apresentadas neste âmbito façam uso de tais informações, poucas versam sobre a maneira de garantir a persistência das mesmas, ou então, quando fazem, utilizam repositórios relacionais para tal fim [Veiga and Neto 2016, Sena and Bulcão Neto 2016]. Uma forma mais eficiente de manter a persistência de dados ontológicos é utilizando um modelo de armazenamento em triplas RDF (*Resource Description Framework*).

O modelo de armazenamento na forma de triplas pode ser dividido em três grandes categorias com base na arquitetura de sua implementação: em memória; nativo; e externo.

No armazenamento em memória, todo o conjunto de triplas é mantido na memória principal do dispositivo para manipulação, o que o torna ineficiente quando o volume de dados é grande [Maharajan 2012]. O modelo de armazenamento nativo fornece persistência pelo uso de uma base de dados. As ferramentas que implementam este modelo oferecem seus próprios recursos de manipulação da base de dados, utilizando linguagens como a SPARQL para acessá-lo. Na categoria de armazenamento externo, as triplas são persistidas em bancos de dados de terceiros, podendo, neste caso, serem utilizadas bases de dados não apropriadas para triplas, como uma base de dados relacional [Can et al. 2017].

Nota-se que o armazenamento nativo de triplas está ganhando força e popularidade, em grande parte devido ao seu desempenho por contar com uma base de dados apropriada para o modelo de triplas (RDF) [BioOntology 2011]. Apesar disso, poucos trabalhos presentes na literatura tratam efetivamente sobre a persistência e a manipulação dos dados provenientes de ontologias, sendo estas tarefas fundamentais para o provimento da ciência de contexto. A manipulação eficiente desses dados tem impacto nos serviços oferecidos, podendo tornar o processo de tomada de decisão mais eficiente.

Visando contribuir para a pesquisa na área, neste artigo é analisado o desempenho de cinco ferramentas populares para armazenamento nativo de triplas: AllegroGraph, GraphDB, MarkLogic, Stardog e Virtuoso, para identificação daquela mais apta a compor parte da solução proposta para provimento de ciência de contexto. O estudo de caso foi realizado aplicando o *benchmark* WatDiv [Aluç 2014], o qual disponibiliza um número diverso de consultas para teste, possibilitando uma análise mais criteriosa das ferramentas. Os experimentos foram realizados em duas dimensões, na primeira foi analisada a escalabilidade de cada ferramenta, considerando diferentes tamanhos da base de dados. Na segunda os desempenhos fornecidos pelas ferramentas foram comparados entre si. O objetivo do experimento é o de identificar a ferramenta que apresente a melhor relação entre escalabilidade e desempenho.

O restante do artigo está dividido como segue. A Seção 2 apresenta trabalhos relacionados ao armazenamento de dados descritos por ontologias. A Seção 3 versa sobre as ferramentas de gerenciamento de dados nativas em formato de triplas, mostrando um breve resumo sobre as cinco ferramentas analisadas. A Seção 4 apresenta o *benchmark* WatDiv e a Seção 5 a análise de desempenho das ferramentas com este *benchmark*. Por fim, são apresentadas as considerações finais e os trabalhos futuros na Seção 6.

2. Trabalhos relacionados

Nos trabalhos [Faye et al. 2012] e [Modoni and Terkaj 2014] são apresentados *surveys* sobre o armazenamento de dados no formato de triplas. Os trabalhos retratam o conceito do armazenamento em formato RDF, explicando seu funcionamento e principais usos. Com relação as ferramentas para gerenciamento de dados, são apresentadas algumas ferramentas explicando brevemente o funcionamento de cada uma delas e fazendo uma comparação de suas principais características. É importante ressaltar que os trabalhos em questão não fazem qualquer tipo de análise de desempenho, limitando-se a comparar as diferentes ferramentas quanto as suas características técnicas.

Já em [Rajabi et al. 2015] e [Can et al. 2017] o armazenamento de dados em formato de triplas RDF é apontado como mais eficiente do que o armazenamento dos mesmos em um modelo relacional, nos contextos tratados pelos trabalhos. Os artigos, além

de comparar o desempenho de seus estudos de caso com armazenamento no formato RDF e no modelo relacional, discutem diferenças de suas vocações. Como o foco destes trabalhos é a inclusão do modelo de armazenamento em triplas em seus respectivos contextos, não é dada atenção especial às diversas ferramentas disponíveis para realizar tal armazenamento, apresentando análises de desempenho simples entre uma ferramenta que utiliza o modelo relacional e outra que utiliza o modelo de triplas.

Não foram encontrados trabalhos comparando o desempenho de diferentes base de dados nativas ao modelo de triplas. Os trabalhos relacionados identificados limitam-se a comparar o desempenho de uma determinada alternativa a este modelo com opções tradicionais baseadas no modelo relacional. Já trabalhos que tratam especificamente de armazenamento em formato RDF, geralmente não realizam uma análise de desempenho das ferramentas, limitando-se a apresentar características técnicas das mesmas. Também é possível notar que os trabalhos relacionados que tratam efetivamente do desempenho de tais ferramentas encontram-se desatualizados, e tendem a focar mais nas próprias propostas do que nas análises de desempenho em si. Dito isso, este trabalho possui o diferencial de tratar especificamente de uma análise de desempenho baseada no tempo de execução de consultas entre as ferramentas mais populares para o armazenamento de dados no formato de triplas, comparando os resultados com base em um teste estatístico.

3. Ferramentas de gerenciamento de dados nativo em triplas

Os bancos de dados de armazenamento nativos em triplas foram construídos para o armazenamento e recuperação de dados no formato RDF. Para a análise desenvolvida, foram escolhidas cinco ferramentas para armazenamento nativo de triplas, com base principalmente em sua popularidade. Estas cinco ferramentas estão descritas a seguir.

AllegroGraph é uma estrutura de banco de dados e aplicação de alto desempenho para o armazenamento e consulta de dados no formato de triplas. Ela pode ser implantada como um servidor de banco de dados independente e oferece interfaces para acesso remoto, onde a comunicação entre os processos de servidor e cliente é realizada pela Web.

GraphDB (antigamente conhecido como OWLIM) é uma ferramenta para armazenamento de dados no formato de triplas. Possui a possibilidade de realização de inferência semântica, permitindo aos usuários criar novos fatos semânticos de dados existentes. Tanto as consultas como as inferências realizadas sobre os dados armazenados podem ser realizadas em tempo de execução.

MarkLogic é um sistema gerenciador de dados não relacional multi-modelo, capaz de armazenar nativamente documentos JSON e triplas RDF. Além de possuir um modelo de dados flexível, o MarkLogic possui uma arquitetura distribuída que permite trabalhar com uma grande quantidade de dados, mantendo consistência entre as transações.

Stardog é um sistema gerenciador de banco de dados gráfico semântico, implementado em Java. Ele possui suporte para RDF e OWL *Web Ontology Language*, fornecendo capacidades de raciocínio e utilizando SPARQL como linguagem de consulta. Ele possui disponibilidade de acesso aos dados utilizando a web, e *plugins* que possibilitam a utilização de outros *frameworks*.

Virtuoso é um sistema gerenciador de banco de dados que combina a funcionalidade de um banco de dados relacional, XML e RDF, sendo projetado para tirar vantagem

do suporte de segmentação do sistema operacional e múltiplos processadores. O armazenamento de dados é realizado utilizando uma quádrupla onde, além de armazenar a tripla básica (sujeito, predicado e objeto), também é armazenado o grafo relacionado. Com isso o Virtuoso consegue trabalhar com múltiplos grafos de forma simultânea.

4. Benchmark

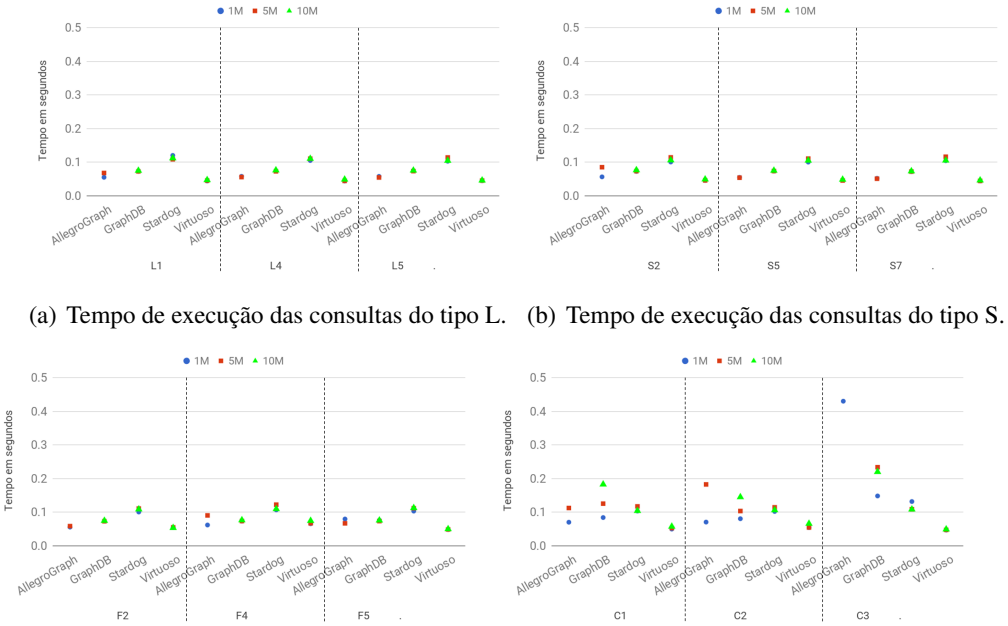
O *benchmark* escolhido para os testes desenvolvidos neste artigo foi o Waterloo SPARQL Diversity Test Suite (WatDiv), desenvolvido pela Universidade de Waterloo [Aluç 2014]. Dentre os principais critérios de escolha deste *benchmark* destacam-se os diferentes tipos de consulta disponibilizados e o fato de ele ser baseado realmente em triplas. Desta forma, permitindo que o *benchmark* identifique problemas de desempenho sobre sistemas gerenciadores de triplas existentes que não são detectados por outros *benchmarks*. O WatDiv fornece um gerador de *datasets* com um fator de escala variável, permitindo assim a geração de *datasets* de diversos tamanhos.

Acompanha o pacote do *benchmark* um gerador de consultas SPARQL. As consultas são geradas em quatros grupos: “Lineares” (L), mais simples e diretas; “Estrela” (S) consultas que referenciam diversos nodos em um formato de estrela; “Floco de Neve”(F), consultas que referenciam vários nodos que por sua vez referenciam outros nodos; e “Complexas” (C), consultas que utilizam uma mistura dos formatos anteriores. É importante observar que variações no tamanho de uma base RDF não devem refletir significativamente no desempenho de seu acesso. O mesmo não é válido para as diferenças de tempo de acesso entre consultas pertencentes aos diferentes grupos.

5. Testes e resultados

Foram realizados testes com 30 execuções, sendo estas focadas no tempo de execução de doze consultas SPARQL, utilizando *datasets* com 1, 5 e 10 milhões de triplas nas cinco ferramentas analisadas. Os testes com 10 milhões de triplas não foram realizados na ferramenta AllegroGraph devido a limitações de sua licença gratuita. Os tempos médios de execução em segundos para cada tipo de consulta, “Lineares” (L), “Estrela” (S) ,“Floco de Neve”(F) e “Complexas” (C) são apresentados na Figura 1. Os desempenhos com a ferramenta MarkLogic não são apresentados pois, além de ter um desempenho muito inferior às demais ferramentas, o desvio padrão se apresentou muito elevado.

Com os resultados apresentados é possível concluir que a ferramenta Virtuoso apresentou os melhores resultados de desempenho na grande maioria das consultas. Também é possível observar que as ferramentas Virtuoso e Stardog apresentaram boa escalabilidade, com baixa variação de tempo de execução entre os diferentes tamanhos de base de dados, enquanto que as ferramentas AllegroGraph e GraphDB apresentaram uma escalabilidade também boa, porém inferior as duas ferramentas anteriormente mencionadas. Esta conclusão foi validada aplicando o teste t de Student com intervalo de confiança de 95% entre as cinco ferramentas, em cada um dos 3 tamanhos e em cada uma das 12 consultas. Como resultados destes testes foram detectados que os resultados da consulta C1 com o tamanho 5M entre Stardog e AllegroGraph e entre Stardog e GraphDB não obtiveram diferenças estatísticas significativas. Ainda, a consulta C3 para o tamanho 1M entre Stardog e GraphDB também não obteve diferenças estatísticas significativas. Os demais resultados apresentaram diferenças estatísticas significativas, assegurando os resultados obtidos e apresentados na Figura 1.



(a) Tempo de execução das consultas do tipo L. (b) Tempo de execução das consultas do tipo S.

(c) Tempo de execução das consultas do tipo F. (d) Tempo de execução das consultas do tipo C.

Figura 1. Tempo de execução, em segundos, nos diferentes tipos de consultas.

Também pode ser notado que o Stardog obteve resultados muito similares em todos os testes realizados, tendo sido a ferramenta com menor alteração de desempenho entre tamanhos diferentes de base de dados. Além dos testes entre as ferramentas, foi analisado se o tamanho da base impactava no desempenho dos sistemas. Para isso, foi realizado o teste t entre as médias de execução de cada ferramenta comparando o tamanho 1M com 5M e 5M com 10M. As consultas que não apresentaram diferenças significativas são apresentadas na Tabela 1. Pode-se observar que a ferramenta AllegroGraph foi a que apresentou menor número de consultas com diferenças não significativas. No entanto, relembra-se que nesta ferramenta não foi possível realizar consultas em uma base de tamanho de 10M. A ferramenta que apresentou maior número de diferenças não significativas foi a MarkLogic, pois foi a ferramenta que apresentou altos valores para os desvios padrões anotados.

Tabela 1. Consultas que não apresentaram diferenças significativas

Tamanho	AllegroGraph	GraphDB	MarkLogic	Stardog	Virtuoso
1M ≠ 5M	S5	L1, L4, L5, F4, S5, S7	C2, C3, L4, L5, F2, S2, S5, S7	C3, L1, F4, S7	C2, L5, F5, S7
5M ≠ 10M		L5	C2, C3, F2, F4, S2, S5, S7	C3, L1, L4, F2, F4, F5, S7	

6. Considerações finais

Este trabalho realizou uma comparação de desempenho relacionada ao tempo de execução de consultas SPARQL entre cinco ferramentas de armazenamento em formato de triplas: AllegroGraph, GraphDB, MarkLogic, Stardog e Virtuoso. Foi apresentada uma descrição de cada ferramenta analisada e também do *benchmark* utilizado para a execução dos testes, o WatDiv. Os testes consideraram o tempo de execução, em segundos, de doze di-

ferentes consultas em três tamanhos de bases de dados. Ao fim dos testes foi possível concluir que a ferramenta Virtuoso foi a que mostrou o melhor desempenho geral, apresentando o tempo de execução mais baixo na grande maioria das consultas, apesar de apresentar uma flutuação no desempenho um pouco maior que Stardog quando varia-se a quantidade de dados tratada.

Estes resultados foram importantes nas decisões de pesquisa a serem tomadas pelo grupo, pois forneceram embasamento para a escolha da ferramenta mais adequada para gerenciar o modelo de triplas implementado no *middleware* para Computação Ubíqua desenvolvido pelo grupo. Esta escolha impacta diretamente nos serviços cientes de contexto oferecidos, pois as aplicações enviam continuamente requisições de consulta aos contextos gerenciados pelo *middleware* para sua tomada de decisão, sendo importante se ter o menor tempo possível de acesso aos dados.

Como trabalhos futuros destaca-se a continuidade da pesquisa ligada a provimento de serviços cientes de contexto, aplicando a ferramenta Virtuoso para acesso aos dados do modelo de triplas.

Agradecimento

O presente trabalho foi realizado com apoio do Programa Nacional de Cooperação Acadêmica da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – PROCAD/CAPES/Brasil.

Referências

Aluç, G. (2014). Diversified Stress Testing of RDF Data Management Systems. Master's thesis, David R. Cheriton School of Computer Science, Waterloo, ON, Canada.

BioOntology (2011). Comparison of Triple Stores. Disponível online em: <https://www.bioontology.org/wiki/images/6/6a/Triple_Stores.pdf>. acesso em novembro 2016.

Can, O., Sezer, E., Bursa, O., and Unalir, M. O. (2017). Comparing relational and ontological triple stores in healthcare domain. *Entropy*, 19(1):30.

Faye, D. C., Curé, O., and Blin, G. (2012). A survey of RDF storage approaches. *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées*, 15:11–35.

Maharajan, S. (2012). Performance of native SPARQL query processors. Master's thesis, Department of Information Technology, Uppsala University.

Modoni, G. E.; Sacco, M. and Terkaj, W. (2014). A survey of RDF store solutions. In *International ICE Conference. IEEE.*

Rajabi, E., Sicilia, M.-A., and Sanchez-Alonso, S. (2015). Interlinking educational resources to web of data through IEEE LOM. *Computer Science and Information Systems*, 12(1):233–255.

Sena, M. V. O. and Bulcão Neto, R. F. (2016). A solution to discard context information using metrics, ontology and fuzzy logic. In *WebMedia*.

Veiga, E. F. and Neto, R. F. B. (2016). Um serviço de representação ontológica de contexto baseada no padrão de projeto estímulo-sensor-observação. In *SBCUP*.

Análise por Kernel do Comportamento de Aplicações de Benchmarks Típicos de GPUs

Pablo Carvalho[1], Cristiana Bentes[2], Esteban Clua[1], Lúcia M. A. Drummond[1]

[1]Instituto de Computação – Universidade Federal Fluminense (UFF)

[2]Dep. Engenharia de Sistemas – Universidade do Estado do Rio de Janeiro (UERJ)

pablocarvalho@id.uff.br, {lucia,esteban}@ic.uff.br, cris@eng.uerj.br

Abstract. *GPUs have been widely used in computer systems and have presented an increasing amount of resources over the years. Benchmark suites were created to evaluate GPU architectures, but this evaluation heavily depends on the diversity of the applications contained in the package. The characterization of a benchmark suite investigates the diversity of its applications. In this work, we present a characterization of Rodinia, Parboil and SHOC benchmark packages, performed per kernel instead of per application. Our aim is to show the applications that have kernels with very distinct characteristics and also the kernels that dominate the execution of each application.*

Resumo. *GPUs têm sido largamente utilizadas em sistemas computacionais e apresentado um aumento crescente da quantidade de recursos disponíveis ao longo dos anos. Diversos pacotes de benchmarks foram criados para avaliar as arquiteturas de GPUs, mas esta avaliação depende fortemente da diversidade das aplicações contidas no pacote. A caracterização de um pacote de benchmarks estuda a diversidade de suas aplicações. Neste trabalho, apresentamos uma caracterização dos pacotes de benchmarks Rodinia, Parboil e SHOC, realizada por kernel ao invés de por aplicação. Nosso objetivo é mostrar as aplicações que possuem kernels com características muito distintas e também quais kernels dominam a execução de cada aplicação.*

1. Introdução

Os avanços nas arquiteturas de alto desempenho têm sido notáveis. Novas arquiteturas surgem rapidamente e, dessa forma, mecanismos para avaliar estas arquiteturas são fundamentais. Tipicamente, arquiteturas são avaliadas através da execução de *benchmarks*, que permitem avaliar a utilização dos recursos, identificar potenciais gargalos de desempenho e suportar decisões de projeto.

Uma série de *benchmarks* foram criados para a avaliação de arquiteturas paralelas ao longo dos anos [Bailey et al. 1991, Woo et al. 1995, Bienia et al. 2008]. Recentemente, entretanto, arquiteturas de alto desempenho têm se orientado para um modelo de computação heterogênea consistindo de CPUs de vários núcleos combinadas com aceleradores - como GPUs ou FPGAs. As GPUs, em particular, oferecem paralelismo em larga escala, com elevada razão custo/desempenho aliada

a um alto poder computacional. GPUs têm apresentado um aumento crescente da quantidade de recursos disponíveis. A cada ano, aproximadamente, uma nova arquitetura de GPU é lançada com diferentes avanços na sua microarquitetura.

Alguns pacotes de *benchmarks* foram criados com intuito de avaliar as arquiteturas de GPUs. São eles: Rodinia [Che et al. 2009], Parboil [Stratton et al. 2012] e SHOC [Danalis et al. 2010]. Pela diversidade de aplicações e aplicabilidade das mesmas, estes *benchmarks* são considerados de alta relevância nas GPUs. A avaliação dos diferentes aspectos da arquitetura de uma GPU depende, porém, da diversidade das aplicações contidas no pacote. O estudo da diversidade das aplicações de um pacote de *benchmarks* é realizado normalmente na caracterização do pacote em termos de uso dos recursos.

Há alguns esforços anteriores na caracterização de pacotes de *benchmarks* para GPUs. Che *et al.* [Che et al. 2010] caracterizam as aplicações de Rodinia em comparação com as do pacote Parsec em termos de instruções por ciclo (IPC), mix de instruções de memória e divergência de *warp*. Kerr *et al.* [Kerr et al. 2009] analisam em um emulador de GPU aplicações do NVIDIA SDK e do pacote Parboil em termos de fluxo de controle, fluxo de dados, paralelismo e comportamento de memória. Goswami *et al.* [Goswami et al. 2010] estudaram os pacotes Rodinia, Parboil e CUDA SDK em termos de caraterísticas dos *kernels*, divergências e coalescência, mas o estudo é baseado em uma simulação de uma GPU genérica, que eles instrumentaram fortemente para extrair as características estudadas. Caracterizações de aplicações irregulares em termos de controle de fluxo e acesso à memória foram estudadas em [Burtscher et al. 2012] e [O'Neil and Burtscher 2014].

Estas caracterizações, entretanto, foram realizadas levando em conta a aplicação completa. As GPUs, porém, possuem particularidades específicas em seu modelo de programação. Uma aplicação inicia sua execução na CPU e despacha *kernels* para serem executados na GPU. Uma aplicação pode ser composta de inúmeros *kernels* que podem ser descarregados para a GPU simultaneamente ou em momentos diferentes. *Kernels* de uma mesma aplicação podem ter características completamente diferentes em termos de uso dos recursos da GPU, o que tem implicação direta na interferência causada na execução concorrente com outros *kernels*. Portanto, a caracterização do uso dos recursos de hardware da GPU pode ficar imprecisa se for feita por aplicação. Este trabalho apresenta uma análise com uma granularidade mais fina, focando apenas nos *kernels*.

Em trabalho anterior [Carvalho 2017], propusemos a caracterização dos pacotes Rodinia, Parboil e SHOC por *kernel*. Esta caracterização, entretanto, mostra apenas os tipos de *kernels* encontrados nos pacotes. Classificamos os *kernels* em relação a: uso restrito de recursos, alto uso de recursos, médio uso de recursos, e baixa ocupância com alta eficiência. Este estudo não mostra as aplicações que possuem *kernels* com características muito diferentes e que, portanto, podem se comportar de maneira muito distinta ao longo de sua execução. É importante destacar quais aplicações possuem este comportamento para que a avaliação da execução da aplicação leve em consideração todos os *kernels* que a compõem.

Neste trabalho, apresentamos um estudo sobre as características dos *ker-*

nels de todas as aplicações dos pacotes de *benchmarks* Rodinia, Parboil e SHOC. Mostramos as aplicações que possuem *kernels* com características muito distintas e também quais tipos de *kernels* dominam a execução de cada aplicação.

2. Caracterização dos Kernels

Em [Carvalho 2017], foi realizada uma caracterização dos *kernels* das aplicações do Rodinia, Parboil e SHOC em termos de número de operações com números inteiros e ponto flutuante, tempo de execução em relação à aplicação, uso da hierarquia de memória, eficiência e ocupação. A caracterização foi realizada em uma arquitetura NVIDIA Maxwell. Foi feita uma análise estatística detalhada e grupos de *kernels* foram destacados nesta análise. Ao todo foram criados quatro grupos diferentes. O primeiro grupo foi caraterizado pelo baixo uso de recursos e baixa porcentagem de tempo de execução em relação ao tempo de execução da aplicação. O segundo grupo foi caracterizado pelo alto uso de recursos apresentando alta eficiência, ocupação média dos *Stream Multiprocessors* em 70%, alto número de operações de memória compartilhada e global e alto número de operações com inteiros e com ponto flutuante de precisão simples. O terceiro grupo foi caracterizado por utilização mais equilibrada dos recursos quando comparado aos outros. Já o quarto grupo é praticamente composto de *kernels* da aplicação s3d do pacote SHOC e sua principal característica consiste na ocupação baixa e eficiência alta.

Os *kernels* do primeiro grupo são pequenos e não são apropriados para avaliar o hardware GPU. Os *kernels* do segundo grupo apresentam alta utilização de recursos, o que indica que eles não são bons candidatos para a execução simultânea com outros *kernels*. Os *kernels* dos terceiros e quartos grupos possuem uma utilização média dos recursos e uma ocupância relativamente baixa. Esses kernels são mais propensos a deixar recursos não utilizados e proporcionar espaço para execução simultânea. Os resultados mostraram também que Rodinia e Parboil apresentaram mais diversidade em seus *kernels*. SHOC, por outro lado, fornece menos diversidade, mas é o único pacote que explora massivamente a concorrência dos *kernels*.

3. Análise por Aplicação

Ao todo foram analisadas 49 aplicações dos três pacotes de *benchmarks*, sendo 11 do pacote Parboil, 24 do pacote Rodinia e 14 do pacote SHOC (não incluímos as aplicações do SHOC nível zero porque contêm somente *microbenchmarks* que testam detalhes de baixo nível do hardware). De todas as aplicações, 13 são compostas de *kernels* que apresentam características bem distintas e, portanto, divergem bastante dos grupos da caracterização proposta. As outras 36 aplicações restantes possuem *kernels* com comportamento semelhante, sendo todos caracterizados no mesmo grupo.

Das 36 aplicações em que todos os *kernels* possuem características semelhantes, a caracterização dos *kernels* é suficiente para a caracterização da aplicação. Os *kernels* das aplicações tpacf (Parboil), myocite e Particlefilter-naive (Rodinia) e BFS (SHOC) pertencem ao grupo 1, o que significa que consomem muito pouco os recursos analisados. Estas aplicações não são apropriadas para avaliação de arquiteturas de GPU. Os *kernels* das aplicações cutcp, lbm, Mri-q, sgemm, spmv e stencil

(Parboil), heartwall, hotspot, huffman, LavaMD, Leukocyte e Srad-V2 (Rodinia) e Stencil2D (SHOC) pertencem ao grupo 2. São as aplicações que mais utilizam recursos da GPU devido à sua alta ocupação, o que significa que os SMs estão perto de sua capacidade máxima em termos do número de *warps* que suportam. Os *kernels* das aplicações bfs e histo (Parboil), b+tree, BFS, hotspot3d, hybridsort, kmeans, mummergpu, nn, nw, pathfinder, sc-gpu, Srad-V1, MD5-Hash (Rodinia) pertencem ao grupo 3, o que significa que eles apresentam utilização de recursos mediana. Os *kernels* das aplicações FFT, GEMM, S3D (SHOC) pertencem ao grupo 4 e apresentam alta eficiência e baixa ocupância. As aplicações cujos *kernels* estão nos grupos 3 e 4 apresentam ocupância relativamente baixa e são mais propensas a deixar recursos não utilizados e fornecer espaço para execução simultânea.

Das 13 aplicações cujos *kernels* possuem características bem distintas, 5 se caracterizam por possuírem *kernels* de um mesmo grupo que dominam o tempo total de execução. São elas, Mri-gridding, sad, Dwt2d, sort e Scan. Em Mri-gridding, sad, Dwt2d, os *kernels* do grupo 2 que consomem muitos recursos de GPU dominam o tempo de execução, consumindo mais de 80% do tempo total. Os outros *kernels* realizam funções auxiliares na execução. Já para sort e Scan, os *kernels* do grupo 3 dominam o tempo de execução. Estas aplicações não realizam muitas operações aritméticas, por este motivo apresentam uma utilização média dos recursos da GPU.

Das 8 aplicações restantes, observamos *kernels* com características distintas com relevância no tempo total de execução. A aplicação Backprop possui 2 *kernels*, sendo um o kernel bpnn_layerforward_CUDA, que consome 40% do tempo de execução e mais recursos, operando com dados de precisão simples e o *kernel* bpnn_adjust_weights_CUDA, que consome 60% do tempo de execução, possui menor eficiência e opera em dados de precisão dupla. A aplicação Gaussian também possui 2 *kernels*: o *kernel* fan1 pertence ao grupo 1, mas possui tempo de execução elevado. Este *kernel* foi classificado no grupo 1 por fazer um baixo número de operações de ponto-flutuante. O *kernel* fan2 realiza mais computações porque é o responsável pela decomposição LU e está classificado no grupo 3. Ambos os *kernels* apresentam menor consumo de recursos por conta das operações de sincronização necessárias. A aplicação Lud possui 3 *kernels*: lud_diagonal, lud_internal e lud_perimeter. O *kernel* lud_internal é o que demanda maior tempo de execução e pertence ao grupo 2. O *kernel* lud_diagonal possui apenas 1 bloco e por isso foi caracterizado no grupo 1. Já o *kernel* lud_perimeter possui uma razão computação/sincronização mais baixa e foi caracterizado no grupo 4. A aplicação Particlefilter-float inicia com a computação com o *kernel* likelihood_kernel que apresenta baixa eficiência e ocupância devido à grande quantidade de sincronização realizada. Em seguida, os *kernels* normalize_weights_kernel e find_index_kernel são executados. Estes *kernels* apresentam menor consumo de recursos da GPU, mas o resultado interessante é que eles dominam o tempo de execução da aplicação. A aplicação MD possui 2 *kernels*, um de *warm-up* que está caracterizado no grupo 4 e um de computação que está caracterizado no grupo 3. A aplicação NeuralNet possui 6 *kernels* no grupo 1 e 3 *kernels* no grupo 3. Os *kernels* do grupo 1 utilizam muito pouco os recursos analisados e são responsáveis por atualizações nos pesos, mas como são muitos *kernels*, eles consomem quase metade do tempo de execução. Os *kernels* do grupo 3 são mais pesados em computação e responsáveis pelas operações de *back propagation*. A aplicação spmv

apresenta comportamento parecido com a aplicação NeuralNet, são 6 *kernels* do grupo 1 que dominam o tempo de execução, enquanto que os 2 *kernels* do grupo 4 realizam mais operações com ponto-flutuante. A aplicação QtClustering possui *kernels* nos grupos 1, 2 e 4. Porém o *kernel* do grupo 4 apresenta porcentagem de tempo insignificante em relação aos outros *kernels*. Nesta aplicação os *kernels* do grupo 1 fazem operações auxiliares como aparar os dados, atualizar máscaras e computar graus. O processamento principal é realizado pelo *kernel* do grupo 2 que consome grande quantidade de recursos da GPU.

Table 1. Contagem de *kernels* de aplicações formadas por vários grupos

Pacote	aplicação	Grupo 1	Grupo 2	Grupo 3	Grupo 4
Parboil	Mri-gridding	0	3	4	0
Parboil	sad	0	2	1	0
Rodinia	Backprop	0	1	1	0
Rodinia	Dwt2d	0	3	0	1
Rodinia	Gaussian	1	0	1	0
Rodinia	Lud	1	1	0	1
Rodinia	Particlefilter-float	2	0	0	1
SHOC	MD	0	0	1	1
SHOC	NeuralNet	6	0	3	0
SHOC	Scan	2	0	4	0
SHOC	Sort	2	0	3	0
SHOC	spmv	6	0	2	0
SHOC	QtClustering	3	1	0	1

4. Conclusões

Este trabalho apresentou um estudo sobre as características dos *kernels* de todas as aplicações de GPU dos pacotes de benchmarks Rodinia, Parboil e SHOC. Nosso estudo focou na caracterização das aplicações que possuem *kernels* com características muito distintas e na caracterização dos *kernels* que dominam a execução das aplicações. Aplicações com *kernels* muito distintos merecem atenção especial principalmente no estudo do consumo de recursos da GPU e na possibilidade de execução concorrente com outras aplicações. Das 49 aplicações estudadas, 36 apresentam todos os *kernels* com a mesma característica de execução. Para estas aplicações, a caracterização dos *kernels* é suficiente para caracterizar a aplicação. Das 13 aplicações restantes, destacamos 8 aplicações que possuem *kernels* com características bem distintas. Os *kernels* dessas aplicações foram estudados em detalhes.

Como trabalhos futuros pretendemos observar como estas características diferentes afetam o comportamento das aplicações em execuções concorrentes. Pretendemos também estudar como esta caraterização é afetada pelo uso de diferentes arquiteturas da GPU.

References

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.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM.

Burtscher, M., Nasre, R., and Pingali, K. (2012). A quantitative study of irregular programs on GPUs. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE.

Carvalho, P. (2017). Caracterização e análise de benchmarks típicos para execução em GPUs. In *Monografia de Trabalho de Conclusão de Curso de Bacharelado em Ciência da Computação, Universidade Federal Fluminense*.

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 Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, page 44:54.

Che, S., Sheaffer, J. W., Boyer, M., Szafaryn, L. G., Wang, L., and Skadron, K. (2010). A characterization of the rodinia benchmark suite with comparison to contemporary CMP workloads. *Proceedings of the IEEE International Symposium on Workload Characterization*.

Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V., and Vetter, J. S. (2010). The scalable heterogeneous computing (SHOC) benchmark suite. *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, page 63:74.

Goswami, N., Shankar, R., Joshi, M., and Li, T. (2010). Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–10. IEEE.

Kerr, A., Diamos, G., and Yalamanchili, S. (2009). A characterization and analysis of ptx kernels. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 3–12. IEEE.

O'Neil, M. A. and Burtscher, M. (2014). Microarchitectural performance characterization of irregular GPU kernels. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 130–139. IEEE.

Stratton, J. A., Rodrigues, C., Sung, I.-J., Obeid, N., Chang, L.-W., Anssari, N., Liu, G. D., and mei W. Hwu, W. (2012). Parboil: A revised benchmark suite for scientific and commercial throughput computing.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. (1995). The splash-2 programs: Characterization and methodological considerations. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 24–36. IEEE.

Avaliação das Interfaces de Ferramentas de Programação Concorrente Com a Suíte Cowichan

Pablo T. Kila, Heitor Augusto G. Almeida
Murilo F. Schmalfuss, Gerson Geraldo H. Cavalheiro

[1]Universidade Federal de Pelotas
Rua Gomes Carneiro, 1 – 96010-610 – Pelotas – RS – Brazil

{ptkila,hagalmeida}@inf.ufpel.edu.br
{mfschmalfuss,gerson.cavalheiro}@inf.ufpel.edu.br

Abstract. *Considering the increasing number of parallel hardware on the market, software solutions must support the new demand and extract the most out of the available resources. This study offers to evaluate concurrency programming tools, specifically C++11, Cilk Plus, OpenMP and TBB, in terms of their interfaces. In this evaluation, versions of applications proposed by the Cowichan benchmark suite are implemented, using the programming resources provided by each tool. The generated source code is then evaluated from the resources used, considering their quantity and complexity of application.*

Resumo. *Considerando o crescente número de opções de hardware paralelo no mercado, é necessário que soluções em software acompanhem a nova demanda, e extraiam o máximo proveito dos recursos disponíveis. Este trabalho se propõe a realizar a avaliação de ferramentas voltadas para a programação concorrente, especificamente C++11, Cilk Plus, OpenMP e TBB, em termos de suas interfaces. Nesta avaliação, são implementadas versões das aplicações propostas na suíte de benchmark Cowichan, utilizando os recursos de programação providos por cada ferramenta. O código gerado é então avaliado a partir dos recursos utilizados, considerando-se sua quantidade e complexidade de aplicação.*

1. Introdução

A utilização de recursos de programação concorrente antes visto como uma característica desejável, tornou-se uma necessidade devido a popularização das arquiteturas paralelas. Mesmo dispositivos cotidianos possuem processadores com mais de um núcleo e a não exploração do paralelismo acarreta em um desperdício de recursos de *hardware*. Diante da necessidade da exploração dos recursos citados, surgiram diversas opções de ferramentas. Estas tem por objetivo facilitar a programação para esse tipo de *hardware*, oferecendo níveis mais elevados de abstração quando comparados aos oferecidos por Pthreads, além de recursos extras. Cada uma dessas novas ferramentas de programação possui suas características específicas ou, ainda, um conjunto de recursos que às tornam mais ou menos adequadas para a representação de determinada aplicação. Mesmo com o alto nível de abstração oferecido pelas ferramentas, determinar suas vocações é uma tarefa complexa, devido principalmente as diversas classes de aplicações existentes. Com o intuito de facilitar essa avaliação, foram desenvolvidas suítes de *benchmarks* voltadas para essa tarefa específica. Essas são compostas por um conjunto de aplicações recorrentes em programação concorrente e por uma metodologia que visa avaliar um conjunto de características, tal como o poder de expressão, quantidade de recursos utilizados, entre outros.

2. Ferramentas Consideradas

A estratégia de execução de uma ferramenta reflete diretamente seu ambiente de execução, desde a definição do número de *threads* até o modo de escalonamento de tarefas. Esse ambiente é definido com base em uma heurística, definida a partir de um conjunto de hipóteses.

A heurística é utilizada para se realizar o mapeamento de recursos de processamento disponíveis. Então, quanto melhor for a heurística de uma ferramenta, melhor será o aproveitamento dos recursos disponíveis, tornando-a mais ou menos adequada para o contexto da aplicação. Como exemplos de ferramentas podem ser citadas: Cilk Plus[INTEL b], TBB[INTEL a] e OpenMP[OpenMP 2015], essas possuem um ambiente de execução definido, sendo possível ainda a partir dos recursos disponíveis realizar uma customização, como por exemplo, alterar o número de *threads*. Existem ainda, a exemplo, dos recursos de manipulação de *threads* presentes em C++ a partir do padrão C++11[Williams 2012], ferramentas onde não há um ambiente de execução.

2.1. C++11

A partir da especificação de 2011 de C++ [Williams 2012] a linguagem sofreu diversas modificações, voltadas predominantemente para melhora de desempenho e conveniência em sua utilização. Para alcançar tais metas foram adicionadas uma gama de novos recursos à sua interface, dentre estes recursos, está um voltado especificamente para a manipulação de *threads*, anteriormente dependente de bibliotecas externas. Devido a linguagem C++ ser orientada a objetos, os recursos de *multhreading* são disponibilizados como classes e objetos e estão no formato de bibliotecas de classes padrão à linguagem, assim ampliando sua capacidade de expressão de forma significativa quando comparado à Pthreads. Dentre os recursos adicionados estão desde mecanismos para o tratamento de exceções, até mecanismos para realização de operações atômicas.

C++11 não possui um ambiente de execução. Sua implementação é realizada normalmente sobre recursos de multiprogramação nativos dos diferentes sistemas operacionais. Basicamente isso significa que sua implementação se dá em torno de Pthreads, ressaltado que C++11 não é apenas um invólucro sobre Pthreads. Essa adição à linguagem possui sua própria gama de recursos, que ou não eram presentes diretamente em Pthreads, ou então sua funcionalidade era mais limitada.

2.2. Cilk Plus

Sua interface básica provê três palavras-chave para definir concorrência na aplicação, que são: `cilk_spawn`, `cilk_sync` e `cilk_for`. A primeira recebe como parâmetro uma continuação que no contexto de Cilk é formado por uma função e seus parâmetros. A segunda palavra-chave serve para realizar a sincronização entre os lançamentos de uma função. Já a terceira palavra-chave permite que um laço que não contenha dependências seja paralelizado. A combinação desses recursos é recomendada na representação de algoritmos paralelos recursivos [INTEL b].

Seu núcleo de escalonamento implementa o modelo $n \times m$, onde a concorrência da aplicação é descrita na forma de n *threads* usuário, sendo mapeada sobre um conjunto de m processadores virtuais, implementados por *threads* sistema, chamados no contexto de Cilk de *workers*. Por padrão o número de *workers* é igual ao de núcleos do processador,

caso seja necessário, é possível alterar-se esse valor. A técnica utilizada em escalonamento das tarefas é a de *work stealing* [Blumofe and Leiserson 1999].

2.3. OpenMP

A especificação de OpenMP [OpenMP 2015] define uma interface de programação concorrente compatível com C e Fortran. Sua interface de programação é formada por três elementos: as diretivas de compilação, a biblioteca de serviços e as variáveis de ambiente. Cada um desses elementos é formado por um conjunto de recursos voltados para partes especificas da execução.

Seu modelo de execução é do tipo $n \times m$, onde n é o número de atividades paralelas, ou tarefas, a serem executadas e m é o número de *threads* em tempo de execução. Por padrão, m é definido como sendo igual ao número de núcleos do processador, sendo possível sua alteração [Chandra et al. 2001]. Ele explora uma estratégia de execução do tipo *fork/join* aninhado. As tarefas são lançadas em tempos iguais, elas compartilham uma barreira localizada ao final do bloco onde foram definidas, que aguarda os seus términos.

2.4. Threading Building Blocks

É uma biblioteca de *templates* desenvolvida na linguagem de programação C++. Seu objetivo é simplificar o desenvolvimento de aplicações que exploram concorrência, por meio de construções abstratas providas pela ferramenta. A biblioteca conta com um grande número de recursos que a tornam basicamente adequada para qualquer formato de aplicação concorrente, não apenas em questão de poder de expressão, como também em opções para obtenção de um melhor desempenho [McCool et al. 2012].

Os recursos disponíveis em TBB estão organizados em várias categorias. Dentre as quais estão: algoritmos, sincronização, escalonamento de tarefas e grupos de tarefas [INTEL a]. Seu modelo de execução é do tipo $n \times m$, assim como 2.2 e 2.3, a técnica utilizada em escalonamento das tarefas é a mesma de 2.2.

3. Avaliação das Ferramentas

Benchmarks são conjuntos de aplicações que focam em analisar alguma característica de um software ou ferramenta de programação. O *benchmark* Cowichan [Wilson and Irvin 1995] propõe um conjunto de 14 problemas simples para comparar a usabilidade de ferramentas para programação paralela. A resolução desses problemas se dá por meio de implementações de laços e operações de redução, assim como no NPB (*NAS Parallel Benchmarks*) [Bailey et al. 1994]. Para este artigo foram escolhidos 6 problemas, *Matrix-Vector Product*, *Game of Life*, *Histogram Thresholding*, *Invasion Percolation*, *Weighted Point Selection* e *Gaussian Elimination*.

Neste trabalho as ferramentas são comparadas levando em conta aspectos qualitativos relacionados as suas interfaces de programação, tendo como ponto de partida a Tabela 1 [Cavalheiro and Du Bois 2014]. Por exemplo, pode ser inferido que o suporte à orientação a objetos provido por TBB e C++11 impacta diretamente no tamanho do executável gerado, ou ainda, que a presença de abstrações de alto nível não necessariamente impactam em uma complexidade de interface menor.

Tabela 1. Comparativo entre as ferramentas [Cavalheiro and Du Bois 2014]

Características	Ferramentas			
	C++11	OpenMP	Cilk Plus	TBB
Complexidade da interface	Média	Baixa	Baixa	Alta
Aderência a linguagem base	Alta	Alta	Média	Alta
Suporte à orientação a objetos	Sim	Não	Não	Sim
Abstrações de alto nível	Não	Algum	Algum	Sim
Descrição da concorrência	Explícita	Explícita	Explícita	Explícita
Escalonamento em nível aplicativo	Não	Sim	Sim	Sim
Decomposição do paralelismo	Explícita	Implícita	Implícita	Implícita
Tamanho do executável	Grande	Pequeno	Médio	Grande

Nas tabelas 2, 3 e 4 são apresentados respectivamente, o número de chamadas a primitivas próprias a gestão da concorrência nas diferentes ferramentas (recursos), o número total de linhas de cada programa e os tamanhos dos executáveis obtidos.

Verifica-se que Cilk Plus realiza muitas chamadas às primitivas de paralelização. No entanto, isto não se reflete, necessariamente, em um maior número de linhas de código. Provavelmente isto deve-se à quantidade extremamente limitada de recursos oferecidos. Em relação a OpenMP, na maior parte das aplicações, o número de recursos utilizados se manteve na média geral, pois, embora de simples utilização, estão presentes em maior número, possibilitando maiores possibilidades com relação ao formato da aplicação. O uso de um maior número de recursos também não implica, necessariamente, no aumento do número de linhas de código. Já em TBB, tem-se de forma geral, a menor utilização de recursos em todas as aplicações. Isso se deve a sua utilização mais complexa, porém mais abrangente em questão de possibilidades. Contudo a utilização destes recursos, não impactou de forma significativa no número total de linhas. No caso de C++11, foi desenvolvida uma classe, denominada Threadpool, para reproduzir o comportamento do modelo de implementação de *threads* $n \times m$. Com esta estratégia foi reduzido consideravelmente o uso de recursos, o que no contexto da ferramenta impacta diretamente também na redução do número de linhas. Sua utilização, no entanto, teve impacto direto no aumento do tamanho do executável.

Tabela 2. Comparativo em relação ao número de recursos utilizados.

Ferramentas	Aplicações					
	Product	Life	Thresh	Invperc	Winnow	Gauss
Cilk Plus	9	4	10	8	12	9
OpenMP	3	5	7	5	11	3
TBB	2	3	4	2	5	2
C++11	2	5	6	2	7	2

4. Conclusão

Devido a modelagem aplicada a versão sequencial já visar a aplicação de recursos de concorrência, a utilização dos mesmos foi bastante simplificada, principalmente por pos-

Tabela 3. Comparativo em relação ao número de linhas.

Ferramentas	Aplicações					
	Product	Life	Thresh	Invperc	Winnow	Gauss
Cilk Plus	88	130	129	146	189	129
OpenMP	87	135	135	160	176	114
TBB	81	126	126	139	190	108
C++11	74	122	110	161	178	99

Tabela 4. Comparativo em relação ao tamanho do executável em Kbytes.

Ferramentas	Aplicações					
	Product	Life	Thresh	Invperc	Winnow	Gauss
Cilk Plus	26	30	31	31	35	31
OpenMP	29	87	37	33	39	32
TBB	38	44	46	59	82	61
C++11	106	151	142	125	175	123

sibilitar estruturas semelhantes entre as ferramentas. Contudo, houveram dificuldades que ocasionaram alguma complexidade, principalmente relacionadas a interface e a curva de aprendizagem de cada ferramenta.

Cilk Plus, possui uma interface mais simples, devido a menor quantidade de recursos providos. A aplicação desses recursos nos problemas propostos foi realizada de forma direta, predominantemente com o uso do `cilk_for`, sem a necessidade de se alterar a estrutura da implementação sequencial e nem sua linguagem base. Considerando interface e aplicação simplificadas, esta ferramenta possui uma curva de aprendizagem menor. Para os casos de exceção, uma estrutura recursiva teve de ser desenvolvida, ampliando a complexidade da implementação, e com isso o número de linhas e uso de recursos.

OpenMP, possui uma interface simples, porém com uma maior quantidade de recursos. Para a maior parte dos problemas, a adição direta das diretivas foi suficiente para a aplicação de concorrência. Novamente não houve a necessidade de se alterar a linguagem base, trazendo ainda mais simplicidade. Para os casos de exceção, foram poucas as modificações necessárias, não acarretando em um grande aumento de linhas e tamanho de executável. Porém para esses casos, alguns recursos extras foram utilizados, mas que não ampliaram a complexidade de forma geral. A complexidade real da ferramenta está em se utilizar as cláusulas providas de forma correta, ou seja, a configuração que obtenha a melhor utilização dos recursos.

TBB, possui uma interface mais complexa. Isso advém de seus recursos que embora não ampliem consideravelmente o número de linhas, necessitam de uma adequação completa da estrutura da aplicação, inclusive sendo necessária uma conversão para C++. Essas características acentuam sua curva de aprendizagem, ampliando sua complexidade de utilização. Nos casos de aplicação, mesmo demandando uma estrutura mais complexa, sua utilização foi mais simples se comparada aos formatos empregados nas outras ferramentas. Ainda são oferecidos vários recursos relacionados a definição de características

do ambiente de execução para um melhor aproveitamento dos recursos disponíveis.

C++11 oferece uma gama de recursos, contudo, se comparado as outras ferramentas esses podem ser considerados de baixo nível. O desenvolvimento das aplicações de forma geral foi bastante simplificado pela implementação de uma classe que mimetiza o modelo de execução $n \times m$ e que implementa uma abstração para execução de laços e reduções em paralelo, aproximando-se das demais ferramentas. Essa implementação simplifica bastante o desenvolvimento em termos de confiabilidade, pois, a utilização da classe isenta o programador da implementação de estruturas repetitivas e passíveis de erros, em especial as de laços, que envolvem a alocação de recursos e a execução de tarefas. Isso acabou reduzindo consideravelmente o número de linhas, em contrapartida, gerando em todos os casos um executável com tamanho maior.

Para os trabalhos futuros considera-se realizar mais versões das mesmas implementações, utilizando os diferentes recursos de uma mesma ferramenta. Essa abordagem visa uma comparação interna entre a ferramenta em questão. Na parte de comparação, além dos critérios atuais, serão definidas métricas que visam a mensuração e comparação do desempenho das diferentes ferramentas com os determinados recursos.

Agradecimento

O presente trabalho foi realizado com apoio do Programa Nacional de Cooperação Acadêmica da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – PROCAD/CAPES/Brasil.

Referências

Bailey, D. et al. (1994). The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center.

Blumofe, R. D. and Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748.

Cavalheiro, G. G. H. and Du Bois, A. R. (2014). Ferramentas modernas para programação multithread. In Salgado, A. C. et al., editors, *JAI*, pages 41–83. SBC, Porto Alegre.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R. (2001). *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco.

INTEL. Intel threading building blocks documentation. `https://software.intel.com/en-us/tbb-documentation`. Acessado em: 03/08/2017.

INTEL. Introducing Intel Cilk Plus. `https://www.cilkplus.org/cilk-plus-tutorial`. Acessado em: 03/08/2017.

McCool, M., Reinders, J., and Robison, A. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco.

OpenMP (2015). OpenMP Application Programming Interface. Specification V. 4.5.

Williams, A. (2012). *C++ Concurrency in Action: Practical Multithreading*. Manning Pubs Co Series. Manning.

Wilson, G. V. and Irvin, R. B. (1995). Assessing and comparing the usability of parallel programming systems. Technical report, University of Toronto.

Estudo de Abordagens de Monitoramento de Desempenho e Energia para a Computação Científica

Gabrieli D. Silva, Mariza Ferro, Victor D. Oliveira
Vinicius P. Klôh, André Yokoyama, Bruno Schulze

[1]Laboratório Nacional de Computação Científica – LNCC
Av. Getúlio Vargas, 333 – 25651-075 – Quitandinha, Petrópolis – RJ –Brasil

{gabrieli,mariza,victord,viniciusk,andremy,schulze}@lncc.br

Abstract. *This work presents a study about approaches and monitoring tools for performance and power consumption, pointing out its strengths and limitations. The goal is to find solutions that enable the monitoring of relevant parameters for analysis of scientific applications in HPC environments. In addition, it presents a web environment developed to relate and visualize data from the monitoring of performance and power consumption, with simplicity and agility.*

Resumo. *Neste trabalho é apresentado um estudo sobre as diferentes abordagens e ferramentas de monitoramento de desempenho e consumo de energia abordando suas vantagens e limitações. O objetivo é encontrar soluções que viabilizem a coleta de parâmetros relevantes para análise de aplicações científicas em ambientes de HPC. Além disso, é apresentado um ambiente web desenvolvido para permitir relacionar e visualizar dados obtidos do monitoramento de desempenho e de consumo de energia, com simplicidade e agilidade.*

1. Introdução

O projeto HPC4e[1], no qual este trabalho se insere, visa o desenvolvimento de simulações altamente escaláveis que viabilizem a geração de energia de maneira mais eficiente. Entre seus objetivos técnicos estão a análise de desempenho de aplicações, a identificação de possíveis gargalos na sua execução e a eficiência no consumo de energia. Portanto, monitorar aplicações e ambiente são tarefas indispensáveis. Por meio do monitoramento e da avaliação de desempenho das aplicações científicas, é possível caracterizar, para os diferentes modelos e tamanhos de problema, quais seus principais requisitos (CPU, E/S, Memória) e qual a relação desses requisitos com o consumo de energia [Silva et al. 2016]. O principal objetivo é realizar este tipo de análise e para isso é fundamental a utilização de ferramentas de monitoramento de desempenho. No entanto, as ferramentas existentes possuem diferentes técnicas para realizar o registro de informações comportamentais, as quais implicam em diferentes vantagens e limitações. Além disso, é essencial que os dados coletados sejam analisados, pois a avaliação e a interpretação dos mesmos é o que viabilizará encontrar soluções que poderão contribuir para a melhoria de desempenho e a redução do consumo energético. É raro encontrar ferramentas que relacionem a avaliação de desempenho e energia simultaneamente, permitindo a fácil visualização desses dados. Portanto, o objetivo deste trabalho é realizar um estudo sobre as diferentes ferramentas e abordagens de monitoramento de desempenho e consumo de energia, encontrando

[1] *High Performance Computing for Energy* - https://hpc4e.eu/

soluções que viabilizem a coleta dos parâmetros considerados relevantes. Ainda, apresentar o ambiente que está em desenvolvimento para análise e visualização de dados, com o qual já é possível relacionar e analisar parâmetros de desempenho e energia.

Este trabalho está organizado da seguinte forma: Na Seção 2 são apresentadas as abordagens de monitoramento de desempenho e consumo de energia, além de alguns trabalhos relacionados. Na Seção 3 é apresentado o ambiente de análise e visualização de dados. Finalmente, na Seção 4 são apresentadas as considerações finais.

2. Abordagens de monitoramento e Trabalhos relacionados

Nesta seção são apresentados conceitos sobre as diferentes abordagens de monitoramento de desempenho e energia bem como ferramentas, apontando as vantagens e limitações observadas. Além disso, dada a grande abrangência da área e dos diversos trabalhos que propõem suas próprias ferramentas e abordagens para a análise de desempenho ou para o consumo de energia, os trabalhos relacionados se restringem a essas ferramentas.

Todo processo de análise de desempenho de aplicações passa por duas fases primárias, a coleta de dados e a análise dos mesmos [Schnorr 2014]. Na abordagem *online*, tanto a coleta quanto a análise, são realizadas simultaneamente e ferramentas de monitoramento são executadas em paralelo com a aplicação para observá-la em tempo real. Entre as ferramentas que fazem uso dessa abordagem, avaliando o impacto que a aplicação está causando no ambiente, foi avaliada a ferramenta Nagios [Guthrie 2013]. Porém, o objetivo deste trabalho não é o monitoramento do desempenho do sistema, mas de um processo específico (aplicação). Na abordagem *offline*, ambas as fases são realizadas separadamente, com a análise da aplicação ocorrendo após a execução da mesma. Nessa abordagem, ainda que ferramentas de monitoramento sejam utilizadas, normalmente é necessário inserir diretivas no código da aplicação para coletar seus dados. Essas diretivas são inseridas manualmente ou por alguma biblioteca que integra a própria ferramenta utilizada, como no trabalho de [Shende and Malony 2006].

A técnica mais comum empregada pelas ferramentas de monitoramento é a amostragem [Reed 1994], a qual consiste em coletar, em intervalos fixos, o estado quantitativo do consumo de diferentes recursos computacionais pela aplicação (por exemplo, Nagios e Intel Vtune). No entanto o rastreamento também é utilizado para coletar os parâmetros. Porém, com essa técnica se obtém uma quantidade de dados significativa, pois todos os eventos ocorridos durante a execução da aplicação são registrados. Dentre as ferramentas que fazem uso desta técnica encontram-se Pablo [Reed et al. 1992] e Paraver [Pillet et al. 1995]. Pablo utiliza um padrão, *SDDF*, para representação dos dados, enquanto o Paraver possui maior flexibilidade por representar graficamente arquivos de rastros vindos de diferentes ferramentas, como DIMEMAS e OMPItrace. Em muitos casos a aplicação precisa ser instrumentada para que seja possível coletar todos os rastros sobre sua execução. Porém, a limitação é justamente a instrumentação, pois é necessário a compreensão do código da aplicação e a cada experimento o mesmo processo deve ser realizado, com dificuldades que serão inerentes a complexidade do código analisado.

As abordagens para a medição do consumo de energia, podem ser classificadas como métodos diretos ou indiretos. Nos métodos diretos as medições podem ser feitas por meio de sensores de hardware externos ou internos, os quais coletam amostras da potência consumida durante um intervalo de tempo [Bridges et al. 2016]. Os medido-

res externos são componentes de hardware conectados entre a fonte de alimentação e o componente sob investigação. Esse tipo de medição não permite uma amostragem muito precisa e detalhada, especialmente para ambientes HPC. Já os medidores internos são obtidos diretamente dos sensores internos, permitindo uma amostragem mais precisa.

As medidas indiretas são utilizadas quando não é possível a medição direta e são feitas por técnicas de modelagem e simulação. Com a modelagem estima-se o consumo de energia usando um modelo que correlaciona a potência com os contadores de desempenho de hardware [Ferro et al. 2017]. Um grande número de trabalhos na área de eficiência energética são focados no uso de modelos para estimar o consumo de energia [Burtscher et al. 2014]. Dentre eles o trabalho de [Bourdon et al. 2013], no qual encontra-se a ferramenta PowerAPI, que a partir do acesso aos sensores e do Identificador de Processo (PID) da aplicação, estima seu consumo energético por componente (CPU, rede, memória,etc). No entanto, como o PID é criado somente ao executar a aplicação, não há possibilidade de obter o consumo de energia no instante inicial e alguns instantes antes ou depois da execução da aplicação, o que é importante na análise para estabelecer precisamente qual a energia consumida pelo sistema, pelo monitor e pela aplicação. Esse processo de monitoramento é detalhado em [Ferro et al. 2017].

Também foram avaliadas abordagens de medida direta usando sensores internos, tal como a ferramenta Intel VTune[2] que permite monitorar tanto desempenho como potência. Como monitor de desempenho oferece uma visão detalhada de cada função existente no código da aplicação. Também dispõe de um driver (*powerdk*) para coletar energia mas, devido à necessidade de várias dependências, sua instalação e uso possuem alta complexidade. Além disso, os dados de desempenho possuem definições próprias, dificultando a interpretação dos resultados. A *NVIDIA System Management Interface* (NVSMI) [3] é um programa que por meio de linhas de comando permite consultar os sensores internos das GPUs NVIDIA e monitorar alguns parâmetros de utilização, tais como a utilização das GPUs e memória, temperatura e consumo de potência. Porém, só pode ser usada para monitorar algumas gerações de GPUs NVIDIA.

Com o Nagios é possível coletar energia e outros parâmetros, mas esse processo é custoso, pois há necessidade de desenvolver o script para coletar energia de acordo com o sensor disponível no ambiente em que a aplicação é executada. Além do processo desse script, outros processos referentes ao Nagios são executados para o funcionamento da ferramenta, consequentemente, o *overhead* causado pela coleta dos dados é maior. Além disso, como o Nagios utiliza uma abordagem online, voltado para o monitoramento de ambientes, há uma perda na eficiência quando ele é utilizado para monitorar aplicações, pois os scripts são executados a altas taxas de amostragem.

Embora seja possível coletar o consumo de energia com diferentes ferramentas, geralmente elas não apresentam módulos de visualização para analisar os dados graficamente (exceto Intel VTune). Além disso, não é comum encontrar uma única ferramenta que permita a coleta e análise de todos os aspectos relevantes no projeto simultaneamente. O que se encontra, na maioria das vezes, são ferramentas que obtém somente os parâmetros relacionados ao desempenho ou ao consumo de potência pela aplicação. Outras ferramentas, como o PAPI, TAU, Paraver e Intel VTune auxiliam mais na análise

[2]https://software.intel.com/en-us/intel-vtune-amplifier-xe

[3]https://developer.nvidia.com/nvidia-system-management-interface

dos rastros da aplicação, como as funções estão sendo executadas e paralelizadas, importantes para fase de otimização das aplicações, mas que não atendem aos objetivos deste trabalho. Além disso, a taxa de amostragem para algumas ferramentas, como o NVSMI, é um pouco baixa (1 Hz) o que pode ser insuficiente para perceber mudanças na potência consumida por aplicações com menor tempo de execução.

Com o objetivo de monitorar a execução de uma aplicação, analisando percentuais e tempos de utilização dos recursos computacionais, traçando um perfil da aplicação e 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. Assim, as limitações apresentadas motivaram o desenvolvimento da nossa própria ferramenta, com a qual é possível realizar não somente o monitoramento de desempenho e energia, mas também relacionar graficamente esses parâmetros por meio de um ambiente de visualização e análise. Esse ambiente é um dos principais objetivos deste trabalho, apresentado a seguir.

3. Ambiente de análise e visualização de dados

No desenvolvimento da ferramenta para monitoramento de desempenho e consumo de energia do grupo ComCiDis [4], o monitoramento é feito utilizando uma abordagem *offline* e por amostragem, consideradas mais apropriadas ao nosso tipo de análise. A abordagem para medir a potência consumida é a medição direta usando os sensores internos, tanto para CPUs como GPUs pois, foi considerada a forma mais precisa para HPC, além de não ser necessário a aquisição de hardwares externos. O monitoramento se dá em três fases (Figura 1): configuração dos experimentos, coleta dos dados e análise. Vale ressaltar que neste trabalho o foco está na terceira etapa, destacada na figura. Os detalhes da primeira e segunda etapas, como por exemplo o processo de coleta de dados sobre o consumo de recursos computacionais e potência, são descritos no trabalho de [Ferro et al. 2017].

Antes do desenvolvimento do ambiente de análise e visualização, após a execução das duas primeiras fases, os dados do monitoramento eram coletados e organizados em planilhas e novos gráficos gerados manualmente, a cada novo conjunto de experimentos. Porém, esse processo era altamente custoso por envolver um grande conjunto experimental, com diferentes aplicações e tamanhos de problema. A necessidade de aprimorar e agilizar essa etapa de análise motivou a proposta de desenvolver e integrar à ferramenta de monitoramento o ambiente de análise e visualização (Figura 1 - etapa 3). Esse ambiente (Figura 2) foi desenvolvido como um servidor web, oferecendo aos usuários acesso aos dados experimentais e a geração de gráficos. Por ser um servidor web facilita a portabilidade, podendo ser utilizado em diferentes dispositivos (tablets e smartphones) e sistemas operacionais e pode ser acoplado a um ambiente de nuvem computacional com facilidade. O ambiente é composto por um conjunto de tecnologias para o funcionamento do servidor web (módulos do Node.js), para geração dos templates dos gráficos (Morris.js) e para converter os dados do monitoramento em formato adequado as demais tecnologias (Parse_data). A utilização dos módulos do Node.js permitem agilizar a construção de servidores web, pois há uma estrutura pronta para criação desses servidores.

Para geração dos gráficos, o usuário inicialmente escolhe quais diferentes parâmetros deseja relacionar e o gráfico é gerado automaticamente, o que facilita a análise

[4]Computação Científica Distribuída - http://comcidis.lncc.br/

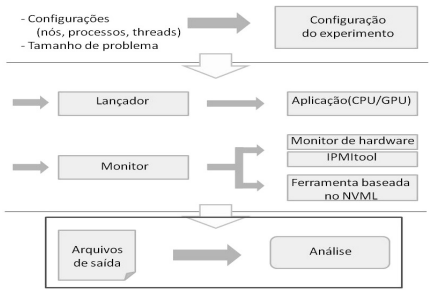


Figura 1. Etapas do monitoramento.

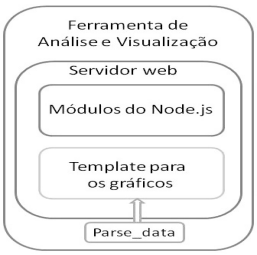


Figura 2. Ambiente de análise e visualização de dados.

dos resultados. Na Figura 3 é apresentado um dos modelos de gráfico gerado, no qual foram selecionados parâmetros referentes ao percentual de utilização dos recursos computacionais (CPU e memória) e ao consumo de potência e temperatura ao longo da execução da aplicação. No eixo esquerdo, apesar dos parâmetros apresentados possuírem diferentes unidades, a escala de valores é construída com base no valor máximo obtido pelos parâmetros. Porém, a medida em que o mouse percorre diferentes áreas do gráfico, aquela área é marcada por pontos, os quais indicam o valor exato de cada parâmetro envolvido. Esses parâmetros também são apresentados ao lado do gráfico com seus valores exatos e suas respectivas unidades. No eixo inferior a escala de valores é construída com o tempo de execução da aplicação. Esses gráficos são gerados utilizando templates em javascript, os quais interpretam dados de entrada no formato *Java Script Object Notation (JSON)*. Também foi desenvolvido um script shell (parse_data), o qual modifica os arquivos de saída gerados pela segunda etapa para o formato JSON.

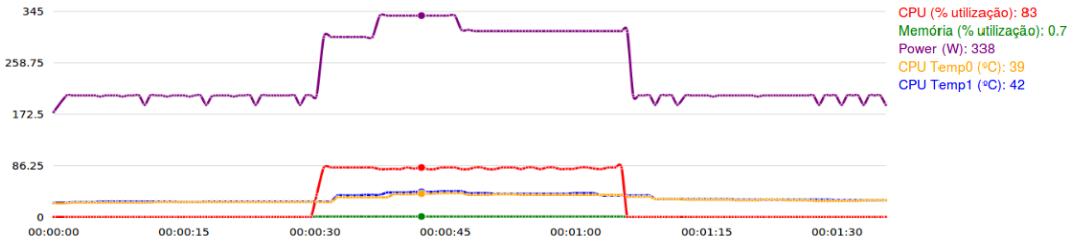


Figura 3. Gráfico gerado pela ferramenta desenvolvida.

Esse tipo de ambiente de visualização foi desenvolvido com foco em se obter boa usabilidade e portabilidade, o que vem agilizando muito a análise e visualização dos diversos conjuntos de dados experimentais gerados pelas pesquisas.

4. Considerações Finais e Trabalhos Futuros

Embora existam diversas ferramentas para análise de desempenho, o conjunto de parâmetros coletados nem sempre atendem as necessidades da pesquisa, pois cada grupo busca atender às suas próprias demandas, como é o caso deste trabalho. Além disso, não é comum encontrar uma única ferramenta que permita coletar e analisar aspectos de desempenho e energia sobre a execução de uma aplicação, e as que coletam informações sobre o consumo de energia, geralmente não apresentam módulos de visualização para análise dos dados. O estudo de diversas ferramentas e das limitações encontradas, motivaram o desenvolvimento de uma ferramenta de monitoramento com ambiente para a

análise e visualização de dados. O ambiente apresentado agiliza o processo de análise de resultados experimentais e a busca de soluções que poderão contribuir para a melhoria de desempenho e a redução do consumo energético das aplicações científicas. Em trabalhos futuros novos modelos de gráficos serão incluídos, além de permitir a combinação de um conjunto maior de parâmetros na geração desses gráficos. Além disso, serão criados templates para visualização do modelo de análise de desempenho Roofline e será feita a integração com o ambiente de nuvem computacional VirtualIS do ComCiDis.

Agradecimentos

Os autores agradecem ao Conselho Nacional de Desenvolvimento Cientifico e Tecnológico (CNPq) e a *European Union's Horizon 2020 Programme* (2014-2020) através do projeto *HPC4e grant agreement nº 689772*.

Referências

Bourdon, A., Noureddine, A., Rouvoy, R., and Seinturier, L. (2013). Powerapi: A software library to monitor the energy consumed at the process-level. *ERCIM News*, (92).

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.

Burtscher, M., Zecena, I., and Zong, Z. (2014). Measuring gpu power with the k20 built-in sensor. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 28:28–28:36, New York, NY, USA. ACM.

Ferro, M., Yokoyama, A., Klôh, V. P., Silva, G. D., Gandra, R., Bragança, R., Bulcão, A., and Schulze, B. (2017). Analysis of gpu power consumption using internal sensors. In *Anais do XVI Workshop em Desempenho de Sistemas Computacionais e de Comunicação*, São Paulo - SP. Sociedade Brasileira de Computação (SBC).

Guthrie, M. (2013). *Instant Nagios Starter*. Packt Publishing.

Pillet, V., Labarta, J., Cortes, T., and Girona, S. (1995). Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: transputer and occam developments*, volume 44, pages 17–31. IOS Press.

Reed, D. A. (1994). Experimental analysis of parallel systems: techniques and open problems. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 25–51. Springer.

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.

Schnorr, L. M. (2014). Análise de desempenho de programas paralelos. In *XIV Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul*, pages 57 – 82, Alegrete, RS, Brazil. Sociedade Brasileira de Computação.

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, 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.

Exemplos e aplicações utilizando a ferramenta ADD

Michael Canesche[1], **Vanessa Vasconcelos**[1], **Fernando Passe**[1], **Jeronimo Penha**[1], **Ricardo Ferreira**[1]

[1]Departamento de Informática – Universidade Federal Viçosa (UFV)
CEP: 36.570-900 – Viçosa – Minas Gerais – Brazil

michael.canesche@gmail.com, vanessa.cr.vasconcelos@gmail.com

fernando.passe@ufv.br, jeronimopenha@gmail.com

ricardo@ufv.br

Abstract. *Accelerators using algorithm modeling throught dataflow graphs on FPGA are nowadays very promising in matter of computational performance and energy consumption efficiency. This paper will present examples of a new tool recently created, which is called Accelerator Design and Deploy(ADD). The ADD offers a library with an amount of synchronous operators ready to use and it also allows the inclusion of new operators in the project.*

Resumo. *Aceleradores utilizando modelagem de algoritmos através de grafos de fluxo de dados com FPGA são atualmente promissores em função de ganhos em desempenho computacional e eficiência energética. Este artigo irá abordar exemplos de uma ferramenta recentemente criada chamada Accelerator Design and Deploy (ADD), a qual possui uma biblioteca de operadores síncronos e, além disso, possibilita a inclusão de novos operadores no projeto.*

1. Introdução

Com a evolução tecnológica, a melhoria de desempenho computacional em conjunto com a eficiência energética se colocaram entre os maiores desafios quando o assunto é computação de alto desempenho. Nesse ínterim, houve a aparição dos aceleradores com FPGA (*Field Programmable Gate Array*) e da GPU (*Graphics Processing Unit*), os quais trouxeram grandes contribuições na área justamente por buscarem o equilíbrio entre gasto energético e desempenho computacional. Todavia, para trabalhar com FPGA é necessário o conhecimento de linguagens de descrição de hardware, como por exemplo, Verilog ou VHDL e, dependendo do problema, existe também a demanda de um árduo e demorado esforço na criação dos projetos.

Este artigo busca apresentar exemplos de utilização da ferramenta *Accelerator Design and Deploy* (ADD) [Penha et al. 2017], com o intuito de demonstrar a praticidade e facilidade que os projetos com FPGA podem oferecer para os meios científico e empresarial. Através da utilização do ADD pode-se realizar simulações com fluxo de dados para validação, bem como para geração automática de códigos Verilog. Desta maneira, economiza-se grande parte do tempo e do esforço na criação de projetos para FPGAs e aumenta-se a praticidade da sua utilização.

2. *Accelerator Design and Deploy*

A ferramenta ADD foi desenvolvida para facilitar a simulação e implementação de algoritmos em hardware. Os softwares disponíveis no mercado atualmente para esse fim, não dispõem de formas visuais e iterativas de construção de algoritmos, ou seja, não permitem ao usuário implementar e testar trechos de código a cada alteração, algo trivial no desenvolvimento tradicional de software e conhecido como técnica TDD (*Test Driven Development*) [Guernsey 2013].

Para o desenvolvimento da ferramenta utilizou-se o simulador HADES [Hendrich 2000] que permite desenvolver trechos de código em Java responsáveis por criar macros (caixinhas), as quais o usuário pode conectar de forma a gerar os algoritmos desejados. Este programa é bastante útil e é utilizado no ensino de disciplinas de hardware [Armenski and Gusev 2014]. Para a construção do ADD foi desenvolvido um conjunto de macros que adicionam funcionalidades ao HADES e permitem a criação e simulação visual de códigos Verilog. Posteriormente este código pode ser executado em um FPGA compatível após sintetização em um software como o Quartus ou o Vivado.

Os operadores que compõem o conjunto desenvolvido são acumuladores, aritméticos, desvios, comparadores, I/O, lógicos, shift e registradores. A escolha de operadores foi feita com base nas instruções disponíveis no processador MIPs [Patterson and Hennessy 2007]. O usuário também pode utilizar as caixinhas previamente disponíveis no HADES, porém estas só funcionarão na simulação já que as mesmas ainda não possuem componentes necessários para geração de código Verilog.

Após a montagem do algoritmo no simulador HADES, caso o usuário queira executá-lo no FPGA, ele deve salvar o arquivo do projeto (.hds) e indicar o seu caminho para o parser, que por sua vez fará a conversão das macros para a linguagem Verilog. O Parser foi feito utilizando a biblioteca Veriloggen [Takamaeda-Yamazaki 2015]. Os arquivos gerados nesse passo devem ser adicionados a um projeto base do Quartus/Vivado para serem sintetizados, e posteriormente o bitstream gerado pode ser carregado no FPGA.

Outro fator importante para a utilização do ADD é o arquivo de configuração que fornece os dados de entrada para o *dataflow* em tempo de execução. Além disso, ele fornece outras informações necessárias, tais como a quantidade de portas da FIFO de entrada e da FIFO de saída e as IDs das macros.

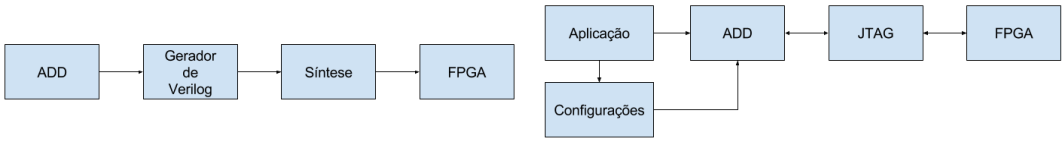


Figura 1. Diagrama de compilação (à esquerda) e criação e execução do projeto (à direita).

A Figura 1 à esquerda mostra as fases do processo de compilação descritas acima. Além disso há a fase de execução ilustrada na Figura 1 à direita. Para a execução ou simulação do algoritmo, é necessário código escrito em Java onde a biblioteca do ADD deve estar inclusa. O código da Figura 2 é um exemplo de como acionar a simulação de um projeto gerado pelo ADD.

```
package add.examples.dataflow_sync;
import add.dataflow.DataflowSyncSimulBase;

public class Teste {
    public static void main(String argv[]) {
        DataflowSyncSimulBase dataflowBase = new DataflowSyncSimulBase
            ();
        dataflowBase.startSimulation("Seu_projeto/Conf.txt", "
            Caminho_HDS/Teste.hds", "\nValor esperado ...\n", 46);
    }
}
```

Figura 2. Exemplo de classe para a simulação de um algoritmo

3. Exemplos e resultados

O formato escolhido para os exemplos deste artigo é o dígrafo. Esta escolha foi feita levando em consideração o didatismo e a economia de espaço. Sendo assim, algumas informações são omitidas, como por exemplo, os fios das ligações do *reset* e do *clock*. Além disso, o grafo está balanceado, por isso a presença de nós registradores e do nó de entrada *IN*. A ferramenta permite até 32 nós de entrada e de saída *OUT*, ou seja, é possível criar projetos que executam até 32 entradas e saídas simultaneamente.

O exemplo da Figura 3 é um algoritmo que possui desvio. É mostrado o código com desvio (à esquerda) e um grafo construído (à direita) que mostra uma das possibilidades ao construir um *dataflow* utilizando a ferramenta ADD. No grafo, nota-se alguns componentes, os quais são: *IN*, *Modi*, *Beqi*, *Addi*, *Reg*, *Merge* e *OUT*, sendo os seus significados, respectivamente, entrada de dados, módulo de um imediato, desvio, adição de um imediato, registrador, merge e saída de dados. Nas arestas do *Beqi* existe a indicação dos possíveis caminhos (if ou else) que o controle pode ativar e, consequentemente, o *Merge* receberá a entrada de acordo com essa resposta.

```
...
for(int i=0; i<x.length; i++){
  if(x[i] % 3 == 0){
    out = x[i] + 3;
  }else if(x[i] % 2 == 0){
    out = x[i] + 2;
  }else {
    out = x[i];
  }
}
...
```

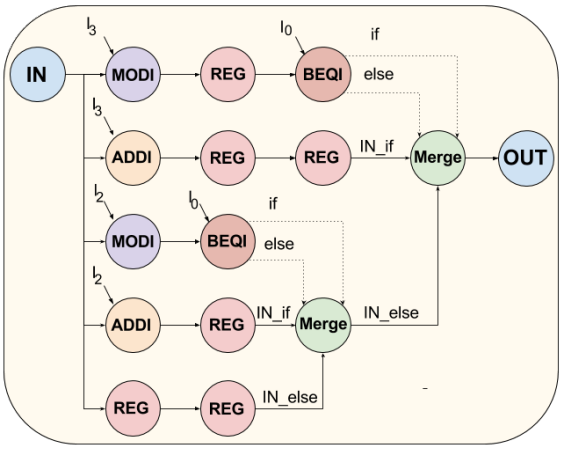


Figura 3. Exemplo de algoritmo com *branch* (à esquerda) e o dataflow com os componentes da biblioteca do ADD (à direita).

O código da Figura 3 possui 3 tratamentos diferentes, um para cada possível condição satisfeita, e há somente uma condição satisfeita por iteração. Cada dado pas-

sado para a configuração atual é derivado da configuração anterior do circuito. A figura 4 mostra a saída do código acima executado em um FPGA Mercurio IV da Altera, em que as entradas vão de 0 a 49. A saída mostra o valor retornado e o esperado para a checagem de erros.

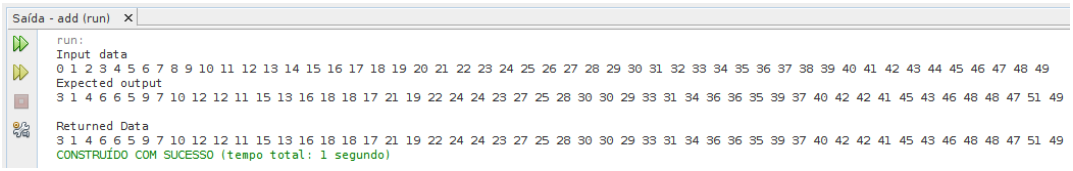


Figura 4. Resultado do exemplo 1 na placa Mercurio IV da Intel/Altera.

O segundo exemplo apresentado é o FIR4, um algoritmo muito conhecido pela comunidade [Parker and Parhi 1997]. Na figura 5 vê-se o dígrafo correspondente. Neste, nota-se algumas novas nomenclaturas além daquelas apresentadas na Figura 3, as quais são: MULTI e ADD, respectivamente, multiplicador com imediato e somador.

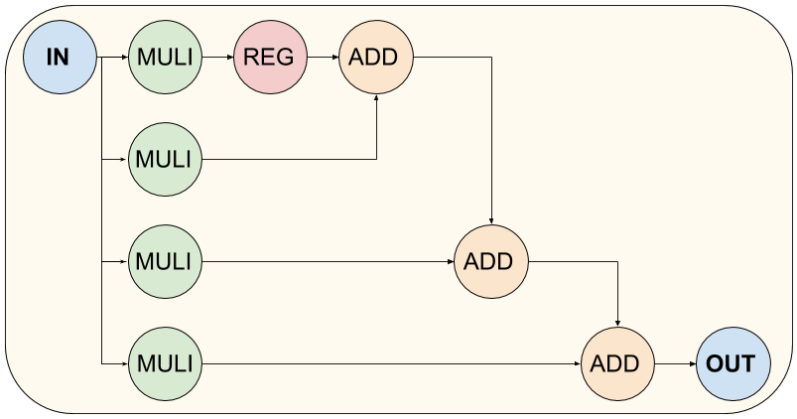


Figura 5. Grafo produzido representando o esquema de um *dataflow* na ferramenta ADD de um algoritmo FIR4.

É possível perceber que a cada *clock* há um deslocamento da informação no circuito, de maneira que de quatro em quatro entradas, cada uma multiplicada por seu respectivo imediato, haja uma saída correspondente a soma destas multiplicações. Na figura 6 está a saída do FIR4 executado no mesmo FPGA utilizado no exemplo da Figura 4. Para este exemplo os dados de entrada variaram entre 0 e 29.

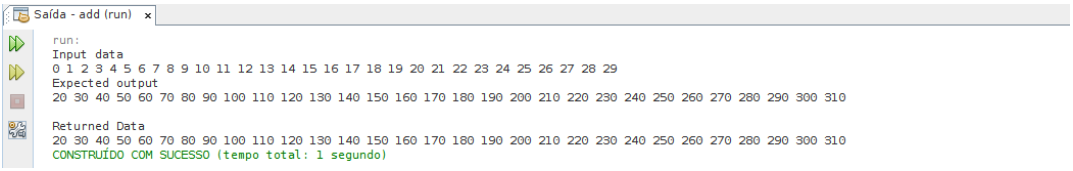


Figura 6. Resultado do exemplo 2 na placa Mercurio IV da Intel/Altera.

O terceiro e último exemplo apresentado utilizando o ADD é um *Map Reduce* de 4:1, isto é, dadas quatro entradas, estas serão reduzidas para uma saída usando um

acumulador. Na figura 7 nota-se o esquema construído em um dígrafo e a inclusão de novos componentes, que são *MAX* e *ACC MAX*, respectivamente, o máximo entre dois valores e o acumulador do máximo.

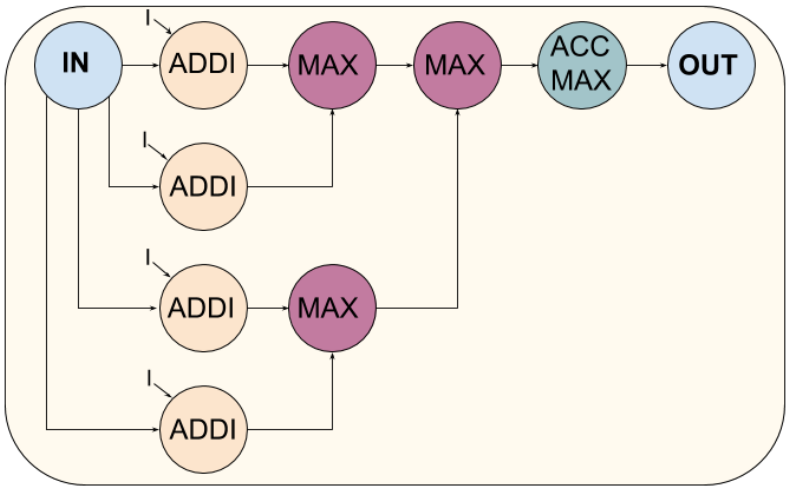


Figura 7. Resultado do exemplo 3 na placa Mercurio IV da Intel/Altera.

A figura 8 apresenta a saída do *Map Reduce*, também executado no Mercurio IV da Altera, com a entrada variando de 0 a 48.

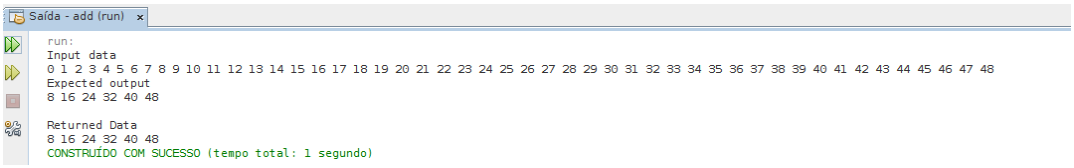


Figura 8. Grafo produzido representando o esquema de um dataflow na ferramenta ADD de um algoritmo MAP REDUCE.

Com os exemplos apresentados é possível perceber que o código gerado a partir do circuito montado no simulador Hades, ao ser executado em um FPGA, retorna os resultados esperados, validando assim a eficácia da ferramenta ADD.

4. Considerações finais

O uso de ferramentas gráficas em projetos de simulação de algoritmos para FPGA é um grande aliado para a reprodução de *dataflows*, principalmente por causa da economia no tempo de criação. Este trabalho mostrou três exemplos relativamente simples de entender (desvio, FIR4 e *Map Reduce*), mas difíceis de criar utilizando Verilog ou VHDL devido a alta quantidade de fios e portas utilizadas. A ferramenta ADD diminui esta dificuldade a ponto de que a criação de código esteja no mesmo nível de desenhar circuitos no simulador Hades, eliminando assim a necessidade de conhecer a fundo uma linguagem descritiva de hardware.

5. Agradecimentos

Os autores agradecem ao CNPq, CAPES e FAPEMIG pelo suporte fornecido no desenvolvimento do trabalho.

Referências

Armenski, G., K. M. R. S. and Gusev, M. (2014). Student satisfaction of e-learning tools for computer architecture and organization course. *Global Engineering Education Conference (EDUCON)*, (1):629–637.

Guernsey, M. (2013). *Test-Driven Database Development: Unlocking Agility*. Addison-Wesley Professional.

Hendrich, N. (2000). A java-based framework for simulation and teaching: Hades—the hamburg design system. In *Microelectronics Education*, pages 285–288. Springer.

Parker, A. D. and Parhi, K. K. (1997). Low-area/power parallel fir digital filter implementations. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 17(1):75–92.

Passe, F., Vasconcelos, V. C., Ferreira, R., Silva, L. B., and Nacif, J. (2016). Virtual reconfigurable functional units on shared-memory processor-fpga systems.

Patterson, D. A. and Hennessy, J. L. (2007). Computer organization and design. *Morgan Kaufmann*.

Penha, J., Bragança, L., Almeida, D., and Nacif, Jose, F. R. (2017). Add: A framework for accerator desing and deploy. `https://github.com/ComputerArchitectureUFV/`. Accessed: 2017-08-10.

Takamaeda-Yamazaki, S. (2015). Veriloggen: A library for constructing a verilog hdl source code in python. `https://github.com/PyHDI/veriloggen`. Accessed: 2017-07-20.

Extensão do Simulador SimuS com uso do Protocolo Firmata

Alonso M. Amparo Neto[1], José Antonio dos S. Borges[2], Gabriel P. Silva[1]

[1]Departamento de Ciência da Computação
[2]Núcleo de Computação Eletrônica
Universidade Federal do Rio de Janeiro (UFRJ) – Cidade Universitária
21941-916 – Rio de Janeiro – RJ – Brasil

alonsoman@ufrj.br, antonio2@nce.ufrj.br, gabriel@dcc.ufrj.br

Abstract. *SimuS is a hypothetical processor simulator intended to be used on computer architecture classes. Its user interface has some virtual I/O devices, but the integration with Arduino allow the users to have access to actual sensors and actuators. The communication between the host and Arduino uses Firmata Protocol. The project is under development and currently SimuS is able to initialize Arduino and control its digital ports, including PWM output. By the end of this project, SimuS will be able to use all the resources of the Firmata protocol, including analog-to-digital conversion. The source code of the Firmata client library written in Object Pascal will be available to download.*

Resumo. *SimuS é um simulador de processador hipotético voltado para o ensino de arquitetura de computadores. Sua interface de usuário possui alguns dispositivos de E/S virtuais, mas a integração com o Arduino permite que os usuários tenham acesso a sensores e atuadores reais. A comunicação entre o host e o Arduino usa o protocolo Firmata. O projeto está em fase de desenvolvimento e atualmente o SimuS é capaz de inicializar o Arduino e controlar suas portas digitais, incluindo a saída PWM. No final deste projeto, o SimuS poderá usar todos os recursos do protocolo Firmata, inclusive conversão analógico-digital. O código-fonte da biblioteca do cliente Firmata, escrito em Object Pascal, estará disponível para download.*

Introdução

O SimuS é um simulador voltado para o ensino de arquitetura de computadores, emulando a arquitetura do processador Sapiens, de relativa simplicidade e cujo funcionamento é de fácil compreensão. A sua simplicidade atende o objetivo final que é possibilitar ao aluno a compreensão do funcionamento da arquitetura do computador. Todavia, apesar de possuir alguns dispositivos virtuais de entrada e saída associados à interface do simulador, boa parte das possibilidades de processamento se resumem ao uso de variáveis com valores previamente carregados na memória do simulador.

Dessa forma, o SimuS ficava bastante limitado aos dados pré-definidos pela entrada do usuário. A integração com o Arduino e plataformas similares permite expandir as possibilidades de programação de baixo nível do SimuS. Por meio de instruções específicas, é possível ao programa executado no simulador controlar LEDs conectados ao Arduino, receber dados de sensores, controlar motores, servos, *displays*, enfim, uma infinidade de dispositivos que podem ser conectados ao Arduino.

A interface entre o computador hospedeiro e o Arduino será realizada por meio do protocolo Firmata para comunicação com microcontroladores. O uso desse protocolo permite simplificar o processo, reduzindo a necessidade de se implementar um novo

protocolo totalmente do zero, já que existe uma implementação disponível do código do servidor para o Arduino.

O objetivo desse projeto é construir uma implementação completa de uma biblioteca cliente para Firmata em Object Pascal, que não está disponível na comunidade, e tornar o SimuS capaz de controlar dispositivos conectados ao Arduino.

Arquitetura do SimuS

A sua arquitetura é uma evolução do processador Neander-X [Borges e Silva 2017], ideal para introduzir o funcionamento de um processador e de sua programação em linguagem de montagem.

O diagrama em blocos do processador Sapiens é mostrado na Figura 1, onde destaca-se as seguintes características:

- Acumulador de 8 bits;
- Apontador de instruções (PC) de 16 bits;
- Apontador de pilha (SP) de 16 bits;
- Códigos de condição (*flag*) N, Z e C;
- Modos direto, imediato e indireto de endereçamento dos operandos em memória;
- Operações de entrada e saída de dados para dispositivos E/S;
- Instrução de TRAP para simular a chamada de instruções no nível de sistema operacional;
- Novas instruções de movimentação de pilha, deslocamento do registrador, soma e subtração com vai-um/vem, entre outras.

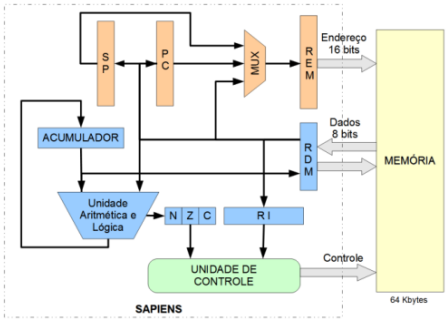


Figura 1. Diagrama em Blocos do Processador Sapiens

Em arquiteturas de processadores reais, a instrução TRAP tem a função de chamar funções do sistema operacional. Como não há um sistema operacional carregado no SimuS, essas operações são desviadas para serem executadas internamente no simulador.

O simulador SimuS interpreta a instrução TRAP como um pedido para que uma determinada operação de E/S seja realizada. O número do *trap,* ou seja, a função que está sendo solicitada é passada no acumulador. A instrução TRAP tem um operando adicional que é o endereço de memória onde os parâmetros adicionais necessários são passados para o módulo do simulador responsável pela realização da operação. Isso ajuda a contornar o problema da carência de registradores do processador Sapiens.

Arduino e Firmata

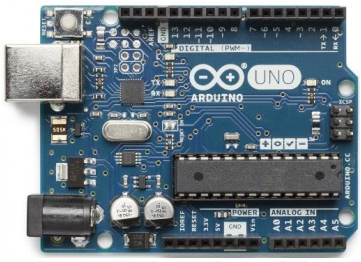


Figura 2. Placa do Arduino

Esse projeto fará uso de um Arduino Uno. Nesse modelo de Arduino há um microcontrolador ATmega328P com 14 portas digitais, numeradas de 0 a 13, sendo que há um LED embutido na porta 13 que pode ser controlado pelo usuário. Além disso, dessas portas digitais, os pinos 3, 5, 6, 9, 10 e 11 são capazes de gerar onda PWM por *hardware*. Adicionalmente, há 6 portas analógicas numeradas de A0 a A5 que podem realizar conversão analógico-digital com 10 *bits* de precisão [SparkFun Eletronics 2012].

O Arduino possui uma IDE própria na qual podem ser escritos programas em linguagem C denominados *sketches,* mas esta IDE não será a ferramenta de programação no escopo deste projeto. O único uso desta IDE será para carregar o *sketch* StandardFirmata.ino no Arduino para realizar o papel de servidor no protocolo. Feito isso, o foco estará presente na interação entre o SimuS e o Firmata.

No Arduino, o funcionamento básico para ter acesso e gerenciar um pino consiste em inicialmente definir um modo de funcionamento para esse pino e então realizar a leitura ou escrita nesse pino, de acordo com o modo de funcionamento definido previamente. O *sketch* apresentado na Figura 3 exemplifica esse funcionamento.

```
int LED_PIN = 13;
void setup () {
    pinMode (LED_PIN, OUTPUT);//habilita o pino 13 para saída digital
}
void loop () {
    digitalWrite (LED_PIN, HIGH); // liga o LED.
    delay (1000);  // espera 1 segundo (1000 milissegundos).
    digitalWrite (LED_PIN, LOW);  // desliga o LED.
    delay (1000);  // espera 1 segundo.
}
```

Figura 3. IDE do Arduino com exemplo que pisca o LED 13

Quando se está programando o Arduino diretamente, essa forma de controlá-lo é suficiente. Entretanto, quando se deseja enviar comandos para o Arduino externamente, por meio de uma aplicação, precisaríamos de uma maneira de enviar comandos ao Arduino para realizar essas tarefas, tais como definir o modo de um pino e o seu nível lógico. É nesse momento que o protocolo Firmata tem sua utilidade.

Esse protocolo funciona com um servidor implementado em uma plataforma com microcontrolador se comunicando através de uma interface serial ou WiFi com um computador hospedeiro, onde um cliente envia os comandos e recebe as respostas do microcontrolador [Hans-Christoph Steiner 2009].

A implementação mais completa que existe para o servidor Firmata é aquela feita para os diversos tipos de Arduino, embora existam implementações disponíveis para outros tipos de plataformas baseadas em microcontroladores.

Em termos de biblioteca Firmata para o cliente, há implementações em diversas linguagens, tais como C, Python, Perl, Javascript, Java, PHP, só para citar algumas. Não existe, contudo, implementações da biblioteca para Object Pascal, a linguagem na qual o simulador SimuS foi programado. [Firmata Protocol Version 2.6 2017]

SimuS e Firmata

No SimuS, como vimos, há uma instrução que tem como função simular as operações de entrada e saída, emulando o funcionamento de um sistema operacional: as instruções de TRAP. Será este o mecanismo que iremos utilizar para a comunicação entre o programa simulado e os pinos do Arduino. Essas ações serão realizadas sob o controle do simulador SimuS, fazendo uso de um cliente Firmata. Essa implementação ao *Framework* Johnny-Five, que é voltado para dispositivos IoT e robótica. [Cvjetkovic e Matijetvic 2016]

Na programação do Arduino, a dinâmica é simples. Deve-se definir um modo de funcionamento para o pino, antes de fazer qualquer atividade de leitura ou escrita referenciando o pino desejado. Para definição do modo de funcionamento de um pino, passaremos o valor 201 no acumulador e, no endereço de memória correspondente ao parâmetro adicional da instrução, serão passados dois parâmetros: um *byte* correspondente ao número do pino e um outro referente ao modo, de acordo com a definição a seguir:

- INPUT (#0): Modo padrão dos pinos. Esse modo necessita de pouca corrente para alternar de estados e é ideal para implementar sensores de toque capacitivos, ler o estado de um LED do tipo fotodiodo.
- OUTPUT (#1): Esse modo de funcionamento permite enviar até 40 mA. Essa corrente é suficiente para acender LED e ativar diversos sensores, mas, é insuficiente para relês, solenóides ou motores.
- PWM (#2): Esse modo de funcionamento tem como função representar as escritas analógicas. E pode representar, de maneira digital, variações de tensão. Com esse recurso, podemos ajustar a intensidade do brilho de um LED, por exemplo. Apesar de o Arduino usar valores entre 0 e 255, estaremos usando valores entre 0 e 1023.
- ANALOG INPUT (#4): Nesse modo de funcionamento, o Arduino irá retornar um valor entre 0 e 1023 para cada leitura realizada no pino. A conversão analógico-digital é realizada pelo próprio Arduino.
- SERVO (#5): Esse modo de funcionamento de pino é uma abstração do Firmata. Quando ativado desse modo, aquele pino fará o trabalho de comunicar com um servo da maneira correta. A ativação desse modo automaticamente realiza um attach_servo() internamente no SimuS.

Os TRAPs necessários para o funcionamento dessa integração do SimuS com o Arduino são:

- DigitalWrite (#202): este TRAP recebe dois parâmetros em memória: o primeiro é o pino a ser escrito e o segundo parâmetro é o valor, HIGH (‘1’) ou LOW (‘0’) que deve ser escrito.
- DigitalRead (#203): este TRAP recebe apenas um parâmetro, que é o pino a ser lido. E retorna no acumulador o valor lido (‘0’ ou ‘1’) no pino definido na entrada.
- AnalogRead (#204): este TRAP recebe um único parâmetro o número do pino a ser lido e retorna na posição seguinte um valor de 16 bits que contém um valor

inteiro de 0 a 1023 representando o valor lido. A tensão de referência assumida é sempre 5 volts.

- PWM (AnalogWrite) (#205): Essa TRAP tem um comportamento semelhante ao DigitalWrite. Ela irá receber dois parâmetros, o primeiro é um byte com o número do pino, o segundo é uma palavra de 16 bits como o "duty cycle" da onda PWM que deverá ser gerada. Uma vez chamada, a porta irá continuamente gerar essa onda repetidamente. Esse segundo parâmetro é um valor entre 0 e 1023, que é mapeado para 0 a 255 no Arduino.
- Servo (#206): Este último TRAP tem por função definir a posição que um servo deve se posicionar. Ela recebe dois bytes como parâmetros: o primeiro define o pino, o segundo define o ângulo da rotação, geralmente um valor entre 0 e 180.

O trabalho consiste em modificar o simulador SimuS e integrá-lo ao protocolo Firmata. Para isso, serão incluídas as novas rotinas de TRAP e criados exemplos de uso para cada um desses novos TRAPs. Ao final de todo o processo, o SimuS deverá ser capaz de fazer uso do máximo de recursos do protocolo.

Atualmente, o SimuS já é capaz de interagir com as portas digitais do Arduino. Uma implementação inicial da biblioteca cliente do Firmata já está em construção e permite iniciar o Arduino que já tenha o protocolo pré-carregado, configurar o modo de funcionamento de um pino e definir um valor lógico *HIGH* ou *LOW* nessa porta. Na Figura 4, é apresentado um exemplo de programa escrito em linguagem de montagem do SimuS que pisca o LED presente no pino 13 do Arduino, equivalente a Figura 3 do Arduino.

```
ORG  0
    LDA #13 ; pino desejado = 13
    STA PIN_TRAP

    LDA #1  ; modo de saída
    STA PIN_TRAP +1
    LDA #201
    TRAP    PIN_TRAP

LOOP:
    LDA #1  ; coloca '1'  no pino 13
    STA PIN_TRAP+1
    LDA #202
    TRAP    PIN_TRAP

    LDA #5  ; espera (500 ms)
    TRAP    T500

    LDA #0  ; coloca '0'  no pino 13
    STA PIN_TRAP +1
    LDA #202
    TRAP    PIN_TRAP

    LDA #5  ; espera (500 ms)
    TRAP    T500

    JMP LOOP

T500:   DW  50
PIN_TRAP:   DS  3
END  0
```

Figura 4. Exemplo em linguagem de montagem do SimuS

O desenvolvimento do cliente Firmata está em andamento. Este apresenta as funcionalidades apresentadas acima, uma vez que está sendo utilizado pelo SimuS em seus *TRAPs*. Como a comunicação com o Arduino é realizada via interface serial, foi incluída no projeto parte da biblioteca Ararat Sinapse em Object Pascal. Essa biblioteca tem seu foco na implementação de redes. Entretanto, ela possui uma adição que é a possibilidade de lidar com portas seriais. Para a implementação do Firmata Pascal, especialmente essa adição que será utilizada como base para o envio e recebimento de dados com o Arduino e similares. [Lukas Gebauer 2012]

Conclusão

Neste artigo, foi apresentada uma proposta de interface do SimuS com o Arduino. Quaisquer dispositivos compatíveis com a plataforma do Arduino poderão ser utilizados, com o uso do protocolo de comunicação Firmata. Toda a implementação em linguagem de montagem deverá ser realizada por meio de TRAPs e elas tem por função ocultar a complexidade das operações de entrada e saída do usuário.

O trabalho a ser realizado consiste no desenvolvimento das rotinas de tratamento dos TRAPs propostos no simulador SimuS, além da implementação de uma biblioteca cliente Firmata em Object Pascal, que pretendemos deixar o código fonte disponível para a comunidade.

Atualmente o projeto ainda está em fase inicial de implementação, sendo assim o SimuS é capaz apenas de controlar as portas digitais do Arduino por meio do protocolo. Os modos de funcionamento e TRAPs propostas são de caráter inicial e estão sujeitas a mudanças de acordo com as necessidades do projeto.

Além disso, outros recursos suportados pelo Firmata serão estudados em uma fase mais avançada do projeto para serem adicionados ao SimuS recursos como motores de passo e os protocolos SPI, I²C e OneWire.

Referências

Borges, J. A. S., Silva, G. P. (2017) “SimuS - Um Simulador Para o Ensino de Arquitetura de Computadores”, WEAC 2017.

SparkFun Eletronics (2012) “Introduction to Arduino”, http://create.coloradovirtuallibrary.org/wp-content/uploads/Curriculum/SparkFun/Beginner/Arduino_final_handout.pdf , Acessado em 24 de Setembro.

Hans-Christoph Steiner (2009) “Firmata: Towards making microcontrollers act like extensions of the computer”, NINE 2009.

Firmata Protocol Version 2.6 (2017) “Firmata Protocol Documentation”, https://github.com/firmata/protocol, Acessado em 20 de Agosto.

Lukas Gebauer (2012) “Ararat Synapse”, https://www.ararat.cz/synapse/doku.php, Acessado em 22 de Setembro.

Vladimir Cvjetkovic, Milan Matijetvic (2016) “Overview of architectures with Arduino boards as building blocks for data acquisition and control systems”, International Journal of Online Engineering . 2016, Vol. 12 Issue 7, p10-17. 8p.

Impacto do uso da Biblioteca ScaLAPACK no Algoritmo de Análise de Componentes Principais (ACP)

Thiago Valença Silva[1], Edward David Moreno[1], Wanderson Roger Azevedo Dias[2]

[1]Departamento de Computação – Universidade Federal de Sergipe (UFS)
São Cristóvão – SE – Brasil

[2]Coordenadoria de Informática – Instituto Federal de Sergipe (IFS)
Laboratório de Arquiteturas Computacionais e Processamento de Alto Desempenho (LACPAD)
Itabaiana – SE – Brasil

{thiagovs23, edwdavid, wradias}@gmail.com

Abstract. *Using parallel features to hone algorithms is a trend in the computing world. This research analyzed the use of parallel libraries for the refinement of the ACP algorithm. From the results obtained from the executions carried out with ScaLAPACK, it was possible to perceive the importance of the correct use of the parallel resources (software and hardware).*

Resumo. *A utilização de recursos paralelos para aprimorar algoritmos é uma tendência no mundo da computação. Esta pesquisa analisou o uso de bibliotecas paralelas para o aperfeiçoamento do algoritmo ACP. A partir dos resultados obtidos das execuções realizadas com o ScaLAPACK, foi possível perceber a importância da utilização correta dos recursos paralelos (software e hardware).*

1. Introdução

Atualmente, aplicações como a rede social Facebook e a máquina de busca Google têm a tarefa de retirar informações, em tempo hábil, de bases de dados cada vez maiores. Sendo assim, necessita-se um poder computacional robusto para a execução desta atividade, mas como fazer isso sem aumentar exponencialmente os custos relacionados a criação deste computador? A resposta pode estar na inserção de uma arquitetura paralela em que são utilizados computadores não tão potentes, mas que possam trabalhar em conjunto na execução de uma tarefa, conhecido como *cluster* heterogêneo.

Vários algoritmos necessitam processar uma quantidade considerável de dados, como é o caso da Análise de Componentes Principais (ACP) (Jolliffe, 2002), necessitando de um tempo maior para apresentar resultados. Um dos pontos que atrasam o processamento em uma arquitetura sequencial é que mesmo que não exista uma dependência entre as tarefas, elas são executadas uma após a outra. Assim, buscou-se reunir dados com o propósito de responder de que forma a aplicação de técnicas de desenvolvimento paralelo pode aumentar o desempenho no processo de obtenção de componentes principais.

Então, para o desenvolvimento deste trabalho foram utilizadas pesquisas bibliográficas e testes de escalabilidade, sendo estes executados utilizando multiplicação de matrizes e o programa que reproduz o algoritmo de Análise de Componentes Principais.

O restante do artigo está organizado da seguinte forma: a Seção 2 apresenta uma breve contextualização sobre a programação paralela, a fim, de ambientar o estudo corrente, nesta seção também explana conceitos sobre *Message Passing Interface* (MPI), ScaLAPACK e Análise de Componentes Principais (ACP); A Seção 3 apresenta as análises de resultados dos testes realizados e a Seção 4 finaliza com as conclusões e idéias para trabalhos futuros.

2. Programação Paralela

Em um cenário onde a indústria não estava conseguindo desenvolver *chips* que acompanhassem a *Lei de Moore*, foi necessário haver uma ruptura com o modelo antigo de processadores mono-core. “A indústria de

computação mudou o curso em 2005 quando a Intel [...] anunciou que dali em diante seus computadores de alta performance iriam conter múltiplos processadores ou núcleos." (Asanović, *et al.*, 2006).

Em consonância com a citação acima, Pacheco (2011, p.1,2) afirma que a maioria dos produtores de *chip* (Intel, AMD, ARM e etc), decidiram que o caminho para o rápido aumento de desempenho é na direção do paralelismo. Além disso, ele afirma que essas mudanças revolucionariam também o modo de programar já que os códigos seriais não iriam se adequar magicamente às novas arquiteturas para aumentar o ritmo de processamento.

Contudo, antes de 2005 a computação paralela não era tão explorada porque até então o processamento serial fazia bem o seu papel, porém isso começou a não acontecer mais e era necessária uma mudança. A programação paralela surge como solução para este problema, porém foi necessário haver mudanças que vão além da arquitetura. Programas deveriam ser reescritos, agora, para realizar instruções paralelas e não só seriais como eram antigamente.

Segundo Rauber & Rünger (2013, p.1) a computação paralela vem sendo bem consolidada ao passar dos anos em simulações de problemas científicos que exigem uma alta performance. Ademais, por causa das mudanças de hardware já citadas, o processamento paralelo tornou-se um campo primordial em técnicas de desenvolvimento de software. Então, subentende-se que cada vez mais a computação paralela deve continuar sendo explorada tanto no meio acadêmico quanto na indústria como resultado da estagnação nos índices de desenvolvimento em processamentos mono-core.

2.1. Message Passing Interface (MPI)

De acordo com Rauber & Rünger (2013, p.228) o modelo de envio de mensagens (*Message Passing Model*) é apropriado para uma arquitetura onde não há memória global, ou seja, a memória está distribuída entre os processadores. Este modelo trabalha enviando mensagens da memória de um processador para a memória de outro. Para isso, instruções de envio e recebimento devem ser realizadas pelos processadores.

É interessante destacar que o modelo MPI foi criado para arquiteturas como *clusteres*, onde a memória está distribuída entre os processadores, além da necessidade de utilização de instruções extras para lidar com a comunicação entre memórias. Mesmo assim espera-se que essas instruções sejam utilizadas de forma a não se sobrepor as comunicações com outras operações em tempo de execução do algoritmo.

Conforme Gropp, *et. al.* (1999), o desafio encontrado para a obtenção de uma interface que pudesse prover paralelismo era a portabilidade, então, a comunidade científica se mobilizou para criar uma biblioteca padrão que pode ser utilizada na maioria dos sistemas paralelos de memória distribuída. Muitas bibliotecas foram criadas, porém a que mais se destacou e a mais utilizada é a Interface de Envio de Mensagens (*Message Passing Interface* ou MPI).

Pode-se dizer que tanto Rauber & Rünger (2013) quanto Gropp, *et. al.* (1999) concordam sobre a importância da criação de uma biblioteca padrão capaz de fazer a interação entre o desenvolvedor e os recursos paralelos e convergem também sobre a adequação do modelo apresentado com a arquitetura que é utilizada em *clusteres*.

2.2. ScaLAPACK

O MPI traz uma forma do desenvolvedor interagir com arquiteturas de memória distribuída, porém essa interação deve ser feita com alguns cuidados para que o programa não apresente uma performance aquém do esperado. O ScaLAPACK (Blackford, *et. al.* 1997) possui funções de álgebra linear otimizadas para sistemas de memória distribuída.

A biblioteca ScaLAPACK utiliza duas outras bibliotecas para fornecer funções que resolvem problemas de álgebra linear em sistemas de memória distribuída (Pacheco, 1996). A primeira é o LAPACK (*Linear Algebra Package*) que viabiliza funções para a resolução de problemas de álgebra em sistemas de memória compartilhada. A segunda biblioteca é o BLAS (*Basic Linear Algebra Subprogram*) que oferece funções operações básicas como multiplicação vetor-matriz, multiplicação entre matrizes, entre outras.

Por se utilizar do LAPACK e entender suas funções para sistemas de memória distribuída, o ScaLAPACK (*Scalable Linear Algebra Package*, ou Pacote de Álgebra Linear Escalável) possui portabilidade e a otimização alcançada nas operações faz com que a biblioteca seja bastante utilizada pelo meio acadêmico em processamento de alta performance.

De acordo com Quinn (2003, p.211), o ScaLAPACK oferece uma variedade de funções como operações básicas em matrizes e vetores, resolução de sistemas lineares de equações e o cálculo de autovalores e autovetores. Trata-se inegavelmente de uma biblioteca de ampla utilidade em algoritmos comumente utilizados no meio acadêmico, além de ser uma biblioteca gratuita.

Pode-se dizer que as funções citadas por Pacheco (1996) e Quinn (2003) podem ser escritas utilizando apenas MPI e que a utilização da biblioteca ScaLAPACK traz algumas vantagens para o programa que contém as funções oferecidas, como: (i) eficiência; (ii) portabilidade e (iii) disponibilidade.

2.3. Análise em Componentes Principais (ACP)

A Análise em Componentes Principais é uma técnica que realiza redução dimensional dos dados sem grande perda de informação. Isso é feito eliminando a parte redundante da informação (Araujo, 2009).

Então, para encontrar os componentes principais é necessário: (i) padronização dos dados; (ii) cálculo da matriz de correlação; (iii) cálculo dos autovalores e autovetores e (iv) transformada de *Hotelling*.

3. Resultados Experimentais

3.1. Multiplicação de Matrizes

Para a realização dos testes com multiplicação de matrizes foi utilizada uma base de dados numéricos contendo 32,66MB de dados. Os testes foram feitos utilizando matrizes quadradas de dimensão variando entre quadrados perfeitos encontrados de 2.000 a 20.000 e número de processos variando entre quadrados perfeitos de 1 a 16.

No gráfico da Figura 1, cada ponto representa o tempo de execução da multiplicação em determinado tamanho de matriz e cada reta representa o número de processos que foi utilizado nas execuções. Então, analisando a Figura 1 destacamos que a reta representando as execuções que utilizaram oito processos teve desempenho melhor que as outras retas, porém esse ganho não é significativo. Também é interessante destacar que mesmo que exista um ganho de performance, as retas têm um padrão de crescimento similar fazendo com que não exista uma diferença significante entre a quantidades de processos em execução.

Ainda analisando a Figura 1 é possível destacar que a execução do programa com dezesseis processos teve a pior performance. Isso pode ter ocorrido principalmente porque apesar de muitos processos em execução, a arquitetura utilizada nos testes só possui um único processador (*core*). Por este motivo, a inserção de n processos em uma arquitetura como esta é prejudicial quanto ao desempenho computacional do programa em execução, quando o melhor caso para o uso do ScaLAPACK é quando se tem vários nós.

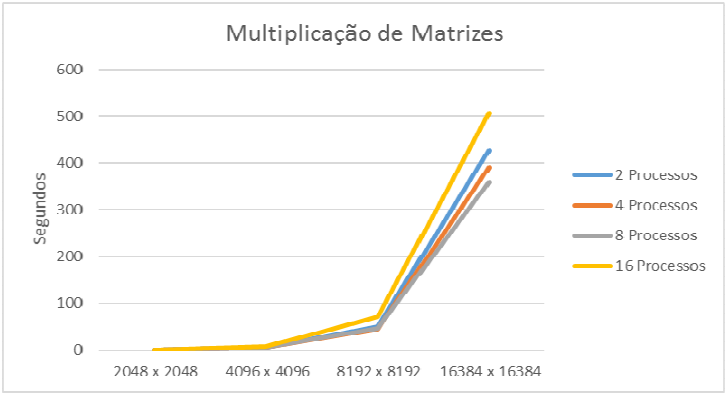


Figura 1. Tempos de execução do algoritmo de multiplicação de matrizes

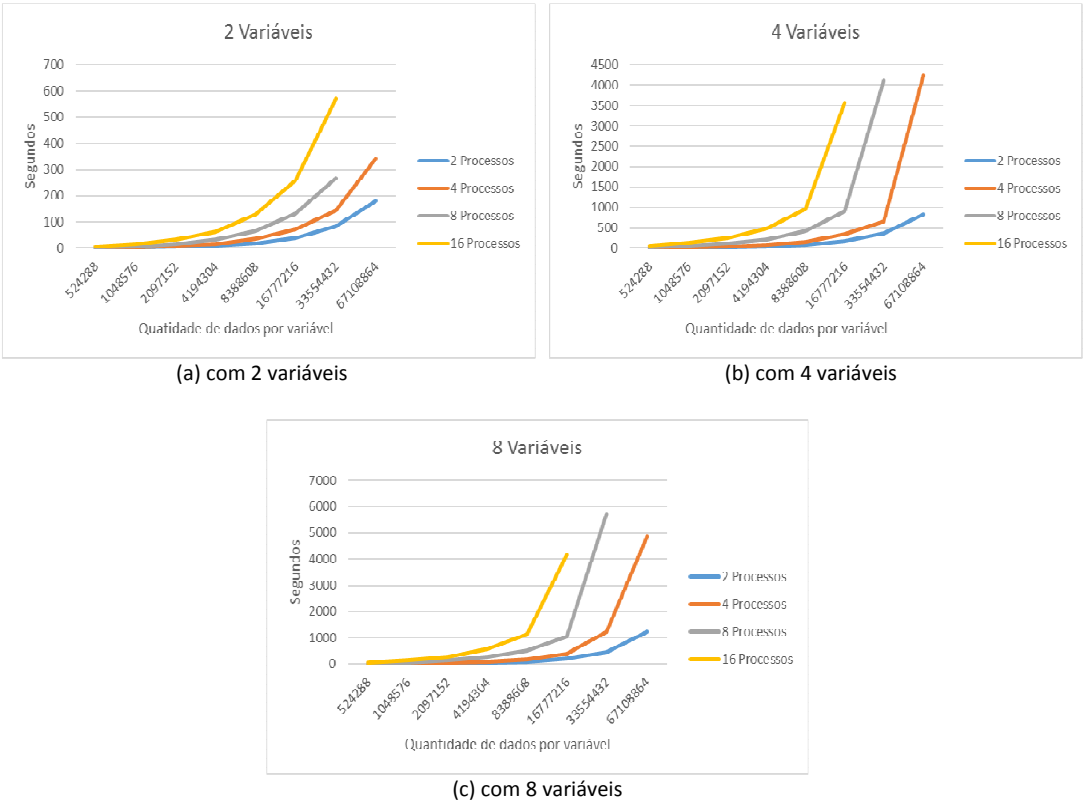
Também destacamos que o comportamento do crescimento das retas na Figura 1 é que existe um padrão de crescimento linear levando em consideração que o tamanho das matrizes utilizadas nos testes é sempre 4x menor que a sua subsequente.

Por fim, é imprescindível lembrar que os testes representados na Figura 1 foram executados em uma máquina em que só existe um nó de processamento e os processos são organizados em um único processador (*core*) o que resulta em pequeno ganho ou até mesmo em *overhead*. Então, espera-se que em uma arquitetura com mais de um nó a computação seja melhor distribuída entre esses nós o que possivelmente resultará em um melhor desempenho na execução do algoritmo.

3.2. Multiplicação de Matrizes com ACP

Igualmente para os testes de multipliação de matrizes (conforme Seção 3.1), também foi utilizada uma base de dados numéricos contendo 32,66MB de dados. Os dados de entradas foram lidos como uma matriz em que as linhas representavam as variáveis e as colunas as informações sobre as mesmas. Então, os testes foram feitos utilizando matrizes de entrada com números de linhas alternando entre 2, 4 e 8 e número de colunas alternando em quadrados perfeitos entre 500.000 e 100.000.000. Também alterou-se a quantidade de processos utilizados em cada bloco de testes usando os valores 1, 2, 4, 8 e 16.

Os testes foram divididos em três gráficos (a, b e c) apresentados na Figura 2, conforme a quantidade de variáveis que são representadas pelas linhas da matriz de entrada. A Figura 2(a) mostra os resultados obtidos na execução do ACP em uma matriz com apenas duas variáveis, onde cada linha representa o programa sendo executado por diferente número de processos.



(a) com 2 variáveis

(b) com 4 variáveis

(c) com 8 variáveis

Figura 2. Tempo de execução da multiplicação de matriz usando ACP

Observando a Figura 2(a), percebe-se que quanto maior o número de processos utilizados, pior é a performance do programa. Isso pode parecer não lógico devido ao fato de um maior número de processos organizarem melhor as partes do programa em diferentes nós, porém é necessário perceber que o ambiente onde o programa foi executado possui apenas um nó, ou seja, apenas um processador, então a criação de vários processos nesse caso adiciona complexidade (*overhead*) à execução sem haver uma compensação.

Analisando a Figura 2(b), destacamos que execução com 8 e 16 processos em uma matriz com 67.108.864 colunas não foi finalizada com sucesso devido a complexidade adicionada pelo aumento das variáveis. A utilização de uma arquitetura em que esses processos sejam divididos entre diferentes nós pode ser uma possível solução para este problema de execução. Também destacamos que o mesmo fato ocorreu quando a execução foi realizada com 8 variáveis (ver Figura 2(c)).

No entanto, é interessante destacar a disparidade quando se compara a execução utilizando dois processos com a execução que utiliza dezesseis processos, é visível que o *overhead* foi causado por causa do acréscimo de complexidade ao aumentar o número de processos criados em uma arquitetura com processador único. Por fim, a Figura 3 mostra a execução do ACP em matrizes com 2, 4 e 8 variáveis utilizando apenas 2 processos que foi constatado como melhor caso.

Observando a Figura 3, é perceptível ver o aumento de complexidade quando o número de variáveis é aumentado. É esperado que a execução deste algoritmo em uma arquitetura que contenha vários nós apresente melhores resultados e uma menor disparidade quando comparado os tempos de execução com quantidades de variáveis distintas.

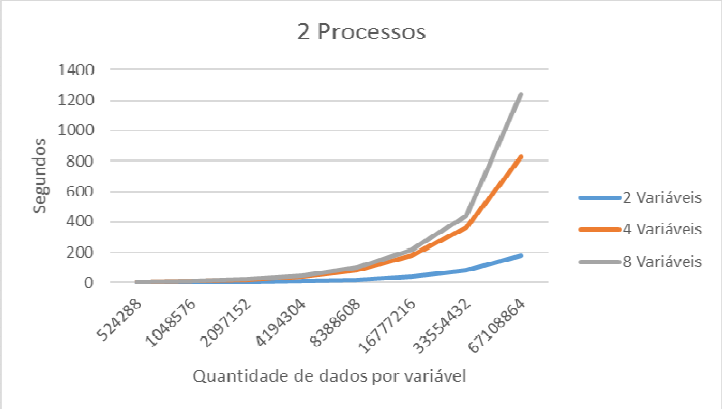


Figura 3. Tempo de execução do ACP utilizando 2 processo com 2, 4 e 8 variáveis

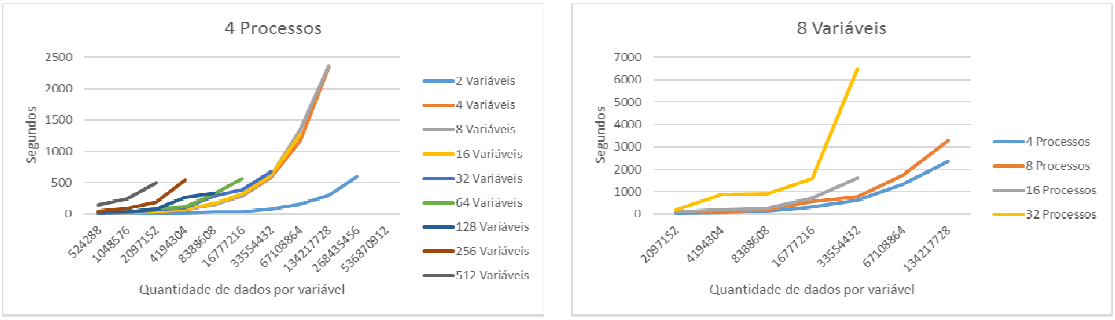
3.3. Multiplicação de Matrizes com ACP no Cluster

Para a realização das execuções no *cluster* foram gerados em tempo de execução dados numéricos aleatórios, a plataforma usada foi o *cluster* Cristal localizado no Departamento de Computação (DComp) da Universidade Federal de Sergipe (UFS). Os dados de entradas foram lidos como uma matriz em que as linhas representavam as variáveis e as colunas as informações sobre as mesmas.

Os testes foram feitos utilizando matrizes de entrada com número de linhas alternando entre 2, 4, 8, 16, 32, 64 e 128 e número de colunas alternando em quadrados perfeitos entre 2 a 600.000.000 de acordo com a capacidade do hardware de processar esses dados. Além do tamanho da matriz, alterou-se também a quantidade de processos utilizados em cada bloco de testes usando os valores 4, 8, 16 e 32.

A Figura 4(a) apresenta parte dos testes executados no *cluster* Cristal e mostra a quantificação de dados computados no *cluster* que foi maior que nos testes realizados anteriormente. Pode-se perceber que com 8 variáveis, o *cluster* conseguiu computar uma matriz com 134.217.728 colunas resultando em uma matriz de mais de um bilhão de dados. Além disso, pode-se aumentar o número de variáveis para 512 conseguindo ainda sucesso nas execuções, algo não atingido nas outras execuções.

O programa utilizando a biblioteca ScaLAPACK se mostrou mais escalável no *cluster*. Outro cenário importante para a comparação é o efeito gerado a partir da mudança de quantidade de processos na execução como é mostrado na Figura 4(b). Execuções com 2, 64 e 128 processos também foram executados, mas não apresentaram resultados coesos, ou seja, algumas execuções apresentaram resultado e outros erros. Ainda observa-se na Figura 4(b) que as execuções que utilizaram 4 e 8 processos tiveram um crescimento mais contido em relação às outras configurações.



(a) com 4 processos (b) com 8 processos

Figura 4. Biblioteca ScaLAPACK executando uma matriz no *cluster* Cristal

4. Conclusões e Trabalhos Futuros

O desenvolvimento deste trabalho possibilitou uma análise do uso de ferramentas paralelas, no caso o MPI e o ScaLAPACK, na execução do algoritmo ACP de forma paralela. Ao mesmo tempo o trabalho ampliou a noção de que é necessário um ambiente apropriado para que a execução de códigos paralelos sejam eficientes.

De forma geral, foi demonstrado que a utilização de ferramentas paralelas em um ambiente (hardware) que não dá suporte aos recursos necessários, como por exemplo a disponibilidade de vários processadores independentes, estará acrescentando complexidade ao programa e por consequência aumentando o tempo de execução. Então, é válida a idéia do uso do ScaLAPACK como instrumento para aumentar o desempenho de um algoritmo que executa rotinas de álgebra linear, porém é necessário prover recursos suficientes para que todo o potencial da biblioteca possa ser extraído. Portanto, a utilização de técnicas de desenvolvimento paralelo tem o potencial de aumentar o desempenho no processo de obtenção de componentes principais.

Dada a necessidade de investigar mais profundamente o tema, sugere-se como trabalhos futuros: (i) configurar um ambiente que atenderá todos os requisitos para explorar o potencial do ScaLAPACK; (ii) execução do ACP em um *cluster* embarcado; (iii) utilizar outros métodos de cálculo numérico para identificação dos autovalores e dos autovetores, tais como: método das Potências, ou o método de Leverrier.

Referências

Araujo, W. O. de (2009) “Análise de Componentes Principais (PCA)”. Centro Universitário de Anápolis, Relatório Técnico RT-MSTMA_003-09, Maio, 2009, 12p.

Asanović, K.; Bodik, R.; Catanzaro, B. C.; Gebis, J. J.; Husbands, P.; Keutzer, K.; Patterson, D. A.; Plishker, W. L.; Shalf, J.; Williams, S. W. and Yelick, K. A. (2006) “The Landscape of Parallel Computing Research: A View from Berkeley”. EECS Department, University of California, Berkeley, Technical Report Nº UCB/EECS-2006-183, December 18, 2006, 56p.

Blackford, L. S.; Choi, J.; Cleary, A.; D'Azevedo, E.; Demmel, J.; Dhillon, I.; Dongarra, J.; Hammarling, S.; Henry, G.; Petitet, A.; Stanley, K.; Walker, D. and Whaley, R. C. (1997) “ScaLAPACK Users Guide. Series: Software, Environments and Tools”. Book Code, 345p.

Gropp, W.; Lusk, E.; Skjellum, A. (1999) “Using MPI: Portable Parallel Programming with the Message-Passing Interface”. Cambridge: MIT Press, 2nd edition, 350p.

Jolliffe, I. T. (2002) “Principal Component Analysis, Series: Springer Series in Statistics”. New York: Springer, 2nd edition, 487p.

Pacheco, P. S. (2011) “An Introduction to Parallel Programming”. San Francisco: Elsevier, 1st edition, 392p.

Pacheco, P. S. (1996) “Parallel Programming with MPI”. San Francisco: Morgan Kaufmann, 1st edition, 500p.

Quinn, M. J. (2003) “Parallel Programming in C with MPI and OpenMP”. Mc Graw Hill, 1st edition, 544p.

Rauber, T.; Rünger, G. (2013) “Parallel Programming for Multicore and Cluster Systems”. New York: Springer, 2nd edition, 529p.

Saturnus: Um Simulador de Sistemas de Arquivos Paralelos

Lucas P. Bordignon[1], **Eduardo C. Inacio**[1],
Marcos A. Rodrigues[2] , **Mario A. R. Dantas**[1]

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

[2]Sheffield Hallam University (SHU)
Sheffield, U.K.

lucas.bordignon@grad.ufsc.br, eduardo.camilo@posgrad.ufsc.br
M.Rodrigues@shu.ac.uk, mario.dantas@ufsc.br

Abstract. *This paper has as it main goal to introduce the Saturnus, a parallel file systems (PFS) simulator. The proposal consists in, utilizing it, extracts data with an agility and reliability related with the workload of such systems. Doing so, the simulador is capable of helping on the decision taking process of the main factors and aspects that impact a PFS performance.*

Resumo. *O presente trabalho tem como foco introduzir o Saturnus, um simulador de sistemas de arquivos paralelos. A proposta consiste em utilizar o mesmo para extrair, de modo ágil e confiável, dados e informações sobre o comportamento de tais sistemas, dando foco em aspectos de balanceamento de carga. Assim sendo, o mesmo é capaz de auxiliar na tomada de decisão com relação aos fatores que mais impactam no desempenho de aplicações desse escopo.*

1. Introdução

Sistemas de arquivos paralelos (SAPs) tem sido amplamente utilizados pela comunidade acadêmica como uma forma de extrair o melhor desempenho e contornar problemas relacionados ao desempenho de entrada e saída de aplicações. Diversas implementações existem nos dias de hoje, como é o caso do Lustre [J. Braam and Schwan 2002] e do PVFS [Ligon and Ross 1996]. A utilização de tais sistemas tem influenciado diversos profissionais a estudarem sobre o tópico, tornando-se alvo de pesquisas recentes na área de computação de alto desempenho e sistemas distribuídos.

Porém, identificar e definir quais são os aspectos e configurações que causam impactos, tanto positivos como negativos, sobre um SAP é uma árdua tarefa dado que cada elemento de sistemas como tal se comportam de maneiras distintas sob certos parâmetros [Inacio et al. 2015b]. Outro aspecto muito relevante é que a realização de pesquisas e experimentos sobre SAPs são, em sua maioria, custosos tanto em questão de hardware, quanto no aspecto do tempo gasto com modelagem e prototipação de soluções.

Um exemplo de aplicação na qual pode beneficiar-se de tal abordagem é apresentada em [Siddeq and Rodrigues 2016]. Em projetos como esse, onde o tempo gasto com o armazenamento, transmissão e, principalmente, processamento (compressão e descompressão) de dados é crucial, se faz necessário o uso de diversas ferramentas que sirvam de base para obter resultados cada vez melhores por meio de experimentos.

Assim, o presente artigo apresenta um simulador de SAPs, chamado *Saturnus*. O mesmo foi desenvolvido com o intuito de facilitar o estudo e o entendimento do comportamento desses sistemas, dando enfoque principalmente em aspectos relacionados ao balanceamento de carga.

Na seção 2 exploramos o funcionamento de sistemas de arquivos paralelos reais. Na seção 3, trabalhos correlatos e que servem de base para o projeto serão abordados. A estrutura interna do simulador será apresentada na seção 4, bem como experimentos e dados na seção 5, expondo considerações finais na seção 6.

2. Fundamentação Teórica

Processar e analisar grandes quantidades de dados é uma tarefa complexa e que demanda alto custo computacional, principalmente dada a crescente quantidade de informações gerada todos os dias ao redor do mundo. Com isso, desenvolver aplicações para auxiliar e tratar tais problemas tem se tornado uma árdua tarefa.

Dificuldades em conciliar alto desempenho em aglomerados computacionais, como clusters e grids, com eficiência energética, considerando limitações térmicas, de consumo de energia e de componentes utilizados em tais sistemas são apresentados em [Inacio and Dantas 2014]. Além do desenvolvimento, o uso dessas ideias e técnicas, de modo que aplicações tenham um desempenho ótimo, também é um trabalho árduo, dado que para tomar as melhores decisões é necessário um entendimento muito claro de todos os aspectos e conceitos que norteiam esses sistemas. Sistemas de arquivos paralelos tem como base três componentes principais:

Máquinas Clientes geram requisições de escrita e leitura de arquivos no sistema.
Servidores de Dados recebem requisições, processam e armazenam os dados.
Servidores de Metadados contém dados sobre os objetos presentes nos servidores de dados.

A comunicação entre máquinas clientes e servidores de dados e metadados é feita através de uma rede de interconexão. Quando se faz necessário armazenar algum dado, as máquinas clientes particionam o mesmo em diversas partes, chamadas de faixas, de tamanho configurável, e as distribuem entre os servidores de dados, diferentemente de sistemas de arquivos distribuídos, nos quais arquivos como um todo são enviados para serem armazenados.

Embora vários sistemas existam, diversas propriedades não implementadas ou complexidades de uso acarretam e incentivam grupos de pesquisa e desenvolvimento a criarem seus próprios sistemas, com focos em aspectos específicos. Para tal, o uso de simuladores torna-se uma alternativa muito viável a efetivamente prototipar e testar um modelo como um todo para obter resultados. Além disso, em virtude do alto custo de equipamentos com grande poder de processamento, certos grupos acabam voltando seus olhos a técnicas que possam contornar essa carência. Uma delas é a virtualização, entretanto, [Inacio et al. 2015a] nos mostra que a utilização de tais abordagens podem acarretar em quedas significativas de desempenho, chegando até a 93% de perda em requisições de escritas de dados.

3. Trabalhos Correlatos

Atualmente, uma grande diversidade de trabalhos relacionados a simulação de sistemas de arquivos paralelos existem na literatura. [Yonggang et al. 2013,

E. Molina-Estolano et al. 2009] são exemplos de tal, entretanto, o foco principal de cada um deles não é a mesma da presente neste artigo. [Yonggang et al. 2013] implementa uma abordagem voltada para algoritmos de agendamento de operações de entrada e saída, enquanto [E. Molina-Estolano et al. 2009] foca em aspectos relacionados ao sincronismo de dados, como localidade de dados, técnicas de *locking*, e métodos de replicação de dados.

Uma outra abordagem presente é mostrada por [Carns et al. 2008], na qual o autor modela de uma maneira mais aprofundada e precisa aspectos relacionados a redes de intercomunicação, como uma camada de rede TCP/IP, baseada no framework OMNeT++. Entretanto, o mesmo detalha aspectos presentes no sistema de arquivos paralelos PVFS de maneira mais aprofundada se comparado com o presente projeto, tornando o simulador inflexível. [Settlemyer 2009] introduz o HECIOS, simulador que modela de uma forma muito mais complexa os elementos de cache do sistema, reduzindo a flexibilidade do software e gerando custos relacionados ao tempo necessário para realizar sua utilização e com foco em tópicos distintos, como a distribuição da carga de trabalho, por exemplo.

4. Saturnus

Baseado no SAP existente chamado OrangeFS (antigo PVFS), o *Saturnus* foi desenvolvido utilizando uma abordagem de simulação conhecida como simulação discreta, na qual o simulador possui um contador interno que realiza o registro de operações em sistemas de arquivos paralelos, executa cálculos para estimar a quantia de tempo necessária para executar tal operação e modifica o contador interno. Consequentemente, não há a necessidade de aguardar o término de requisições de escrita ou leitura em um ambiente real, agilizando assim a obtenção de dados e a tomada de decisão com base nos mesmos. Além disso, requisições são mapeadas para eventos internos ao simulador que, ao longo do processo de simulação, são agendados e modificam o estado da mesma.

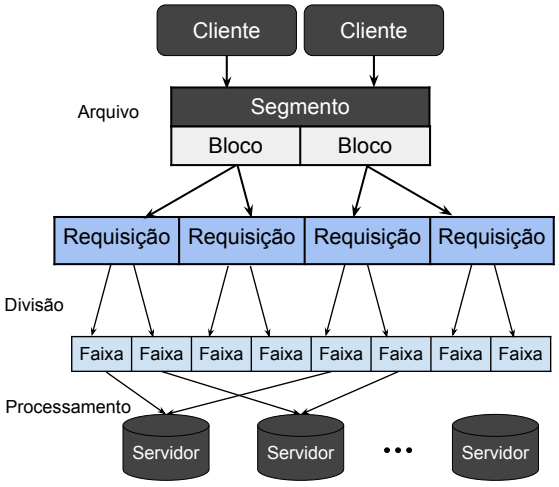


Figura 1. Funcionamento do processo de simulação.

Atualmente, requisições de escrita e leitura de dados estão implementadas e mapeadas para eventos. Além disso, certos aspectos presentes no sistema utilizado como base não estão modelados, como a camada de rede e servidores de metadados. O projeto foi desenvolvido utilizando o framework de simulação DESMO-J[Göbel et al. 2013] e seu código fonte encontra-se disponível na internet (https://github.com/lapesd/saturnus). Seu funcionamento, ilustrado na Figura 1 se baseia em nove parâmetros de entrada:

Tabela 1. Parâmetros utilizados para experimentação

Parâmetros	Valores
número de clientes	1, 2, 4
número de segmentos	1
tamanho de bloco (em MiB)	1, 2, 4, 8, 16, 32, 64
tamanho de requisição (em MiB)	1, 2
tamanho de faixa (em KiB)	64
tipo de arquivo	Compartilhado e Arquivo por Processo
padrão de acesso	Sequencial

Número de clientes quantidade de máquinas clientes no aglomerado computacional.
Número de segmentos número de segmentos para cada arquivo.
Número de servidores de dados quantidade de nodos de armazenamento.
Número de faixas número de servidores usados para armazenar faixas de um arquivo.
Tamanho de bloco tamanho em bytes de um bloco de arquivo.
Tamanho de requisição tamanho de cada requisição de acesso a um arquivo.
Tamanho de faixa tamanho das faixas, distribuídas entre nodos de armazenamento.
Padrão de acesso modo de acesso ao arquivo. Sequencial ou aleatório.
Tipo de arquivo arquivo compartilhado entre clientes ou único por processo.

Inserida a configuração inicial, instâncias das entidades que representam máquinas Cliente e Servidores de Dados são criadas. Inicialmente, servidores de dados não possuem requisições a serem tratadas. Tais requisições são criadas, com base nas configurações iniciais do simulador, e são distribuídas, agendadas e executadas internamente, gerando mudanças de estado do simulador e causando o desenvolvimento do processo de simulação.

Ao final desse processo, um relatório com informações detalhadas sobre cada uma das requisições geradas, como momento de envio, momento de atendimento da requisição, tempo de execução, entre outros, é disponibilizado ao usuário e o mesmo pode extrair dados sobre o comportamento de determinado conjunto de configurações em um sistema de arquivos paralelos, dando foco ao balanceamento de carga.

5. Resultados Experimentais

Os resultados apresentados através do presente trabalho de pesquisa auxiliam na visualização do potencial do simulador, além de expor sua proximidade a resultados contidos em trabalhos presentes na literatura atual com relação a SAPs. Além disso, configurações como o tamanho de faixa foram testados com valores padrão do sistema OrangeFS, SAP escolhido como base para o *Saturnus* por seu amplo uso no meio acadêmico. Combinações de parâmetros presentes na Tabela 1 foram utilizadas nos experimentos, sendo replicados aproximadamente 30 vezes cada.

A Figura 2 simula o comportamento do sistema conforme mais servidores de dados são inseridos ao mesmo, fazendo uso de cargas de tamanho variado. Para esse experimento, o número de máquinas clientes foi fixado em 2 e foi utilizado um valor para a contagem de faixa igual ao número de nodos de dados. Pode-se perceber que, para determinadas cargas, a inserção de mais servidores não causa um impacto tão grande no desempenho do sistema. Esse efeito se deve as políticas utilizadas por SAPs para manter a sincronia de escrita e leitura de requisições.

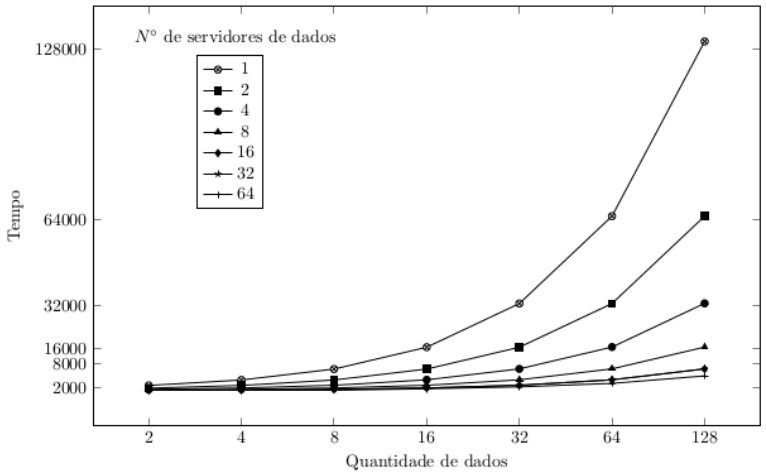


Figura 2. Aplicação de cargas diversas sobre o simulador.

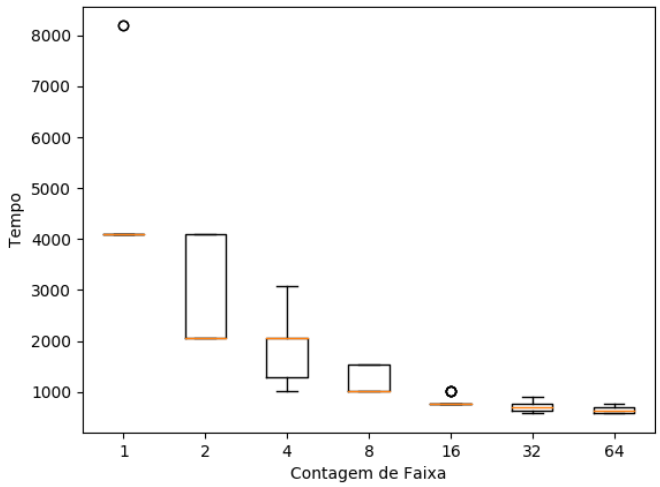


Figura 3. Influência da aleatoriedade na seleção de servidores de dados.

A Figura 3 nos mostra a influência da contagem de faixa, parâmetro que sinaliza ao sistema a quantidade de nodos de dados a serem utilizados para tratar requisições geradas por máquinas clientes, sobre o tempo de execução das requisições.

Idealmente, conforme a quantidade de nodos de dados dobra, o tempo de execução deve se reduzir a metade, dado que mais recurso computacional estará disponível para tratar requisições do sistema de arquivos. Porém, na prática, tal efeito não é observado. Dado que a seleção de servidores de dados é feita de maneira aleatória, não há garantia que máquinas clientes distintas selecionem nodos de dados distintos. Com isso, em diversos casos, requisições originadas de clientes diferentes concorrem por um mesmo recurso, atrasando a emissão de novas requisições e gerando desbalanceamento de carga.

6. Considerações Finais

Como mostrado através do presente artigo, o simulador *Saturnus* é capaz de fornecer dados relacionadas ao comportamento de um sistema de arquivos paralelos, com

foco principal no balanceamento de carga, expondo aspectos que podem influenciar no mesmo. Para futuros trabalhos, a realização de mais testes para validar o modelo, além da implementação de camadas presentes em sistemas reais e que ainda não foram modeladas (como a camada de rede de interconexão) são os focos principais.

Além disso, agradecemos ao fundo Newton (Newton Fund)/FAPESC pela parceria e apoio durante todo o período de desenvolvimento do projeto.

Referências

Carns, P. H., Settlemyer, B. W., and Ligon, W. B. (2008). Using server-to-server communication in parallel file systems to simplify consistency and improve performance. *SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–8.

E. Molina-Estolano, C. M., Bent, J., and Brandt, S. A. (2009). Building a parallel file system simulator. *J. Phys.: Conf. Ser*.

Göbel, J., Joschko, P., Koors, A., and Page, B. (2013). The discrete event simulation framework desmo-j: Review, comparision to other frameworks and latest development. *Proc. - 27th European Conf. on Modelling and Simulation, ECMS 2013*, pages 100–109.

Inacio, E. C. and Dantas, M. A. R. (2014). A survey into performance and energy efficiency in hpc, cloud and big data environments. *International Journal of Networking and Virtual Organisations*, pages 299–318.

Inacio, E. C., Dantas, M. A. R., and de Macedo, D. D. J. (2015a). Towards a performance characterization of a parallel file system over virtualized environments. *IEEE Symposium on Computers and Communication (ISCC)*, pages 595–600.

Inacio, E. C., Pilla, L. L., and Dantas, M. A. R. (2015b). Understanding the effect of multiple factors on a parallel file system's performance. *WETICE '15 Proceedings of the 24th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 90–92.

J. Braam, P. and Schwan, P. (2002). Lustre: The intergalactic file system. *Ottawa Linux Symposium, Ottawa, ON*.

Ligon, W. B. and Ross, R. B. (1996). Implementation and performance of a parallel file system for high performance distributed applications. *Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*, pages 471–480.

Settlemyer, B. (2009). A study of client-based caching for parallel i/o. *Tese de Doutorado. Clemson*.

Siddeq, M. M. and Rodrigues, M. A. (2016). 3d point cloud data and triangle face compression by a novel geometry minimization algorithm and comparison with other 3d formats. *Proceedings of the International Conference on Computational Methods*, pages 379–394.

Yonggang, L., Renato, F., Yiqi, X., and Ming, Z. (2013). On the design and implementation of a simulator for parallel file system research. *IEEE Symposium on Mass Storage Systems and Technologies*.

Uma Análise do *overhead* Introduzido pelo Sistema Operacional Nanvix na Execução de Cargas de Trabalho

Davidson Francis G. Lima, Pedro H. Penna, Henrique C. Freitas

[1]Grupo de Arquitetura de Computadores e Processamento Paralelo (CArT)
Pontifícia Universidade Católica de Minas Gerais

{davidson.francis,pedro.penna}@sga.pucminas.br, cota@pucminas.br

Abstract. *An operating system has many purposes including the resource management that it provides. In this respect, the allocation of CPU time between processes is fundamental. The way each system does it could interfere directly in the execution of applications. In this context, this work aims to analyze the Linux and Nanvix systems through two kernels and six metrics, as well as some changes in the process scheduler. The results show that for workloads with regular memory access, Nanvix performs* 158.5% *better than Linux.*

Resumo. *Um sistema operacional tem muitos propósitos incluindo a gerência de recursos que ele proporciona. Nesse aspecto a alocação do tempo de CPU entre processos é fundamental. O modo como cada sistema o faz pode interferir diretamente na execução de aplicações. Nesse contexto, esse trabalho visa analisar os sistemas Linux e Nanvix por meio de dois kernels e seis métricas além de alterações no escalonador de processos. Os resultados mostram que para cargas de trabalho com padrões regulares de acesso à memória, o Nanvix obteve um desempenho superior ao Linux de* $158,5\%$.

1. Introdução

No contexto de compartilhamento de tempo de processador, vários algoritmos de escalonamento já foram estudados, como FIFO (*First-In, First Out*) e SJF (*Shortest Job First*). No entanto, dentre os clássicos que recaem sobre o conceito de preempção, destacam-se o *Round-Robin* e Múltiplas Filas de Prioridades [Tanenbaum and Woodhull 2005].

Independente do algoritmo escolhido, o escalonador de processos exerce influência direta no desempenho do sistema operacional em questão, uma vez que ele impõe a política de como os processos serão executados. Por exemplo, no algoritmo de escalonamento *Round-Robin* uma escolha não adequada do *quantum* de escalonamento, pode conduzir a uma situação em que o sistema realiza demasiadas trocas de contexto, uma tarefa custosa que envolve salvar e carregar registradores, mapas de memória e tabelas de página; introduzindo assim um *overhead* significativo no desempenho de aplicações [Tanenbaum and Woodhull 2005].

Tendo em vista a significância do escalonador de processos no desempenho de um sistema operacional, o presente trabalho tem como objetivo estudar e avaliar o desempenho dos escalonadores de dois sistemas operacionais: Linux e Nanvix. O primeiro trata-se de um sistema operacional de propósito geral criado originalmente por Linus Torvalds na década de 90. Esse sistema possui um *kernel* de código aberto inspirado no sistema Unix e que rapidamente ganhou aceitação da comunidade científica e industrial. O Linux trata-se

de um sistema maduro, moderno e completo, que devido à sua portabilidade é encontrado em uma grande variedade de dispositivos. Por outro lado, o Nanvix é um sistema operacional completo desenvolvido em meados de 2011 com propósito educacionais [Penna 2017]. O Nanvix é um sistema Unix-like de 32 bits que possui interface compatível com POSIX, arquitetura baseada no Unix System V e possui suporte a multiprogramação, comunicação interprocessos, memória virtual com paginação, interface uniforme de dispositivos além contar com suporte a biblioteca C. Recentemente, o Nanvix vem recebendo contribuições e está gradativamente sendo modificado para endereçar arquiteturas *many-core* emergentes.

Neste trabalho, o escalonador de processos dos sistemas operacionais Linux e Nanvix são avaliados por meio de duas cargas de trabalho sintéticas e seis diferentes métricas. As contribuições deste trabalho consistem em: (i) implementação de um módulo de acesso aos contadores de hardware para o Nanvix; e (ii) a avaliação do escalonador de processos dos sistemas supracitados. No Linux o escalonador de processos tem seu desempenho analisado por meio da biblioteca PAPI [Mucci et al. 1999]. Já no Nanvix, o mesmo algoritmo é avaliado por meio do módulo implementado nesse trabalho, que possibilita acesso aos contadores de hardware da máquina subjacente. Para realizar esse estudo comparativo, execuções de dois *kernels* numéricos, pertencentes ao domínio de Álgebra Linear Densa, foram considerados.

2. Trabalhos Relacionados

Uma comparação de desempenho entre o escalonador de processos de diferentes sistemas operacionais de propósito geral é apresentada por [Chen et al. 1995]. A partir de dados coletados por meio de contadores de hardware nativos da arquitetura considerada (Intel Pentium), os Sistemas Windows Workgroups 3.11, Windows NT 3.5 e NetBSD 1.0 foram analisados frente a uma suíte de *microbenchmarks* e cargas de trabalho sintéticas. No geral, os experimentos relevaram: (i) uma diferença de desempenho de $2\times$ a $7\times$ entre os sistemas estudados; e (ii) que a funcionalidade dos subsistemas de cada um dos sistemas operacionais estudados impacta diretamente nos resultados observados.

Em um contexto mais específico, que refere-se a aplicações de alto desempenho executando em arquiteturas massivamente paralelas, o *overhead* adicionado pelo sistema operacional subjacente é um dos principais fatores que impactam na sincronicidade entre as *threads* de uma aplicação paralela. No trabalho de [Beckman et al. 2008] um estudo do *overhead* de desempenho em aplicações de usuário foi realizada no sistema IBM BG/L, em diferentes plataformas. Para tanto, *microbenchmarks* proprietários e uma biblioteca de monitoramento de desempenho foram desenvolvidos. Os experimentos conduzidos revelaram que o *overhead* introduzido pelo sistema deve ser necessariamente substancial, para que impactos significativos no desempenho sejam observados. Seguindo uma abordagem semelhante, mas visando uma análise da arquitetura, [Ferreira et al. 2013] estudaram a sensibilidade de aplicações reais a ruídos do sistema operacional em diversos tipos de configurações de nós em um sistema de larga escala Cray XT3/4. De forma complementar ao estudo de [Beckman et al. 2008], os resultados revelaram que mínimas modificações na topologia da rede de interconexão são capazes de impactar significativamente no desempenho dos subsistemas do sistema operacional.

Esse trabalho se propõe a averiguar o desempenho de *kernels* com diferentes

métricas, como a mispredição de desvios, vazão de instruções e acessos ao último nível de *cache*. Além disso, assim como nos trabalhos correlatos, esse trabalho propõe-se a avaliar o desempenho de diferentes modificações nos escalonadores de processos dos sistemas estudados, como por exemplo alterações no *quantum* de escalonamento.

3. Sistemas Operacionais Estudados

3.1. Linux

O Sistema Operacional Linux utiliza o escalonador CFS (*Completely Fair Scheduler*) para o escalonamento de processos. o CFS têm como principal objetivo ser o mais justo possível para qualquer tipo de aplicação, seja ela *IO/Bound* ou *CPU/Bound* [Love 2010].

O CFS diferente de outros escalonadores, não apenas realiza preempção em uma quantidade de tempo fixo ou dinâmico para cada processo, ele por sua vez, divide a utilização da CPU entre todos os processos que estão sendo executados. A fatia de tempo calculada para cada processo é baseada em um valor fixo que se baseia na latência desejada que o sistema deve possuir, e não na quantidade de tempo mínimo que os processos devem executar. Consequentemente, isso garante com que aplicações *IO-Bound* tenham uma alta responsividade mesmo que tenham que competir com programas *CPU-Bound*.

Da mesma forma que em outros sistemas Unix-like, o CFS também faz uso do `nice`[1] para aumentar ou diminuir a prioridade dos processos ao longo da sua execução, mas ao contrário dos outros sistemas, o `nice` serve como um 'peso' que irá influenciar na proporção de CPU distribuída para um determinado programa.

3.2. Nanvix

O escalonador do Nanvix implementa um esquema de Múltiplas Filas de Prioridades. As diferentes filas de processos em espera são utilizadas para separar os processos de acordo com o tipo de recurso que ele aguarda, recursos com um grande tempo de espera terão uma maior prioridade sobre os demais. A prioridade efetiva de um processo é dinâmica e composta por três valores: prioridade base, prioridade dinâmica e prioridade de usuário.

A prioridade base é atribuída ao processo de acordo com seu estado atual. Um processo em execução possui prioridade de usuário ao passo que processos que aguardam por recursos do sistema possuem prioridades mais elevadas de acordo com recurso pelo qual o processo espera. A prioridade dinâmica é ajustada à medida que o tempo de espera do processo aumenta no *kernel*. Esse mecanismo é utilizado para priorizar processos que estão esperando há mais tempo, de forma a evitar *starvation*. Por fim, a prioridade de usuário, ajustável por meio da chamada de sistema `nice()`, de forma similar à sistemas baseados em Unix esta prioridade, permite que o usuário diminua ou aumente a prioridade de um processo de modo a fazer um ajuste fino no sistema.

A todo momento que o sistema precisa escalonar um novo processo em execução, o primeiro processo na fila de mais alta prioridade é selecionado e o escalonador atribui a esse processo uma fatia de tempo de processamento. O *quantum* de tempo entregue aos processos é fixo e independente da prioridade efetiva do processo. O valor padrão do *quantum* está definido para 50 ciclos de *clock* mas pode ser ajustado por meio de uma constante definida globalmente no *kernel*.

[1]Chamada de sistema que permite ao usuário definir a prioridade de um processo.

Tabela 1. Eventos utilizados e equivalências

Evento HW	Descrição
BRANCH_INSTRUCTION_RETIRED	Instruções de desvio condicionais executados pela aplicação
BRANCH_MISSES_RETIRED	Instruções de desvio condicionais não previstos pela CPU
UNHALTED_CORE_CYCLES	Quantidade total de ciclos pela CPU para execução da aplicação
UNHALTED_REFERENCE_CYCLES	Mesmo que UNHALTED_CORE_CYCLES mas despreza variações de clock
INSTRUCTION_RETIRED	Quantidade de instruções executadas por intervalo de tempo
LLC_REFERENCE	Quantidade de referências realizadas na cache de nível mais alto

4. Projeto Experimental

Dois *kernels* de aplicações foram estudados frente a seis diferentes métricas de desempenho. O primeiro *kernel*, intitulado Matrix Multiplication (MM), efetua a multiplicação de duas matrizes densas utilizando o algoritmo de multiplicação trivial. Esse *kernel* realiza $O(n^3)$ operações de multiplicação em ponto flutuante de precisão dupla e se caracteriza pela computação intensa, acesso regular à memória e alta afinidade espacial dos dados. Por outro lado, o segundo *kernel* batizado Lower Upper (LU), efetua a resolução de equações lineares usando o Método de Decomposição LU. Assim como o *kernel* MM, o *kernel* LU também se caracteriza pela computação intensa, no entanto o padrão de acesso à memória é irregular e a afinidade espacial de dados é baixa.

A coleta das métricas selecionadas (Tabela 1) foi feita por meio dos contadores de hardware disponíveis na arquitetura experimental em modo usuário. No sistema Linux, a coleta dessas métricas foi realizada por meio do PAPI [Mucci et al. 1999], uma biblioteca de usuário que oferece acesso aos contadores de hardware por meio de uma interface de programação independente da plataforma. Já no Nanvix a coleta dessas métricas foi realizada através da chamada de sistema `acct()`, introduzida durante a elaboração da pesquisa exposta no presente trabalho para que o acesso aos contadores de hardware fosse possível. Na implementação feita, um módulo de monitoramento de desempenho foi integrado ao *kernel* do nanvix, do modo a possibilitar a leitura individual de contadores de hardware para cada processo.

Os experimentos foram executados em uma máquina dedicada Intel Core i7 2600 3.4GHz, com 8GB de memória RAM. Os *kernels* estudados foram os Linux v4.1.6 com Busybox v1.23.2 e Nanvix v1.3, e os *benchmarks* sintéticos analisados foram compilados com o GCC v5.3.0 sem flags de otimização. No total, 10 repetições de cada experimento foram realizadas para garantir acurácia nos resultados observados.

5. Resultados

Nos parágrafos seguintes são apresentados e discutidos os resultados obtidos no Linux (L) e Nanvix com as configurações de quantum default (N-QD, 50 ciclos), quantum 100 ciclos (N-Q100) e 200 ciclos (N-Q200) para os *kernels* LU e MM.

A Figura 1 apresenta os resultados obtidos para o Kernel LU. A Figura 1 (a) representa o total de instruções executadas, no Linux foram executadas $34,51\%$ a mais que no Nanvix. A Figura 1 (b) representa o total de acessos à cache L3. De forma similar, o Linux teve $16,9\%$ mais acessos à cache que no Nanvix, o que pode ser devido a diferenças no gerenciador de memória e no escalonador de processos. A Figura 1 (c) e a Figura

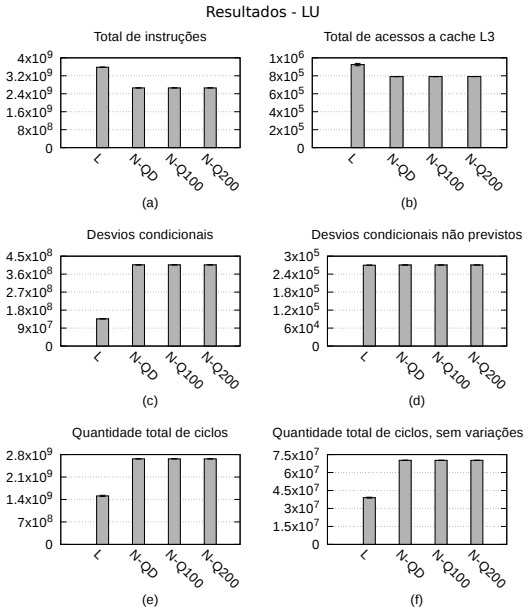


Figura 1. Resultado - LU

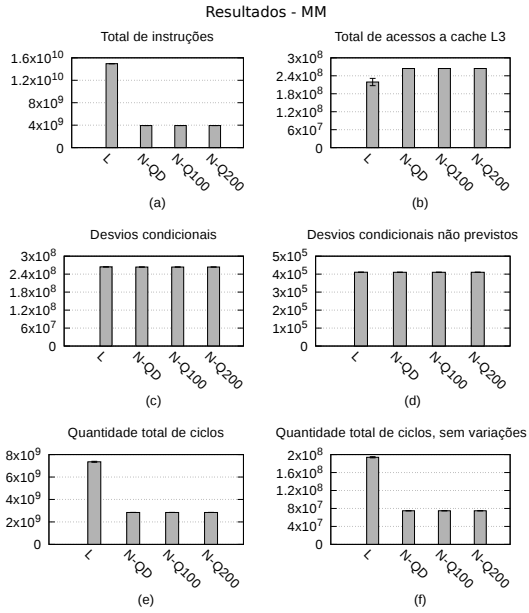


Figura 2. Resultado - MM

1 (d) apresentam as instruções de desvios condicionais total e desvios condicionais não previstos, respectivamente. Em (c) a quantidade de desvios no Nanvix é cerca de 198% superior ao obtido no Linux. Esse resultado sugere diferenças no binário final gerado pelo compilador. Em (d), os desvios não previstos foram similares nos dois sistemas, o que é de se esperar, pois é uma métrica mais dependente do *benchmark* do que o sistema operacional. A Figura 1 (e) e a Figura 1 (f) apresentam a quantidade total de ciclos com e sem variações de clock. Tanto em (e) quanto em (f) os gráficos apresentam comportamento similar – uma vez que o ajuste dinâmico da frequência do processador foi desativado durante a execução dos testes – com o Nanvix cerca de $76,27\%$ superior em quantidade de ciclos com variação e $79,91\%$ sem variação de clock. Em todas as métricas avaliadas com o *kernel* LU, pôde-se perceber que não houveram melhorias de desempenho para as configurações de quantum 50, 100 e 200 ciclos, o que indica que para as configurações de ciclos mencionadas acima a carga de trabalho não foi sensível.

A Figura 2 apresenta os resultados obtidos para o kernel MM. A Figura 2 (a) representa o total de instruções executadas, no Linux foram executadas $279,69\%$ instruções a mais que os resultados obtidos no Nanvix. A Figura 2 (b) representa o total de acessos à cache L3. De forma oposta ao kernel LU, o Linux obteve $17,06\%$ menos acessos à cache que no Nanvix, o que pode ser devido a diferenças no gerenciador de memória e no escalonador de processos. A Figura 2 (c) e a Figura 2 (d) apresentam as instruções de desvios condicionais total e desvios condicionais não previstos, respectivamente. De forma oposta ao que se obteve no kernel LU, tanto em (c) quanto em (d), observa-se um comportamento muito similar em ambos os sistemas, o que indica que o binário final gerado pelo compilador foi muito próximo entre os sistemas. A Figura 2 (e) e a Figura 2 (f) apresentam a quantidade total de ciclos com e sem variações de clock. O Linux apresenta quantidade de ciclos muito superior ao Nanvix, cerca de $158,5\%$ e o gráfico assume o mesmo comportamento que em quantidade de instruções (a), o que indica uma correlação

entre a quantidade de instruções e ciclos, o que não ocorre no kernel LU. Do mesmo modo que no kernel LU, não houveram ganhos de desempenho para as variações de quantum em 50, 100 e 200 ciclos o que também indica que a carga de trabalho não é sensível para essa configuração.

6. Conclusão

A principal contribuição deste artigo consiste na análise dos sistemas mencionados anteriormente e nas alterações no escalador de processos do Nanvix, que permitiram o desenvolvimento de um módulo para acesso aos contadores de hardware, bem como variações de *quantum* para o mesmo. Os resultados revelaram que o Nanvix alcança um desempenho de até $158,5\%$ superior ao Linux, para cargas de trabalho com acesso regular à memória, e que as variações de *quantum* propostas não influenciaram no comportamento dos sistemas operacionais estudados.

O sistema operacional Nanvix está sendo portado para arquiteturas multicore, por meio de uma abordagem mestre-escravo, onde um processador é definido como mestre e é responsável por todo o tratamento de interrupções e chamadas de sistema. Os processadores escravos, por sua vez, apenas executam processos em espaço de usuário. Em seguida, o Nanvix será adaptado para arquiteturas manycores e será novamente realizado um estudo comparativo com o mesmo.

Agradecimentos

Os autores agradecem a CAPES, CNPq e FAPEMIG pelo suporte parcial no trabalho.

Referências

Beckman, P., Iskra, K., Yoshii, K., Coghlan, S., and Nataraj, A. (2008). Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11(1):3–16.

Chen, J. B., Endo, Y., Chan, K., Mazieres, D., Dias, A., Seltzer, M., and Smith, M. D. (1995). The measured performance of personal computer operating systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 299–313, New York, NY, USA. ACM.

Ferreira, K. B., Bridges, P. G., Brightwell, R., and Pedretti, K. T. (2013). The impact of system design parameters on application noise sensitivity. *Cluster Computing*, 16(1):117–129.

Love, R. (2010). *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition.

Mucci, P. J., Browne, S., Deane, C., and Ho, G. (1999). Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10.

Penna, P. H. (2017). The Nanvix Operating System. Research report, Pontifical Catholic University of Minas Gerais (PUC Minas).

Tanenbaum, A. S. and Woodhull, A. S. (2005). *Operating Systems Design and Implementation (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Uso do Método Multi Frontal para Acelerar uma Aplicação de Ablação por Radiofrequência

Marcelo Cogo 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. *RAFEM is an application that calculate the voltage used in a Radiofrequency Ablation procedure. Currently RAFEM uses the Frontal Method for assembly and to solve the linear system equations generated for each treatment case. The method is optimized to reduce memory usage, but it demand more processing time for achieve the results. This work aims to reduce the execution time of the application. For that, the Multi Frontal Method implemented in the MUMPS library was adopted, which allows to processing the solution in parallel. In this way, it was possible to reduce the total computing time of the program by up to 14 times.*

Resumo. *RAFEM é uma aplicação que calcula a tensão usada em um procedimento de Ablação por Radiofrequência. Atualmente o programa usa o Método Frontal para a montagem e solução dos sistemas de equações lineares gerados para cada caso de tratamento. O método é otimizado para reduzir o uso de memória, mas o tempo de processamento dos resultados é alto. Este trabalho tem como objetivo reduzir o tempo de espera pelos resultados. Para tanto, adotou-se o Método Multi Frontal, implementado na biblioteca MUMPS, que permite extrair as vantagens do processamento da solução de forma paralela. Desta forma, foi possível reduzir o tempo total de computação do programa em até 14 vezes.*

1. Introdução

A simulação computacional tem uma grande importância em várias áreas de pesquisa científica. Trata-se de uma metodologia flexível e de baixo custo para a modelagem e solução dos mais diversos problemas [Asanovic and et al 2006]. Usando a simulação computacional é possível, através da compreensão e interpretação dos resultados obtidos, identificar melhorias para um determinado procedimento.

A Ablação por Radiofrequência - RFA - hepática [Jiang et al. 2010] é um tipo de tratamento de saúde que usa a simulação computacional para calcular os valores de temperatura e tensão usados no procedimento médico. A RFA consiste na utilização de um eletrodo especial, que é posicionado no tumor do fígado, na maioria das vezes de forma percutânea, guiado por ultrassonografia, tomografia computadorizada ou ressonância magnética [Possebon 2016]. Um gerador de corrente de radiofrequência é ligado ao eletrodo, que após o seu posicionamento, passa a receber energia por poucos minutos, gerando calor no local estipulado. Desta forma as células cancerígenas são destruídas e posteriormente substituídas por tecido cicatricial naturalmente.

O software *Radiofrequency Ablation Finite Element Method* - RAFEM - possibilita realizar os cálculos necessários para um tratamento do tipo RFA. A aplicação usa atualmente a decomposição LU e o Método Frontal para chegar aos valores numéricos de temperatura e tensão. O Método Frontal é frequentemente usado na análise de elementos finitos, como uma variação do método de eliminação de Gauss. Este método resolve de forma exata e sequencial um sistema linear, fato que gera um elevado custo computacional para a simulação. Além disso, obter a solução de grandes sistemas de equações é uma tarefa que envolvem muitos cálculos computacionais, levando mais de 80% do tempo total em algumas simulações [Camarda and Stadtherr 1998].

O objetivo deste trabalho é reduzir o tempo envolvido na execução de RAFEM. Para tanto, buscou-se adotar um outro método de solução que permita explorar o paralelismo para as atuais arquiteturas computacionais. Para tanto, o Método Multi Frontal foi utilizado [Duff and Reid 1983]. Este opera sobre matrizes densas, explora a localidade dos dados, garantindo desempenho com vetorização, abertura dos laços e paralelismo.

2. RAFEM

RAFEM é uma aplicação desenvolvida em linguagem de programação C++, que tem como objetivo simular o procedimento de RFA, calculando os valores aproximados para a distribuição de temperatura em um fígado durante o procedimento [Jiang et al. 2010]. A aplicação modela o domínio através do uso do Método de Elementos Finitos - FEM. Desta forma, um domínio, neste caso a distribuição de calor em um fígado, é discretizado, dividindo-o em um número finito de partes menores denominadas elementos. Os elementos possuem uma forma geométrica e comportamento bem definidos. No RAFEM são utilizados elementos tetraedrais conforme a Figura 1. Estes elementos são conectados entre si por nós, formando uma malha como retrata a Figura 2.

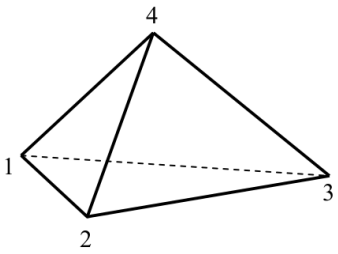


Figura 1. Um elemento tetraedral [Jiang et al. 2010].

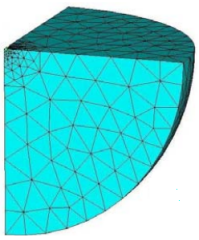


Figura 2. Malha de elementos tetraedrais [Jiang et al. 2010].

O código de RAFEM inicia com uma malha de elementos finitos passada como entrada para o programa. Após, uma série de pré-processamentos é realizada e, posteriormente, prossegue-se para a etapa iterativa, onde a maior parte do tempo de processamento ocorre, conforme identificado no trabalho de [Kapelinski et al. 2016]. Na etapa iterativa é feita a combinação de todos os elementos representando o domínio. A partir disto, é possível montar um sistema de equações lineares e encontrar a solução de cada um dos nós/incógnitas. Esse sistema de equações é chamado de matriz global de rigidez, que é uma matriz simétrica e tende a ser esparsa, como é possível observar na Figura 3.

Para resolver o sistema de equações podem ser usadas diversas abordagens. Atualmente RAFEM utiliza-se da decomposição LU e do Método Frontal. Neste trabalho

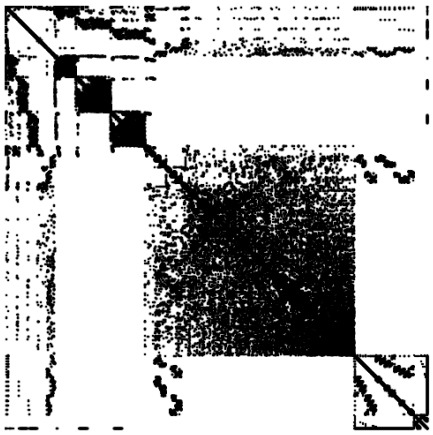


Figura 3. Matriz global de rigidez para uma malha composta por 8364 nós e 444811 elementos. Pontos em preto significam valores diferentes de zero e correspondem a cerca de 16% das entradas da matriz.

deseja-se avaliar o método de solução Multi Frontal.

2.1. Método Frontal

O Método Frontal ou *Frontal Solver* foi proposto inicialmente por [Irons 1970] no início da década de 70. O método foi desenvolvido para ser utilizado na análise dos elementos finitos com o propósito de montar e solucionar os sistemas de equações provenientes de aplicações que usavam o FEM. Naquela época os computadores tinham menos memória RAM, então para problemas muito grandes não era possível armazenar todo o sistema resultante da contribuição de todos elementos do problema em memória.

Conforme [Scott 2003] descreve em seu trabalho, a principal ideia do Método Frontal é representar a matriz A como uma soma de todas as matrizes elementais correspondentes a cada elemento presente no problema, agrupando um elemento por vez em uma matriz menor e densa chamada de matriz frontal. Assim que uma variável se torna totalmente somada (ou seja, não está envolvida em nenhuma das matrizes elementais ainda a serem montadas), ela se torna um candidato a eliminação. Desta forma, é possível intercalar as operações de montagem e eliminação. Quando ocorre uma eliminação, linhas e colunas da matriz frontal são escritas em disco, reduzindo o uso de memória RAM.

2.2. Método Multifrontal

O Método Multi Frontal é um método direto para a solução de sistemas de equações lineares esparsas desenvolvido como uma versão paralela do Método Frontal. O objetivo do método é acelerar a execução, sem se preocupar com o consumo de memória como o Método Frontal. A principal característica desta abordagem é o uso de uma árvore de montagem ou *assembly tree* ilustrada na Figura 4, onde os nós dessa árvore são compostos por submatrizes do problema, as matrizes frontais. A abordagem Multi Frontal explora o paralelismo de três diferentes formas, definidas em [Amestoy et al. 2000] como paralelismo na árvore, nos nós e na raiz da árvore. Estes são nomeados como paralelismo do tipo 1, 2 e 3, respectivamente, conforme identificados na Figura 4.

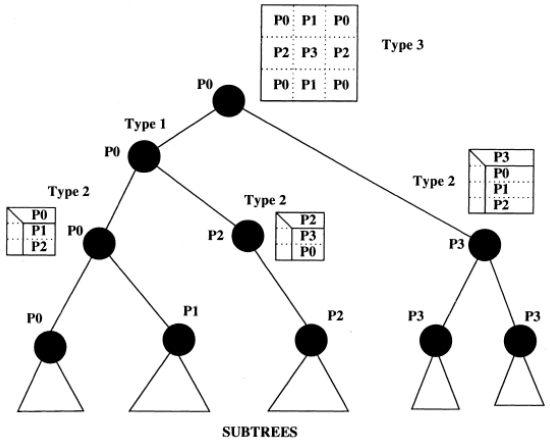


Figura 4. Distribuição das computações da árvore de montagem Multi Frontal [Amestoy et al. 2000].

Neste trabalho foram feitas pesquisas buscando encontrar implementações já existentes do Método Multi Frontal. Entre as opções disponíveis, MUMPS é uma das bibliotecas mais utilizadas e atualizadas que provê uma implementação do método escolhido. A biblioteca MUMPS implementa o Método Multi Frontal usando outras APIs e bibliotecas como o OpenMPI para a troca de mensagens e *Scotch* [Chevalier and Pellegrini 2008] e *Metis* [Karypis and Kumar 1995] para a ordenação da matriz do sistema. Para obter a solução final do sistema, a matriz de entrada passa por três principais etapas durante a execução do código análise, fatoração e solução.

Na etapa de análise é feito o pré-processamento da matriz do sistema. A matriz é passada para a biblioteca através de três vetores, um contendo os valores diferentes de zero da matriz, e os demais para armazenar a linha e coluna daquele valor. A etapa busca reduzir o custo computacional de se realizar as etapas de fatoração e solução, realizando uma série de ordenamentos, sempre mantendo a esparsidade e a simetria caso a matriz seja simétrica. É nesta etapa que é gerada a árvore de montagem.

A etapa de fatoração usa a matriz pré-processada para realizar a decomposição LU. Primeiro distribui-se a matriz nos processadores de acordo com o a ordem expressa pela árvore de montagem. Em seguida é feito o processamento das matrizes frontais, simultaneamente, partindo dos nós folha da árvore até a raiz. Após a fatoração, as matrizes resultantes são armazenadas em memória ou em disco para serem usadas na fase de solução. Por fim, o sistema linear esparso inicial **Ax = B**, após a etapa de fatoração, é escrito na forma **A = LU**, que pode ser representado na forma **LUx = B**. Assim para a etapa de solução do sistema são usadas as substituições para frente e para trás.

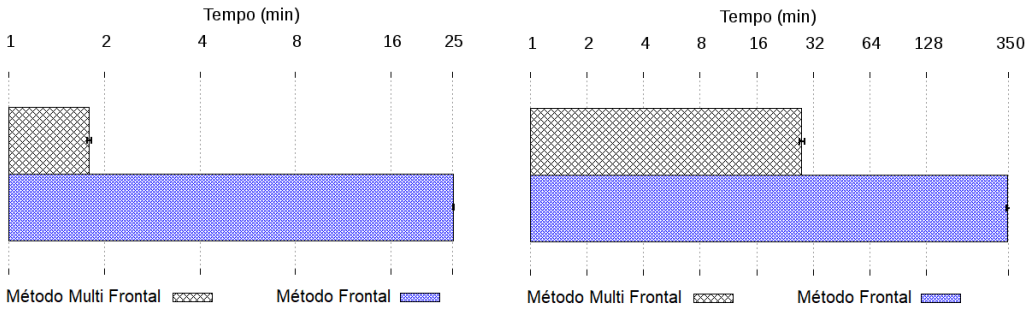
3. Análise Experimental

A fim de avaliar os benefícios do Método Multi Frontal no desempenho de RAFEM, foram feitos experimentos avaliando os tempos de execução da aplicação. Primeiramente, RAFEM foi executado considerando o Método Frontal para a resolução dos sistemas lineares. Posteriormente, adotou-se o Método Multi Frontal. Em ambos os casos os dados numéricos resultantes são compatíveis. Os testes foram conduzidos em um

notebook Dell Inspiron 3442, composto por um processador quad-core i5-4210U de 1.7 GHz e 4 GB de memória RAM.

Para validar os experimentos, foram considerados duas resoluções de malha. A primeira malha contém 3.548 nós e 18.363 elementos e a segunda possui 8.436 nós e 444.418 elementos. O número de elementos é maior do que o número de nós porque um nó pode ser compartilhado entre vários elementos vizinhos. As malhas foram criadas a partir de modelos de fígado reais, onde o domínio foi discretizado em uma estrutura tetraedral usando o software Ansys e armazenadas em arquivos. O arquivo resultante fornece os valores de temperatura e tensão inicial e os valores das posições dos nós da malha, mapeando a relação de cada nó com os seus elementos vizinhos.

Os testes foram compostos por 25 execuções de RAFEM considerando a primeira malha e 15 execuções para a segunda malha. Na Figura 5 são apresentados os valores médios para cada um dos casos. A média dos tempos de execução obtidos para o Método Frontal foram de 25,08 minutos para a primeira entrada e 348,09 minutos para a segunda, já para a abordagem Multi Frontal os tempos foram de 1,79 e 27,79 minutos.



(a) Primeira malha (b) Segunda malha

Figura 5. Média e desvio padrão dos tempos de execução do RAFEM.

A partir da análise dos resultados foi possível observar uma boa redução no tempo de execução do código usando o Método Multi Frontal. O novo método reduziu o tempo gasto na parte de solução do sistema linear, o qual é a etapa do programa que é executada iterativamente, sendo chamada várias vezes. A execução desta etapa é realizada cerca de 280 vezes para o caso de teste menor e cerca de 980 vezes para o teste maior. Em média, usando-se o Método Frontal, o tempo de execução de cada chamada era de aproximadamente 5 e 21 segundos, respectivamente para cada malha. Ao usar o Método Multi Frontal o tempo gasto por chamada da etapa foi reduzido para 0,8 e 2,76 segundos.

4. Conclusão e Trabalhos Futuros

Atualmente o método RAFEM demanda uma quantidade significativa de tempo de processamento de parâmetros para o procedimento de Ablação por Radiofrequência. Este trabalho colabora na investigação de como acelerar a aplicação. Para tanto, foi feito uma avaliação do desempenho do Método Multi Frontal. Os resultados mostraram que o método gerou um ganho de 12 e 14 vezes em relação ao Método Frontal, anteriormente usado no RAFEM. Além disso, os resultados numéricos são compatíveis.

Este trabalho não investigou detalhadamente o paralelismo que a biblioteca MUMPS oferece, pois ela permite fazer uso de processos MPI para executar o Método Multi Frontal, além do paralelismo em laços de OpenMP. Assim, como trabalhos futuros pretende-se analisar a execução do código de RAFEM usando o Método Multi Frontal em outros ambientes computacionais como em ambientes de memória distribuída, ou utilizando coprocessadores. Utilizar outras bibliotecas com a mesma finalidade e comparar os resultados obtidos também é uma das atividades previstas. Por fim, foi identificada a possibilidade de se paralelizar o laço de montagem do sistema linear usando diretivas OpenMP, uma vez que esta etapa passa a se tornar mais custosa conforme menos tempo é gasto para a solução do sistema.

Agradecimentos

Este trabalho foi desenvolvido com recursos da agência de fomento CNPq: Edital Universal - Processo N. 457684/2014-3 e com uma bolsa PIBIC durante o ano de 2017.

Referências

Amestoy, P. R., Duff, I. S., and L'excellent, J.-Y. (2000). Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer methods in applied mechanics and engineering*, 184(2):501–520.

Asanovic, K. and et al (2006). The landscape of parallel computing research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Berkeley, CA.

Camarda, K. V. and Stadtherr, M. A. (1998). Frontal solvers for process engineering: local row ordering strategies. *Computers & chemical engineering*, 22(3):333–341.

Chevalier, C. and Pellegrini, F. (2008). Pt-scotch: A tool for efficient parallel graph ordering. *Parallel computing*, 34(6):318–331.

Duff, I. S. and Reid, J. K. (1983). The multifrontal solution of indefinite sparse symmetric linear. *ACM Transactions on Mathematical Software (TOMS)*, 9(3):302–325.

Irons, B. M. (1970). A frontal solution program for finite element analysis. *International Journal for Numerical Methods in Engineering*, 2(1):5–32.

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.

Kapelinski, K., Schepke, C., and Serpa, M. S. (2016). Uma abordagem inicial para a paralelização de uma aplicação de simulação de ablação por radiofrequência para o tratamento de câncer. *Anais do WSCAD-WIC 2016*, page 37.

Karypis, G. and Kumar, V. (1995). Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0.

Possebon, R. B. (2016). Investigação de relações entre temperatura e impedância elétrica e permeabilidade de biomaterial (fígado) usadas para ablação de tumor por radiofrequência. Master's thesis, Mestrado em ENGENHARIA - UNIPAMPA.

Scott, J. A. (2003). Parallel frontal solvers for large sparse linear systems. *ACM Transactions on Mathematical Software (TOMS)*, 29(4):395–417.