Anais

WSCAD 2017

XVIII Simpósio em Sistemas Computacionais de Alto Desempenho

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

Organização e Edição
César Augusto Fonticelha De Rose
Márcio Bastos Castro

ISSN 2358-6613

Promoção

Co-Sponsor

Organização

Patrocínio Diamante

Patrocínio Ouro

Patrocínio Prata

Patrocínio Bronze

Patrocínio Básico

Agências de Fomento

FICHA CATALOGRÁFICA

XVIII Simpósio em Sistemas Computacionais de Alto Desempenho
(17-20 outubro 2017: Campinas – SP, Brasil)

Anais / Organizadores:
César Augusto Fonticielha De Rose, Márcio Castro.

Campinas, SP: SBC, 2017 324f. : 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. César Augusto Fonticielha De Rose. IV. Márcio Castro.

WSCAD 2017

XVIII Simpósio em Sistemas Computacionais de Alto Desempenho
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 foi composta por 8 sessões técnicas com 26 apresentações orais de artigos completos, os quais foram publicados digitalmente na Biblioteca Digital Brasileira de Computação (BDBCOMP). Os autores dos melhores artigos aceitos na trilha principal do WSCAD serão convidados a submeter uma versão estendida dos mesmos para uma edição especial do periódico *Concurrency and Computation: Practice and Experience (CCPE)* da editora Wiley (ISSN 1532-0634).

Índice

Mensagem do Coordenador Geral

Em nome do Comitê Organizador, nós damos as boas vindas à Campinas. A organização de um evento deste tamanho é sempre um trabalho de equipe. Gostaria de agradecer aos coordenadores do Comitê de Programa, César Augusto Fonticielha De Rose (PUCRS) e Márcio Castro (UFSC), e revisores pelo trabalho de selecionar uma excelente combinação de artigos. Um grande agradecimento também é devido à organização local, em nome de Alexandro Baldassin, Emilio Francesquini, Guido Araujo, Hermes Senger, Laércio Pilla, Lucas Wanner, Maurício Palma, Sandro Rigo, and Tiago Falcão.

Além disto, este evento não poderia acontecer sem o apoio de patrocinadores Capes, CNPq, FAPESP, SDC, Dell EMC, IBM, ER Soluções, Cray, NVidia, NewRoute, Google e Versatus. Também agradecemos a todo o suporte da Sociedade Brasileira de Computação (SBC) e IEEE Computer Society.

Rodolfo Azevedo (UNICAMP)
Coordenador Geral do SBAC-PAD 2017

Mensagem dos Coordenadores de Programa

É com imensa satisfação que apresentamos o programa do *XVIII Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*. O evento ocorreu entre os dias 17 e 20 de outubro de 2017 em Campinas, São Paulo.

Nesta edição contamos com 81 submissões de artigos científicos realizados por estudantes e pesquisadores de mais de 40 instituições de ensino e pesquisa nacionais, demonstrando assim, um claro fortalecimento da pesquisa na área de Computação de Alto Desempenho no Brasil. Todos os trabalhos submetidos ao WSCAD foram avaliados por um comitê formado por 64 pesquisadores doutores com renomada experiência em ensino e pesquisa. Cada artigo recebeu no mínimo 3 avaliações de diferentes revisores. Após a análise do teor das revisões, o comitê organizador optou pelo aceite de 26 trabalhos.

Gostaríamos de agradecer a todos que, de alguma forma, contribuíram para que o evento se tornasse uma realidade. Primeiramente, aos jovens pesquisadores de Iniciação Científica e de Pós-Graduação, assim como aos seus orientadores, os quais dedicaram-se a escrita e submissão de seus artigos ao WSCAD 2017. Também, a todos os pesquisadores que participaram das sessões técnicas do WSCAD 2017, apresentando os seus trabalhos, fazendo perguntas e trocando experiências com outros colegas. Além disso, a todos os membros do Comitê de Programa que abraçaram a ideia do evento e fizeram suas contribuições com intuito de aperfeiçoar os trabalhos submetidos ao evento. Por fim, mas não menos importante, a todos os envolvidos com a organização do SBAC-PAD 2017 por todo o apoio e suporte prestado ao WSCAD 2017.

Um grande abraço,

César Augusto Fonticielha De Rose (PUCRS)
Márcio Castro (UFSC)
Coordenadores de Programa do WSCAD 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

- Abel Guilhermino da Silva Filho (UFPEL)
- Adenauer Yamin (UFPEL)
- Alba Melo (UnB)
- Aletéia de Araújo (UnB)
- Alexandre Sena (UERJ)
- Alexandro Baldassin (UNESP)
- Alfredo Goldman (USP)
- Aline Nascimento (UFF)
- Alvaro Coutinho (UFRJ)
- Alvaro Fazenda (UNIFESP)
- Bruno Schulze (LNCC)
- Calebe Bianchini (Mackenzie)
- Carlos Augusto Martins (PUC Minas)
- César Augusto Fonticielha De Rose (PUCRS)
- Claudio Amorim (UFRJ)
- Claudio Geyer (UFRGS)
- Cristiana Bentes (UERJ)
- Cristiano Costa (Unisinos)
- Cristina Boeres (UFF)
- Daniel Cordeiro (USP)
- Daniel de Oliveira (UFF)
- Denise Stringhini (UNIFESP)
- Diego Dutra (UFRJ)
- Douglas Macedo (UFSC)
- Edna Barros (UFPE)
- Edson Borin (UNICAMP)
- Edson Cáceres (UFMS)
- Edward Moreno (UFS)
- Felipe França (UFRJ)
- Gabriel P. Silva (UFRJ)
- George Teodoro (UnB)
- Gerson Geraldo H. Cavalheiro (UFPEL)
- Henrique Cota de Freitas (PUC Minas)
- Hermes Senger (UFSCar)
- Igor Machado Coelho (UERJ)
- Jairo Panetta (ITA)
- João Vicente Ferreira Lima (UFSM)
- Laércio Pilla (UFSC)
- Leandro Marzulo (UERJ)
- Liria Sato (USP)
- Luis Fabrício Góes (PUC Minas)
- Lucas Schnorr (UFRGS)
- Lucia Catabriga (UFES)
- Lucia Drummond (UFF)
- Luiza Mourelle (UERJ)
- Márcio Castro (UFSC)
- Marco Netto (IBM Research)
- Maria Clicia Castro (UERJ)
- Mario Antonio Ribeiro Dantas (UFSC)
- Philippe Olivier Alexandre Navaux (UFRGS)
- Raphael Camargo (UFABC)
- Renato Ishii (UFMS)
- Ricardo Ferreira (UFV)
- Ricardo Santos (UFMS)
- Roberto Hexsel (UFPR)

- Rodolfo Azevedo (UNICAMP)
- Rodrigo Righi (Unisinos)
- Rosiane de Freitas (UFAM)
- Siang Song (USP)
- Tiago Alves (UFRJ)
- Tiago Ferreto (PUCRS)
- Vinod Rebello (UFF)
- Wagner Meira (UFMG)
- Walfredo Cirne (Google)

Revisores Convidados

- Alexandre Lima Santana (UFSC)
- Alexandre Nery (UERJ)
- Carlos Henrique Nicodemus (UFF)
- Dalvan Griebler (PUCRS)
- Edson Luiz Padoin (UNIJUI)
- Emilio Francesquini (UNICAMP)
- Gabriel Paillard (UFC)
- Igor Monteiro Moraes (UFF)
- Kassiano Matteussi (PUCRS)
- Lucas Morais (USP)
- Luiz Branco (UFRJ)
- Maicon Melo Alves (UFF)
- Marcos Amaris (USP)
- Pedro Bruel (USP)
- Rafael da Silva (UFRJ)
- Rafaelli de Carvalho Coutinho (CEFET/RJ)
- Regina Toledo (UFF)
- Rogério Gonçalves (UTFPR)
- Tiago Cariolano (UFRJ)

Sessões Técnicas

Sessão I – Ferramentas
Terça-feira, 17/10, 16:30-18:00

Sessão II – Manycore
Quarta-feira, 18/10, 09:00-10:30

Sessão III – Aplicações
Quarta-feira, 18/10, 13:30-15:00

Sessão IV – Avaliação de Desempenho
Quarta-feira, 18/10, 15:30-17:30

Sessão V – Otimização
Quinta-feira, 19/10, 08:00-09:30

Sessão VI – Escalonamento
Quinta-feira, 19/10, 10:00-12:00

ADD - Uma Ferramenta de Projeto de Aceleradores com DataFlow para Alto Desempenho*

Jeronimo Penha[1], Lucas Bragança[1], Danilo Almeida[1], Jose Nacif[1], Ricardo Ferreira[1]

[1]Departamento de Informática – Universidade Federal Viçosa (UFV)
CEP: 36.570-900 – Viçosa – Minas Gerais – Brazil

{jeronimopenha,danilooalmeida94}@gmail.com

{lucas.braganca, jnacif, ricardo}@ufv.br

Resumo. *Aceleradores com FPGA baseados em fluxo de dados se tornaram uma alternativa promissora para se conseguir alto desempenho com eficiência energética. Este artigo apresenta a ferramenta ADD (Accelerator Design and Deploy) para descrição de algoritmos com fluxo de dados, que também possibilita a simulação, a prototipação em FPGA e a integração em um ambiente heterogêneo CPU-FPGA. A ferramenta possui uma biblioteca de operadores síncronos, além de possibilitar o desenvolvimento de novos operadores. Oferece suporte para acoplamento com linguagens de programação de alto nível e foi validada na plataforma para computação heterogênea CPU-FPGA de alto desempenho da Intel/Altera. Como resultado, obteve-se ganhos no tempo de processamento dos benchmarks de até seis vezes em relação às execuções single thread, o que mostra a eficiência da ferramenta proposta.*

1. Introdução

Um dos grandes desafios da computação é obter alto desempenho com eficiência energética. Neste cenário, aceleradores como GPUs e FPGAs tem alto desempenho ao explorar o paralelismo, e ao mesmo tempo, baixo consumo de energia em relação aos processadores de uso geral. Ademais, aplicações mapeadas em FPGA na forma de grafo de fluxo de dados (*data flow graphs* - DFG) reduzem o consumo de energia, pois não é necessário fazer a busca e a decodificação das instruções a cada ciclo de relógio. Entretanto, os FPGAs têm como maior barreira a dificuldade para modelagem, programação e compilação de aplicações como mostrado em [Stitt 2011].

Uma alternativa é o uso de OpenCL (*Open Computing Language*) [Munshi 2009]. A Intel/Altera e a Xilinx (os dois maiores fabricantes) disponibilizaram novas versões de compiladores com C/C++ e OpenCL para mapeamento direto e eficiente em FPGA [OpenCL , IntelFPGA , Xilinx]. A vantagem do OpenCL é o fato de ser difundido na comunidade de alto desempenho, ter a linguagem C/C++ como base e o usuário precisaria aprender apenas as primitivas para expressar o paralelismo. Entretanto, se faz necessário conhecimento do processo de mapeamento do código no fluxo de dados e das arquiteturas de FPGA para fazer ajustes, compreender as informações geradas pelas ferramentas e otimizar o código.

Outra alternativa é o OpenSPL [Kim et al. 2010] que é baseado em Java. O ambiente OpenSPL permite a visualização dos grafos de fluxo de dados gerados para código

*Financiamento: FAPEMIG, CAPES e CNPq

e possui primitivas para trabalhar com os fluxos. Assim como em OpenCL, o OpenSPL mapeia o código parte para processador, parte para o FPGAs de modo automático, assim como o acoplamento entre as duas plataformas. Em ambas abordagens, o código é transformado de forma implícita em um *Data-Flow Graph* (DFG) pelo compilador, portanto o programador não tem total controle sobre o DFG gerado e ao mesmo tempo precisa saber codificar e observar o DFG gerado para fazer ajustes.

Este artigo propõe a ferramenta ADD (*Accelerator Design and Deploy*) para o desenvolvimento de algoritmos por meio de grafos de fluxo de dados a serem mapeados em FPGA que possuem uma interface com códigos escritos em linguagens de alto nível (*Accelerator Design*). Provê facilidades de mapeamento e execução em FPGA (*Deploy*) o que reduz a curva de aprendizado e desenvolvimento. O objetivo é auxiliar a modelagem algoritmos e operadores com o intuito de familiarizar os projetistas com os conceitos de fluxo de dados. Como contribuição, a ferramenta permite realizar a simulação em nível de DFG, com geração automática de código para execução em FPGAs acoplados a processadores. O projeto gerado inclui também uma interface de acoplamento com as plataformas de software para que possam fazer chamadas e transferir dados para o acelerador. O ADD complementa as abordagens de OpenCL e OpenSPL, porém o DFG deve ser explicitamente descrito.

Este trabalho se divide em cinco seções. Na Seção 2, a ferramenta de desenvolvimento e implantação de aceleradores ADD é apresentada. Nesta Seção, as características de funcionamento e utilização da ferramenta são descritas, tais como: a construção de DFGs, suporte à criação de algoritmos, operadores contidos na biblioteca, desenvolvimento de novos operadores, a geração de *Verilog*, simulação e execução em FPGA. A apresentação dos resultados dos experimentos com a execução dos algoritmos: *Gourand*, *FIR8*, *FIR16*, *Histogram*, *Paeth* e *Reduce SUM*; projetados no **ADD** e executados no ambiente de alto desempenho híbrido **CPU-FPGA**, da fabricante **Intel/Altera**, são apresentados e discutidos na Seção 3. A Seção 4 aborda trabalhos relacionados e, por fim, as considerações finais são apresentadas na Seção 5.

2. Ferramenta ADD

A ferramenta ADD permite a criação, simulação e validação de DFGs em uma interface gráfica que possibilita a edição e alteração dos algoritmos a serem construídos. A simulação foi implementada como extensão do editor/simulador HADES [Hendrich 2000] através da criação de uma biblioteca com operadores descritos em Java. O usuário pode também desenvolver novos operadores e adicionar a biblioteca. A ferramenta automaticamente converte o grafo criado para a linguagem de descrição de hardware, no caso *Verilog*, para que seja sintetizada em FPGAs usando ferramentas da Xilinx ou Intel/Altera.

As linguagens de descrição de *hardware* como VHDL e *Verilog* HDL são complexas porque a sua utilização exige um conhecimento sobre *hardware* e seu funcionamento. Isto traz barreiras aos profissionais de desenvolvimento de *software* [Stitt 2011]. Para contornar essa dificuldade as linguagens OpenCL/OpenSPL inserem palavras reservadas (*pragmas* e/ou estruturas de dados) em linguagens de alto nível que possibilitam a criação de algoritmos que serão mapeados em *hardware* de modo integrado às linguagens C/Java. O código gerado é sintetizado com ferramentas comerciais de FPGA, como

Quartus da Intel/Altera e depois executados na plataforma CPU-FPGA. Estes ambientes permitem apenas visualizar o grafo de fluxo de dados gerado mas sem a possibilidade de edição [Ling et al. 2017, Winans 2015].

Complementando o OpenCL e OpenSPL, o ADD exercita o desenvolvimento de algoritmos na forma explícita de DFG que são automaticamente mapeados em FPGA, além de permitir a criação de novos operadores que amplia as possibilidades de descrição de novos DFGs. O processo envolve dois passos. O primeiro é a especificação do DFG onde é possível simular, realizar depuração e analisar o desempenho no nível de DFG. Posteriormente, um gerador mapeia o DFG em código *Verilog*, que posteriormente pode ser sintetizado para um FPGA, semelhante ao fluxo de projeto de OpenCL/OpenSPL. O segundo passo é a execução do código em uma plataforma heterogênea CPU-FPGA.

O ADD oferece suporte às linguagens Java e C/C++ para comunicação entre seu código na transferência de dados e acionamento do acelerador que é encapsulado por uma chamada de função. A interface de comunicação foi desenvolvida para duas plataformas de FPGA. A primeira de baixo custo com fins educacionais, usa a interface JTAG para a comunicação computador-FPGA e pode ser usada em diversos kits da fabricante Altera conectados a computadores comuns (*notebook* ou *desktop*), permitindo acesso a muitos programadores e estudantes. A segunda plataforma é de alto desempenho, desenvolvida pela Intel, que combina um processador Xeon e um FPGA acoplados à memória compartilhada por meio do barramento QPI [Gupta 2016, Gupta 2015].

2.1. Operadores

Para descrição dos DFG, a biblioteca de operadores é organizada nas seguintes categorias: acumuladores, aritméticos, *branches*, comparadores, I/O, lógicos, registrador, memória e *shift*. Os operadores foram baseados em uma equivalência com instruções RISC. A lista de operadores pode ser observada na Tabela 1. Na tabela pode-se ver os nomes dos operadores separados por categorias. Os operadores que possuem a letra "I"no fim do nome são operadores que trabalham com valores imediatos. Além dos operadores de uso geral, foram implementados alguns operadores específicos para exemplificar novas possibilidades para uso de DFGs. Por exemplo, o operador *Histogram* que possui uma memória interna para a construção de um histograma e os operadores acumuladores.

Diferente de uma GPU, o DFG permite a execução simultânea de diversos caminhos divergentes em pipeline. Para a lógica de controle de fluxo foram propostos inicialmente os operadores *Branch* e *Merge*. As possibilidades de execução dos desvios geram fluxos independentes e o operador *Merge* é o responsável por selecionar qual deles será repassado aos próximos operadores. Os operadores podem ser configurados para se ajustar ao algoritmo. Estes ajustes podem ser, por exemplo, a quantidade de bits de uma porta de entrada ou saída.

A Figura 1 apresenta um exemplo com controle de fluxo e um código ilustrativo (a). Nela pode-se observar como modelar os *Branches*. O problema possui três possibilidades diferentes para a computação do fluxo de saída *Y*. Neste exemplo, as três computações são executadas e ao fim, apenas uma é selecionada. Observe que as condições e as computações são executadas em paralelo. As variáveis *I* representam os valores imediatos de cada operador.

O grafo apresentado na Figura 1 pode ser replicado várias vezes, para explorar

Tabela 1. Lista de operadores disponíveis na biblioteca do ADD.

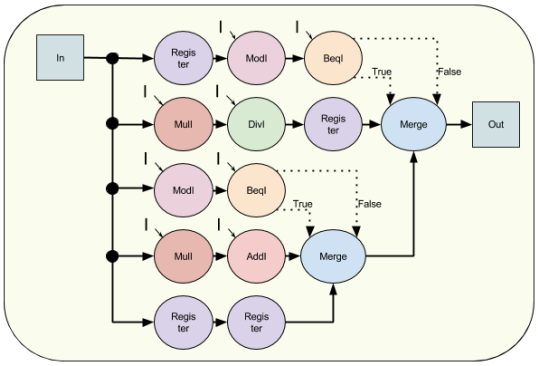
Categoria	Operador	Função
Aritméticos	Abs	$y = \|a\|$
	Add	$y = a + b$
	Div	$y = a/b$
	Mod	$y = a\%b$
	Mul	$y = a * b$
	Sub	$y = a - b$
	AddI	$y = a + I$
	DivI	$y = a/I$
	ModI	$y = a\%I$
	MulI	$y = a * I$
	SubI	$y = a - I$
Lógicos	And	$y = a\&b$
	Or	$y = a\|b$
	Not	$y = \ a$
	AndI	$y = a\&I$
	OrI	$y = a\|I$
I/O	In 1-32	-
	Out 1-32	-
Memória	Histogram	-
Registrador	Register	$y = a$

Tipo	Operador	Função
Controladores de Fluxo	Beq	$if = (a == b) : 1; 0$ $else = (a == b) : 0; 1$
	Bne	$if = (a! = b) : 1; 0$ $else = (a! = b) : 0; 1$
	BeqI	$if = (a == I) : 1; 0$ $else = (a == I) : 0; 1$
	BneI	$if = (a! = I) : 1; 0$ $else = (a! = I) : 0; 1$
	Merge	$y = a$ se $if == 1$ $y = b$ se $else == 1$
Shift	Shl	$y = a << b$
	Shr	$y = a >> b$
	ShlI	$y = a << I$
	ShrI	$y = a >> I$
Comparadores	Max	$y = Max(a, b)$
	Min	$y = Min(a, b)$
	Slt	$y = (a < b)?1 : 0$
	MaxI	$y = Max(a, I)$
	MinI	$y = Min(a, I)$
	SltI	$y = (a < I)?1 : 0$
Acumuladores	AccAdd	-
	AccMax	-
	AccMin	-
	AccMul	-

```
...
foreach x in  stream_In
  if(x%2 == 0){
    y = (x*2)/1;
  }else if(x%3 == 1){
    y = (x*3)+2;
  }else {
    y = x;
  }

}
...
```

(a)



(b)

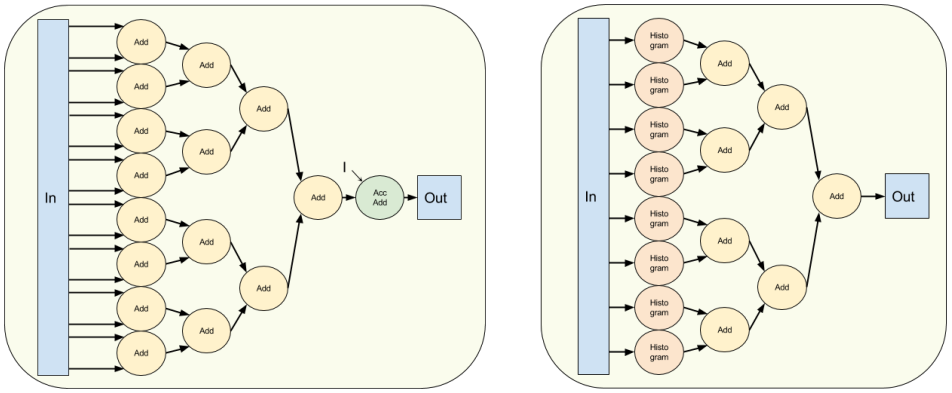
Figura 1. Exemplo de algoritmo com *Branches* (a) e o DFG equivalente esquemático com os operadores da biblioteca do ADD (b).

o paralelismo espacial. A versão atual do ADD traz a possibilidade da utilização de operadores de entrada de dados ("*In*") com até trinta e duas saídas, o que permite que trinta e duas cópias sejam executadas em paralelo. É possível alterar estes operadores e criar novos com mais recursos, como será mostrado na Seção 2.2. Outra opção para o exemplo anterior seria calcular a condição e usá-la para alimentar apenas um dos fluxos que irá executar a computação correta através do uso de novos operadores.

Outro exemplo de operador específico é a operação de redução com o auxílio dos

operadores *acumuladores*. O acumulador processa e armazena o resultado temporário. Após todos os elementos do vetor serem processados, o resultado da redução é entregue para o estágio posterior. Na Figura 2(a), um exemplo de redução de dezesseis elementos por vez. Diferente de uma GPU que pode ter baixa ocupação no final da redução, a abordagem com DFG alocou, neste exemplo, 15 operadores e todos são usados ao mesmo tempo em *pipeline*.

Outro exemplo é a modelagem de histogramas onde o operador possui uma memória interna e contadores. A execução é realizada em paralelo seguida de uma redução. A quantidade de dados a serem lidos pelo operador pode ser configurada a cada execução da mesma forma que a configuração dos operadores que trabalham com imediatos e dos operadores acumuladores. Um exemplo de histograma com oito operadores trabalhando em paralelo com redução é apresentado na Figura 2 (b) seguido de uma redução.



(a) (b)

Figura 2. Algoritmos *Reduce Add* (a) e *Histogram* (b).

2.2. Desenvolvimento de Novos Operadores

Novos operadores podem ser criados para darem suporte à implementação de mais algoritmos, modelos e ferramentas de projetos com novas linguagens e compiladores. A biblioteca oferece operadores genéricos e parametrizáveis que foram criados para servirem como uma base para novos operadores. Na versão atual, os operadores base possuem as características: número de portas de entrada (1 ou 2), acúmulo de valores, controle de fluxo (*Branches*) com e sem uso de valores imediatos, controle de entrada de dados, controle de dados de saída de dados. Através da herança na linguagem Java, os novos operadores podem ser rapidamente codificados e testados para simulação. Para execução no FPGA, a descrição *Verilog* do novo operador deve ser adicionada ao gerador do ADD.

A Figura 3 apresenta o código que implementa o operador *MulI*, incluso na biblioteca do ADD. Este operador executa a multiplicação do valor do barramento de entrada de dados por uma constante (imediato) e herda as características básicas do operador genérico *GenericI*.

2.3. Geração de *Verilog*

O gerador do ADD permite ao programador abstrair do código *Verilog* e da complexidade de todos os detalhes de sincronismo no nível de circuito, que envolvem máquinas de esta-

```
package add.dataflow.sync;
public class MulI extends GenericI {
    public MulI() {
        super();
        setCompName("MULI");
    }
    @Override
    public int compute(int data) {
        setString(Integer.toString(id), Integer.toString(immediate));
        return (int) (data * immediate);
    }
}
```

Figura 3. Código que o implementa o operador MulI.

dos, sinais de relógio, *reset*, habilitação, codificação, etc. A descrição *Verilog* sintetizável é gerada com o auxílio da biblioteca Veriloggen [Takamaeda-Yamazaki]. Semelhante ao fluxo de projeto em OpenCL da Intel, o código gerado pelo ADD deve ser sintetizado com uma ferramenta de projeto de FPGA como o Quartus.

2.4. Simulação, Comunicação e Execução no FPGAs

A comunicação com o acelerador é feita através de filas de entrada e saída de dados, seja no nível de simulação e/ou execução. O uso das filas de entrada e saída desacopla o DFG da plataforma FPGA que irá implementar o sistema. Duas interfaces foram implementadas, uma fracamente e outra fortemente acopladas. A primeira usa a interface JTAG para a comunicação com os FPGAs e pode ser usadas em kits para fins didáticos e de depuração. A segunda desenvolvida com o uso da API AAL/QPI da Intel/Altera para acoplamento CPU-FPGA por memória compartilhada.

O ADD traz consigo uma API com métodos que permitem a simulação e a execução do circuito desenvolvido. O simulador HADES [Hendrich 2000] foi escolhido como base para o desenvolvimento da ferramenta ADD por ser portátil e por possuir flexibilidade para a criação de novos operadores, várias extensões do HADES já foram propostas [Penha et al. 2016, Ferreira et al. 2015, Ferreira et al. 2005, Ferreira et al. 2004, Marwedel et al. 2002], porém outros simuladores podem ser utilizados futuramente.

A execução com o FPGA se dá de forma semelhante à execução no simulador. O aplicativo a ser executado deverá utilizar a API do ADD para iniciar a transferência de dados para o FPGA e após a execução repassar os dados processados de volta à chamada da função. O método responsável por esta execução é bloqueante, isto é, a instrução seguinte a chamada será executada apenas ao término do processamento do DFG. Para que o aplicativo que fará uso do acelerador não fique bloqueado, a chamada para a API deve ser executada em uma *thread* independente, assim o aplicativo ficará livre para executar outras tarefas durante a execução do DFG.

Para a execução dos aceleradores em FPGAs, é necessária a conversão do arquivo estrutural criado pelo ADD para uma linguagem de descrição de hardware e uma interface de acoplamento entre o sistema e o FPGA. O ADD traz consigo uma implementação com JTAG capaz de executar DFGs em kits de FPGA. Detalhes sobre o funcionamento da interface disponível podem ser vistos na Figura 4. A transferência de dados se dá por

meio do JTAG e, internamente, os circuitos de interface repassam e recebem os resultados do DFG. O ADD e juntamente com a interface JTAG podem ser obtidos em: `https://github.com/ComputerArchitectureUFV/ufv_add.git`.

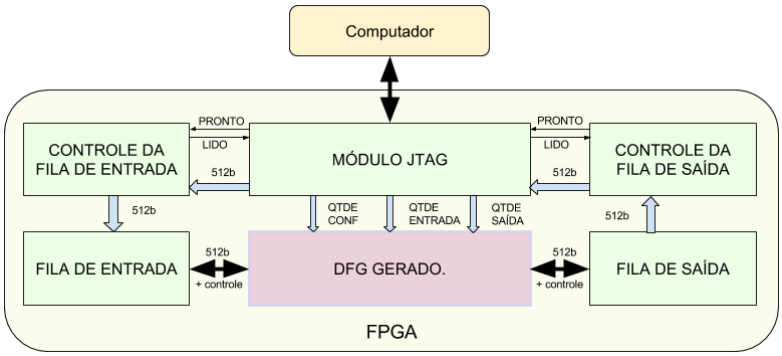


Figura 4. Interface para execução em FPGAs.

O ADD pode suportar novas plataformas, como o Catapult da Microsoft [Putnam et al. 2014], com a adição de pequenas modificações em sua API. Além da interface JTAG, o ADD possui suporte para a execução na plataforma híbrida CPU-FPGA da fabricante Intel/Altera. A Figura 5 detalha o processo de execução dos aceleradores com as duas interfaces: JTAG e XEON/FPGA com QPI.

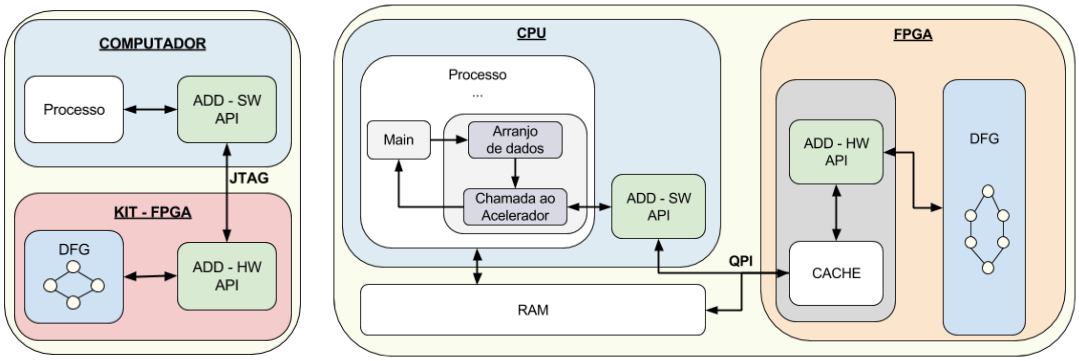


Figura 5. (a) Execução de um DFG em kit de FPGA; (b) Ambiente de alto desempenho da fabricante Intel/Altera

2.5. Protocolo de Comunicação das Interfaces de Entrada e Saída de Dados

A comunicação entre os operadores de controle de dados e as FIFOs de entrada e saída segue um protocolo de comunicação simples e funcional que proporciona um desacoplamento entre o FPGA e o processo solicitante. Para a entrada de dados, a fila de entrada informa ao circuito a existência de dados a serem processados. O controlador de entrada de dados do DFG então realiza leitura dos mesmos e os repassa para processamento. Caso seja necessária a configuração de operadores que trabalham com imediatos, as configurações são executadas antes do início do processamento. Este processo de controle é transparente para a fila de entrada. Sempre que houver a falta de dados para leitura,

devido a atrasos na transferência, o processamento é interrompido e reiniciado até que hajam dados disponíveis.

O retorno dos dados processados segue um protocolo semelhante ao de entrada. Caso a fila de saída fique cheia, o controlador de saída paralisa o processamento momentaneamente até que a fila de saída esteja novamente disponível. O programador não precisa se preocupar com o controle de entrada e saída que é implementado pela API.

3. Experimentos e Resultados

Para a validação do funcionamento e da eficiência dos DFGs desenvolvidos usando o ADD, foi usado um conjunto de *benchmarks* de algoritmos para processamento de sinais que podem ser vistos na Tabela 2. Os recursos necessários para a gravação no FPGA STRATIX V da plataforma de alto desempenho da fabricante Intel/Altera são mostrados na tabela nos campos Elementos Lógicos (*ALMs*), Registradores (*Reg*), módulos de memória embarcadas *M20K* e módulos *DSPs*. Os *benchmarks* foram replicados respeitando as restrições de número de entrada e saída de cada DFG e limitados ao máximo de trinta e duas portas de 16 bits. Esta limitação é devido à restrições do barramento implementado pela Intel/Altera que trabalha com pacotes de 64 bytes, onde os testes foram realizados. Na coluna "*Benchmark*"pode-se ver entre parênteses o número de cópias paralelas que foram sintetizadas para cada DFG.

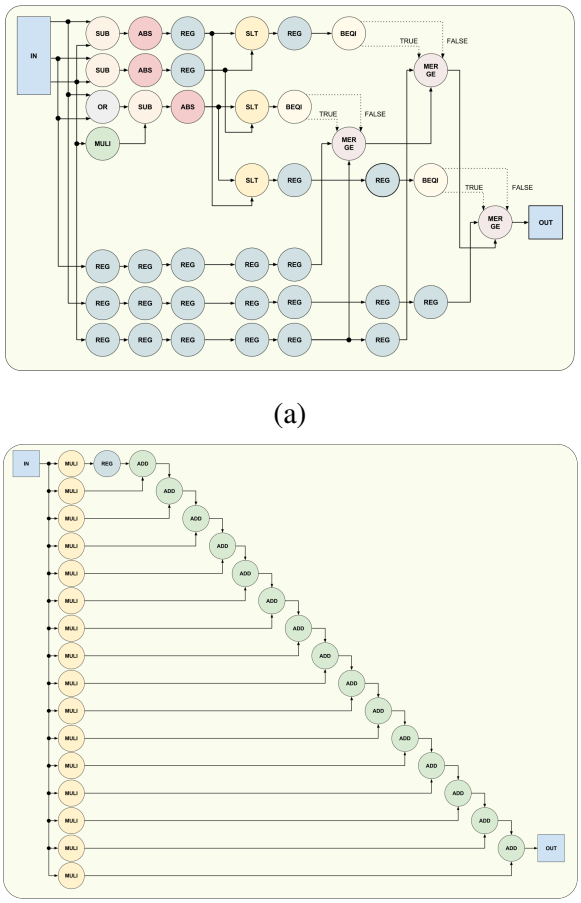
Tabela 2. Relação de resultados de execução dos algoritmos em CPU e FPGA.

Benchmark	Stratix V 5SGXEA7N1F45C1					CPU	T(cpu)/T(fpga)
	ALMs (%)	Registers	M20K	DSP (%)	T(ms)	T(ms)	
Gourand (8x)	35	82.352	168	0	168	358	2,130
FIR 8 (32x)	37	89.726	168	100	229	553	2,414
FIR 16 (32x)	44	104.111	168	100	207	1.394	6,734
Histogram (32x)	98	233.206	168	0	135	165	1,223
Paeth (8x)	35	84.226	168	3	170	335	1,970
Reduce SUM (32x)	35	78.988	168	0	135	77	0,570

O sistema CPU-FPGA usa *FPGA Stratix V 5SGXEA7N1F45C1* fortemente acoplado a um processador XEON com 10-*cores*. Esta plataforma permite o acesso à memória do computador através do barramento QPI que realiza a transferência em pacotes de 64 *Bytes* a uma taxa de transferência de aproximadamente 6 GBps [Choi et al. 2016]. A plataforma traz implementado um sistema de requisições de dados com cache que é responsável pela solicitação e gravação de dados na memória RAM. A API-HW do ADD trabalha em conjunto com esta estrutura para que o acelerador possa ser acessado e utilizado. Os circuitos foram sintetizados através da IDE Quartus II 64-Bit Version 13.1.0 e os recursos utilizados para a implementação de cada algoritmo foi retirado pelo relatório de síntese do mesmo.

Com intuito de demonstrar a eficiência dos DFGs criados, foi feita a comparação com a execução dos mesmos algoritmos no processador Intel(R) Xeon(R) CPU E5-2680 v2 com frequência de *clock* de 2.80GHz. Para a implementação foi utilizada a linguagem C e a compilação e otimização foi feita pelo compilador GCC 4.8.4. Os tempos de execução, levando em consideração o tempo de transferência de dados, em milissegundos, para cada algoritmo foram medidos e podem ser observados na Tabela 2. A intenção

da comparação é demonstrar que os DFGs criados através da ferramenta são eficientes. Como ilustração, os DFGs para uma instância dos algoritmos FIR16 e Paeth podem ser vistos na Figura 6.



(a)

(b)

Figura 6. DFGs para os algoritmos Paeth (A) e FIR 16 (B).

Para a execução dos *benchmarks* foram utilizados 512MB de dados em palavras de 16 *bits* tanto para o acelerador quando para o processador. Os resultados exibidos na Tabela 2 mostram que os tempos de execução para o acelerador são competitivos como os apresentados pela execução em CPU de forma sequencial. O melhor resultado foi observado na execução do algoritmo FIR16, replicado 32 vezes, que foi seis vezes mais rápido do que a execução do mesmo algoritmo na CPU. É importante levar em consideração que o mesmo utilizou 100% dos DSPs disponíveis no FPGA, o que aumenta consideravelmente a resposta deste algoritmo. As instruções básicas do mesmo são multiplicação e soma. Em relação a execução para o Reduce SUM, o tempo medido foi superior ao da CPU pelo fato de este realizar a soma de apenas 32 elementos em paralelo e a operação executada ser simples.

4. Trabalhos Relacionados

Um DFG expressa explicitamente o paralelismo do código, o que possibilita a execução de partes de algoritmos em elementos processadores independentes em diversas plataformas. Trabalhos recentes exploram vários aspectos dos DFGs para as plataformas heterogêneas com FPGAs, GPUs e arquiteturas *multi-core* e *many-cores*. Uma abordagem de otimização de DFG baseada em transformações foi apresentada em [Stewart et al. 2017]. Para um algoritmo de processamento de vídeo, os resultados experimentais mostram uma melhoria de aproximadamente 50% na frequência do FPGA que superaram as otimizações das ferramentas comerciais de FPGA.

Uma abordagem para vetorização de código para GPUs a partir de DFGs com atores foi apresentada [Barford et al. 2014], onde um DFG de um filtro e um DFG com máquinas de estados foram avaliados. Considerando arquiteturas *many-cores* com estruturas de conexão baseadas em *network-on-chip*, a abordagem apresentada em [Ul-Abdin and Yang 2017], mostra, para um estudo de caso, que a partir de um DFG, um ganho de aceleração de até 4 vezes pode ser obtido na arquitetura *many-core* com 10 núcleos em comparação com um processador embarcado.

Um outro exemplo de otimização baseada na modelagem de problemas com DFG para sistemas embarcados foi apresentado recentemente em [Palumbo et al. 2017] onde o objetivo é reduzir a potência dissipada e a arquitetura alvo utilizada foi um arranjo reconfigurável CGRAs (*Coarse-Grained Reconfigurable Arrays*). Em comparação com ASICs de 45nm, os circuitos desenvolvidos conseguiram uma redução de mais de 70% no consumo de energia estático e mais de 90% no consumo dinâmico.

O trabalho proposto tem como finalidade a criação visual de DFGs e a possibilidade de simulação e execução em vários ambientes. Diferentemente dos trabalhos relacionados, o foco é no desenvolvimento e depuração de DFGs, enquanto que nos trabalhos relacionados o foco é mais concentrado na otimização de DFGs já desenvolvidos. Não foram encontrados outros trabalhos atuais com proposta semelhante a deste para que uma melhor comparação pudesse ser feita.

5. Considerações finais

Este trabalho apresenta a ferramenta ADD para uso de grafos de fluxo de dados para a criação de projetos com a possibilidade de simulação e testes de algoritmos em FGPAs. O ADD é mais uma opção de ambiente de desenvolvimento que traz uma forma alternativa de projetar DFGs em um ambiente flexível que pode ser expandido com novos operadores. É acessível por possibilitar seu uso em kits didáticos de FPGAs, além de possibilitar a execução em plataformas heterogêneas de alto desempenho. Um conjunto de algoritmos foi descrito na forma de DFG e o ADD fez a transformação automática para mapeamento em FPGA.

Os exemplos foram avaliados na plataforma de alto desempenho da Intel composta por um processador XEON fortemente acoplado através da memória a um FPGA. A API do ADD abstraiu a API de software e hardware da Intel permitindo o uso da plataforma de forma transparente a partir do DFG inicial. A interface de comunicação gerada pelo ADD provê um sistema de filas que promove o desacoplamento entre o DFG e a plataforma alvo o que permite também a comunicação através da interface JTAG para validação dos DFGs em FPGAs de baixo custo.

Para trabalhos futuros, pretende-se aprimorar a ferramenta para que dê suporte a mais ambientes com FPGA já consolidados, híbridos ou não, acoplamento com compiladores e linguagens de domínio específico, outras ferramentas de geração ou transformação de código e otimização dos DFGs desenvolvidos na ferramenta. Outro aspecto é o mapeamento para arquiteturas de grão grosso como os CGRAs (*Coarse-Grained Reconfigurable Architecture*), que podem ser implementados como *overlays* em FPGAS ou diretamente em silício.

Referências

Barford, L., Bhattacharyya, S. S., and Liu, Y. (2014). Data flow algorithms for processors with vector extensions: handling actors with internal state. In *Signal and Information Processing (GlobalSIP), 2014 IEEE Global Conference on*, pages 20–24. IEEE.

Choi, Y.-k., Cong, J., Fang, Z., Hao, Y., Reinman, G., and Wei, P. (2016). A quantitative analysis on microarchitectures of modern cpu-fpga platforms. In *Design Automation Conference (DAC)*. ACM/EEE.

Ferreira, R., Cardoso, J. M., and Neto, H. C. (2004). An environment for exploring data-driven architectures. In *Int. Conference on Field-Programmable Logic and Applications (FPL)*.

Ferreira, R., Cardoso, J. M., Toledo, A., and Neto, H. C. (2005). Data-driven regular reconfigurable arrays: design space exploration and mapping. In *Int. Conf. on Embedded Computer Systems Architectures, Modeling and Simulation SAMOS*.

Ferreira, R., Nacif, J., Magalhaes, S., de Almeida, T., and Pacifico, R. (2015). Be a simulator developer and go beyond in computing engineering. In *Frontiers in Education Conference (FIE)*. IEEE.

Gupta, P. (2016). Accelerating datacenter workloads. In *26th International Conference on Field Programmable Logic and Applications*.

Gupta, P. K. (2015). Xeon+ fpga platform for the data center. In *Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, volume 119.

Hendrich, N. (2000). A java-based framework for simulation and teaching: Hades—the hamburg design system. In *Microelectronics Education*, pages 285–288. Springer.

IntelFPGA. Intel FPGA - Acceleratiing The Smart and Connected World. `https://www.altera.com/`. Accessed: 2017-08-09.

Kim, E., Kim, K., and In, H. P. (2010). A multi-view api impact analysis for open spl platform. In *Advanced Communication Technology (ICACT), 2010 The 12th International Conference on*, volume 1, pages 686–691. IEEE.

Ling, A. C., Aydonat, U., O'Connell, S., Capalija, D., and Chiu, G. R. (2017). Creating high performance applications with intel's fpga opencl™ sdk. In *Proceedings of the 5th International Workshop on OpenCL*, page 11. ACM.

Marwedel, P., Cong, K., and Schwenk, S. (2002). Ravi: Interactive visualization of information system dynamics using a java-based schematic editor and simulator.

Munshi, A. (2009). The opencl specification. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pages 1–314. IEEE.

OpenCL. Opencl™ zone – Accelerate Your Applications. http://developer.amd.com/tools-and-sdks/opencl-zone/. Accessed: 2017-08-09.

Palumbo, F., Fanni, T., Sau, C., and Meloni, P. (2017). Power-awarness in coarse-grained reconfigurable multi-functional architectures: a dataflow based strategy. *Journal of Signal Processing Systems*, 87(1):81–106.

Penha, J. C., Fontes, G., and Ferreira, R. (2016). MIPSFPGA - Um simulador mips incremental com validação em fpga. *International Journal in Computer Architecture Education (IJCAE)*, 5(1):19–25.

Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., et al. (2014). A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE.

Stewart, R., Bhowmik, D., Wallace, A., and Michaelson, G. (2017). Profile guided dataflow transformation for fpgas and cpus. *Journal of Signal Processing Systems*, 87(1):3–20.

Stitt, G. (2011). Are field-programmable gate arrays ready for the mainstream? *IEEE Micro*, 31(6):58–63.

Takamaeda-Yamazaki, S. Veriloggen: A library for constructing a verilog hdl source code in python. https://github.com/PyHDI/veriloggen. Accessed: 2017-07-20.

Ul-Abdin, Z. and Yang, M. (2017). A radar signal processing case study for dataflow programming of manycores. *Journal of Signal Processing Systems*, 87(1):49–62.

Winans, J. (2015). On dataflow computing with openspl.

Xilinx. Xilinx – All Programmable. https://www.xilinx.com/. Accessed: 2017-08-09.

High-Level and Efficient Stream Parallelism on Multi-core Systems with SPar for Data Compression Applications

Dalvan Griebler[1], Renato B. Hoffmann[1], Junior Loff[1], Marco Danelutto[2], Luiz Gustavo Fernandes[1]

[1] Faculty of Informatics (FACIN), Pontifical Catholic University of Rio Grande do Sul (PUCRS), GMAP Research Group, Porto Alegre, Brazil.

[2]Department of Computer Science, University of Pisa (UNIPI), Pisa, Italy.

{dalvan.griebler, renato.hoffmann, junior.loff}@acad.pucrs.br, marcod@di.unipi.it, luiz.fernandes@pucrs.br

Abstract. *The stream processing domain is present in several real-world applications that are running on multi-core systems. In this paper, we focus on data compression applications that are an important sub-set of this domain. Our main goal is to assess the programmability and efficiency of domain-specific language called SPar. It was specially designed for expressing stream parallelism and it promises higher-level parallelism abstractions without significant performance losses. Therefore, we parallelized Lzip and Bzip2 compressors with SPar and compared with state-of-the-art frameworks. The results revealed that SPar is able to efficiently exploit stream parallelism as well as provide suitable abstractions with less code intrusion and code re-factoring.*

1. Introduction

Over the past decade, vendors realized that increasing clock frequency to gain performance was no longer possible. Companies were then forced to slow the clock frequency and start adding multiple processors to their chips. Since that, software started to rely on parallel programming to increase performance [Sutter 2005]. However, exploiting parallelism in such multi-core architectures is a challenging task that is still too low-level and complex for application programmers. Consequently, parallel programming has been reserved for specialized people. On the other hand, application programmers are more concerned with algorithmic strategies and application constraints instead of supporting multi-core parallelism in their applications [Griebler et al. 2017a].

Data compression applications are one of the most important ways to save storage space. In fact, twenty years ago it was already reported that streaming applications were the most computing intensive applications [Rixner et al. 1998]. They are related to a wide set of application, such as video, big data, deep-learning, and among others. Introduce parallelism in these applications is necessary for obtaining high-performance on current multi-core systems. Therefore, combined with the performance needs and high-level parallelism abstraction requirement, new frameworks were proposed to reduce programming effort. However, it is still a challenge to balance abstraction with performance. Most of the languages are inflexible to model modern stream applications, complex for programmers, which has lead them to deal with low-level hardware optimization, or are inefficient to implement real applications [Benkner et al. 2012, Beard et al. 2015].

The so considered state-of-the-art runtimes and parallel programming interfaces for expressing the stream parallelism are Thread Building Blocks (TBB) [Reinders 2007]

and FastFlow (FF) [Aldinucci et al. 2014]. They aim to abstract parallelism through parallel patterns with building blocks. Among a set of patterns provided, they also support the parallelism implementation on streaming applications. In addition, an emergent DSL (Domain-Specific Language) named SPar that promise to provide suitable and higher-level abstractions for expressing stream parallelism [Griebler et al. 2017a]. Our goals are to assess the programmability and performance of SPar on multi-core systems for real-world data compression applications. Therefore, we extended studies of the Bzip2 parallelization that were focused only on productivity analysis of SPar in our previous work [Griebler et al. 2017b] and we parallelized Lzip for this paper with SPar, TBB, and FastFlow. Thus, our main contributions are the following:

- The code parallelization of Lzip with SPar, FastFlow, and TBB.
- A comparative analysis of two important performance metrics (memory usage and completion times) and programmability (qualitative and quantitative) of SPar, TBB, FastFlow and POSIX-Threads for Bzip2 and Lzip.

In this paper, Section 2 describes the related works. In Section 3, we introduce SPar DSL. Section 4 discusses two real-world loss-less data compressors (Lzip and Bzip2), considering programming and specific aspects related to the high-level stream parallelism. Then, we detail the experiments evaluating programmability and performance in Section 5. To finalize, our conclusions will be presented in Section 6.

2. Related Work

As we focus on data compression applications, we will discuss related works whose frameworks/libraries are able to exploit stream parallelism on multi-core systems and C++ programs as well as previous works that parallelized Lzip and Bzip2 compressors. For parallel programming, FastFlow [Aldinucci et al. 2014], RaftLib [Beard et al. 2015], and TBB [Reinders 2007] are the ones. FastFlow is a framework created in 2009 by researchers at the University of Pisa and the University of Turin in Italy. It provides stream parallel abstractions adopting an algorithmic skeleton perspective and its implementation is on top of efficient fine grain lock-free communication mechanisms. We mainly used FastFlow as the target of our SPar parallelism implementation because it provides ready to use parallel patterns with high-level C++ templates for stream parallelism.

Another available tool is TBB (Threading Building Blocks), an Intel C++ library for general purpose parallel programming. It emphasizes scalable and data parallel programming while completely abstracting the concept of threads by providing a concept of task. TBB builds on C++ templates to offer common parallel patterns (map, scan, parallel_for, and among others) implemented on top of a work-stealing scheduler [Reinders 2007]. More recently, RaftLib [Beard et al. 2015] is a C++ library designed to support pipeline and data parallelism together. The idea is that the programmer implements sequential code portions as computing kernels, where custom split/reduce can be implemented when dealing with data parallelism. Moreover, there is a global online scheduler that can use OS scheduler, round-robin, work-stealing, and cache-weighted work-stealing. Yet, the communication between kernels is performed by using lock-free queues. FastFlow and TBB, RaftLib are lower-level parallelism abstractions to the final application programmer with respect to SPar, but they are considered runtimes for SPar.

To the best of our knowledge, there are no previous works that parallelized Lzip compressor with TBB, FastFlow, or RaftLib. However, the work of [Benkner et al. 2012] implemented a parallel version of Bzip2 with TBB using a pipeline of three stages. They

compared performance and lines of code for TBB and the original POSIX-Threads version (Pbzip2). Moreover, Bzip2 was also parallelized in [Aldinucci et al. 2011] with FastFlow framework by using a `Farm` and software accelerator feature. In contrast, we provided a completest analysis and comparison of programmability and performance for Bzip2 and Lzip with SPar, TBB, and FastFlow.

3. SPar

SPar is a C++ embedded DSL designed to provide higher-level parallelism abstractions for streaming applications without significant performance losses [Griebler et al. 2017a, Griebler 2016]. SPar uses the standard C++ attribute mechanism [ISO/IEC 2014] for implementing its annotation-based language so that coding productivity is improved. To express stream parallelism with SPar, it offers five different attributes. These attributes are used as parameters of annotations that describe key features of a streaming application such as stages, data consumption and degree of parallelism. The SPar compiler will recognize a SPar annotation when at least the first attribute of a double brackets annotation is specified (`[[id-attr, aux-attr, ...]]`). This first attribute must be an identifier (ID) attribute, where a list of auxiliary (AUX) attributes may be specified if necessary. The `ToStream` and `Stage` annotations are ID while `Input`, `Output` and `Replicate` are AUX.

The SPar compiler was developed with the CINCLE (A Compiler Infrastructure for New C/C++ Language Extensions) support tools [Griebler 2016]. It generates parallel code with calls to the FastFlow library. SPar uses the Farm and Pipeline interfaces and customizes them for its particular needs. More details regarding the SPar usage may be found in [Griebler et al. 2017a, Griebler et al. 2017b]. In addition, SPar also supports other options through compiler flags that can be activated when desired (individually or combined) as follows:

- `spar_ondemand`: generates an on-demand item distribution policy by setting the queue size to one. Therefore, a new item will only be inserted in the queue when the next stage has removed the previous one.
- `spar_ordered`: makes the scheduler preserve the stream items order. FastFlow provides us a built-in function for this purpose so that SPar compiler can simply generate it.
- `spar_blocking`: switches the runtime to behave in passive mode (default is active) blocking the scheduler when the communication queues are full. FastFlow offers a pre-processing directive so that the SPar compiler may easily support it.

4. Data Compression Applications

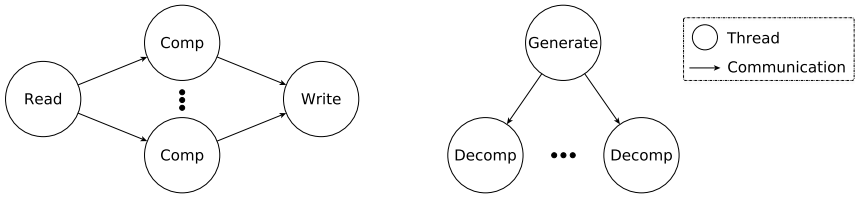
The applications used in our studies are Lzip [Diaz 2017] and Bzip2 [Seward 2017]. Although both are compressors, these applications differ from each other on the encoding/decoding algorithm and details about its parallel implementation. We will extend the discussion on both of these compressors in the following sections.

4.1. Lzip

Lzip is a C/C++ lossless data compressor based on the Lempel–Ziv–Markov chain algorithm (LZMA) used by the zip family compressors. There is a standard POSIX-Threads implementation called Plzip that we used to compare to our SPar parallelization concerning efficiency and programmability. The compression mode splits the input file into

blocks with a fixed number of bytes. Then, it will apply the LZMA algorithm to encode the blocks, subsequently reassembly them in the resulting compacted file. Similarly, the decompression mode will perform the same tasks, except that instead of encoding, the blocks will be decoded. Plzip adopts a different approach when generating the data-flow for the compression mode. Firstly, Lzip outputs a compressed file of a single large block. As multiple smaller blocks are required for parallel processing, Plzip must adapt the compression so that it operates in multiple blocks instead of one. This entails on 0.4% to 2% larger files and files compressed with Lzip can not benefit from the parallelism of the Plzip decompression.

Figure 1 illustrates the parallel activity graphs of Plzip, which can be viewed as a Pipeline and Farm parallel patterns. Figure 1(a) is arranged in the stream parallelism fashion and the computation is abstracted into three entities represented by `read`, `comp` and `write`. In the decompression mode (Figure 1(b)), the `decomp` entity performs `read`, decoding, and `write` entities. The `generate` entity, only distributes a unique data scope to perform the `decomp` to each thread. The communication between all these entities is performed through queues with the same direction as the arrows are indicating.



(a) Compression. (b) Decompression.

Figure 1. Parallel activity graph of Plzip.

Listing 1 presents the SPar annotations in the sequential version of Lzip compression mode, which will generate the activity graph presented in Figure 1(a). As we can see in line 1, the first annotation uses a `ToStream` attribute to mark the stream region and an `Input` attribute to indicate the data consumed by it. In our example, the `other_data` variable is used as a generic representation of the necessary parameters of the region. In line 4, the second annotation was used to specify the compression computation. In addition to `other_data`, this `Stage` will consume the `data` variable read in the previous `Stage`. At the same time, it will produce the compressed `data` for the subsequent stage (see `Input` and `Output`). None of the blocks have shared variables, therefore, they can be safely processed in parallel (see the use of `Replicate` on line 4). Although it is possible to fragment the compression stage into two other stages, this diminished the performance in out tests. Because most of the processing would be executed by one stage, leaving less work for the other stage. Finally, the last annotation in line 7 will consume the compressed data from the previous stage and write it to the output file. In parallel processing, the order of the queue insertion in the last stage is non-deterministic by default. As we want the output file to be equivalent to the input one, the stream must maintain its original order. In SPar, this is achieved by appending the `-spar_ordered` flag to the compilation command.

Unlike compression, the decompression function of Plzip adopts a data parallelism approach that generates the activity graph presented in Figure 1(b). This is why most of the operations computed in the decompress function involve the `file_index` class. An instantiation of this class will read the input file and split it into several blocks. Knowing

the number of available blocks previously, enables a static partitioning of the the blocks to be processed by the parallel threads. Moreover, a finer grain is used to balance the irregular block sizes between threads in the decompression mode. Also, `file_index` contains the offset needed to reorder the blocks in the writing stage. After creating the `file_index` object, the POSIX-Threads implementation will spawn the parallel threads that will decompress and write the data assigned to them.

```
1  [[spar::ToStream,spar::Input(other_data)]]
2  while(true) {
3   // Reading stage
4   [[spar::Stage,spar::Input(data,other_data),
        spar::Output(data),spar::Replicate(
        num_workers)]]{
5   // Compression stage
6   }
7   [[spar::Stage,spar::Input(data)]]{
8   // Writing stage
9   }
10 }
```

Listing 1. SPar annotations for the compression mode.

```
1  [[spar::ToStream,spar::Input(num_workers,
       other_data)]]
2  for(int worker_id = 0; worker_id <
       num_workers; ++worker_id){
3    [[spar::Stage,spar::Input(worker_id,
       num_workers,other_data),spar::Replicate(
       num_workers)]]
4    for( long i = worker_id; i < num_blocks; i
       = i+num_workers){
5    //Full Decompression of i indexed block.
6    }
7  }
```

Listing 2. SPar annotations for the decompression mode.

In Listing 2, there is a representation of the SPar annotations for the Lzip decompression mode. Although the decompression mode uses a data parallelism approach instead of stream, it is possible to introduce data parallelism with SPar. The `ToStream` is used to indicate the stream region, which in this case, simply forwards the data to the next stage that is replicated (it has the `Replicate` attribute). Then, the second annotation in line 3 is used to specify the stage that will perform the decompression computation over the assigned data. This stage will consume the `worker_id`, `num_workers` (both used to determine the data computed by the stage) and `other_data`. Following the execution flow, the available input blocks will be iterated in line 5, and each stage will process the i indexed data that is statically assigned to it. This mode does not require `-spar_ordered` because the stream order is obtained with the support of the `file_index` class as previously explained.

4.2. Bzip2

First introduced in 1996, Bzip2 [Seward 2017] is another lossless data compression application. It uses the Burrows–Wheeler transform and Huffman coding algorithms. We adopted the Pbzip2 [Gilchrist 2004] (that is developed with POSIX-Threads) in our studies to compare with our parallel implementations. Both compression and decompression modes in Pbzip2 have the same activity graph as illustrated in Figure 1(a). Therefore, this can be viewed as a pipeline with three stages, where the first stage will split the input file into independent blocks (100,000 - 900,000 bytes) that are forwarded to the parallel (de)compression stage. The remaining stage will be sequentially writing the resulting blocks to the output file.

The original Pbzip2 implementation maintains global queues protected by lock mechanisms to communicate between stages. Initially, a single thread will start to split the input file and fill the first queue with the generated blocks. Meanwhile, the spawned parallel (de)compress threads will query the queue for an available block. Then, the resulting block of the parallel threads will be inserted in the last queue. Meanwhile, the final thread will be constantly checking the queue for the next block to be written in the

resulting file. Furthermore, this last thread will reorder the blocks with the help of an auxiliary vector used to store the blocks arriving out of order.

SPar's compression and decompression mode were annotated similarly to Listing 1, which performs like the parallel activity graph in Figure 1(a). Consequently, the SPar's version will have the first stage sequential for splitting the input file into blocks. The second stage was annotated with `Replicate`, performing like a poll of threads that receive from the previous stage the data blocks to apply the (de)compression and deliver the resulting block to the to next and last stage. It will write the (de)compressed blocks to the output file. The order of the stream in SPar is guaranteed by adding the `-spar_ordered` flag in the compilation.

5. Experiments

Our experiments aim to evaluate the programmability and performance of SPar compared to FastFlow, TBB and POSIX-threads in the parallel implementations of Lzip and Bzip2. To measure the programmability we used two different metrics: (i) Cyclomatic Complexity Number (CCN) [Laird and Brennan 2006], which represents the number of linearly independent paths within a source code; and (ii) Source Lines Of Code (SLOC). In our performance evaluations, we used a 704.2 MB ISO file as the workload. The number of workers represented in the graphs does not actually represent the real number of threads spawned by the system. However, it represents the degree of parallelism. For instance, a streaming application developed with a pipeline structure will have a pool of replicated stages. This degree of parallelism plotted in all performance graphs represents this pool, which we also refer to as the number of workers. To obtain the results, we ran each version from 1 up to the max number of threads in the target machine. We executed each degree of parallelism 10 times and obtained the average execution time. Standard deviations were plotted in the graphs by using error-bars.The machine in which the tests were executed was equipped with 24GB of RAM memory and two processors Intel(R) Xeon(R) CPU E5-2620 v3 2.40GHz, with 6 cores each and support to hyper-threading, totalling 24 threads. Its operating system was Ubuntu Server 64 bits with kernel 4.4.0-59-generic. Moreover, we used PBzip2 (1.1.13), Plzip (1.6), GCC 5.4.0 with `-O3` compiler flag, TBB (4.4 20151115), and FastFlow (r13).

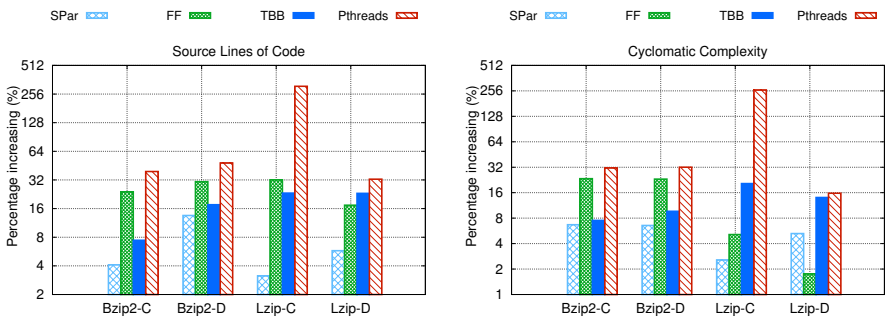
5.1. Programmability

Since we have not presented the FastFlow and TBB versions of Bzip2 and Lzip, we will briefly describe them here:

- *Lzip-TBB*: In this implementation, we used pipeline pattern, where the stages of the pipeline are abstracted into virtual functions of a TBB filter subclass. Each one of these filters is constructed with a parameter provided by the programmer. In the compression mode, this parameter was set up with `serial_in_order` in the first and last stages to maintain the original order of the stream. The middle stage was set up with `parallel` to extend the degree of parallelism. Also, we must give a number of maximum tokens that runs on-the-fly through the pipeline. We also tested a number of tokens equal to the degree of parallelism, but we opted to exclude it from our final version since it degrades the performance due to the writing stage bottleneck. The decompression mode implements a pipeline with two stages, where the second stage will perform the computation. The activity graph generated by both versions is the same as the one presented in Figure 1.

- *Lzip-FastFlow*: Although other algorithmic skeletons could be employed, we used FastFlow's *Farm* template. For this template, we must map the stages into FastFlow's *ff_node* sub-classes that represents emitter, workers and collector elements. This generates an activity graph similar to the one presented in Figure 1(a). Although FastFlow abstracts the communication between stages and thread creation, we still had to re-factoring the code and setting up the *Farm* template. One advantage of FastFlow is that it supports an optimized `Farm` template that preserves the order of the stream. Unlike the compression mode, the decompression `Farm` was developed only with the workers node and executed with an offload FastFlow function. The generated activity graph is the same as presented in Figure 1(b).
- *Bzip2-TBB and Bzip2-FastFlow*: Both TBB's and FastFlow's implementation of the compression and decompression modes produce an activity graph such as the one depicted in Figure 1(a). In FastFlow parallelization, we used a *Farm* template while for TBB, we used its *Pipeline* template with a number of tokens equal to ten times the number of workers.

We can see the code intrusion (SLOC) and complexity (CCN) of each version of Lzip and Bzip2 in Figures 2(a) and 2(b), respectively. The y axis is the percentage increase of the metric regarding the sequential code while the x axis represents the application (Bzip2 or Lzip) followed by the suffix C for compression and D for decompression. Note that all POSIX-Threads versions and both the compress and decompress of Bzip2 implemented with FastFlow were already available in [Aldinucci et al. 2011].



(a) Source Lines of Code. (b) Cyclomatic Complexity.

Figure 2. SLOC and CCN metrics increase with respect to the sequential version.

When collecting the programmability metrics, we only considered the files that effectively implement parallelism. Figure 2 shows that Bzip2 decompression mode has considerably increased the amount of source code needed for all versions. We credit this to the addition of a more complex decompression function used for the parallel versions. This change allows Bzip2's decompression mode to be parallelized since it separates the reading and decompression stage, which were executed by a single external library called in the original sequential code. The compression mode however, could be developed over the original sequential compression function, therefore, the SLOC and CCN increase is less significant. Still observing Bzip2 in Figure 2, we notice that POSIX-Threads has the highest programmability metrics when compared to the sequential code. That is because POSIX-Threads has to manually handle stream reordering, communication protocols, and threads creation. In Bzip2, FastFlow had the second worst result because it reused most of the POSIX-Threads structures, which could be further abstracted by FastFlow.

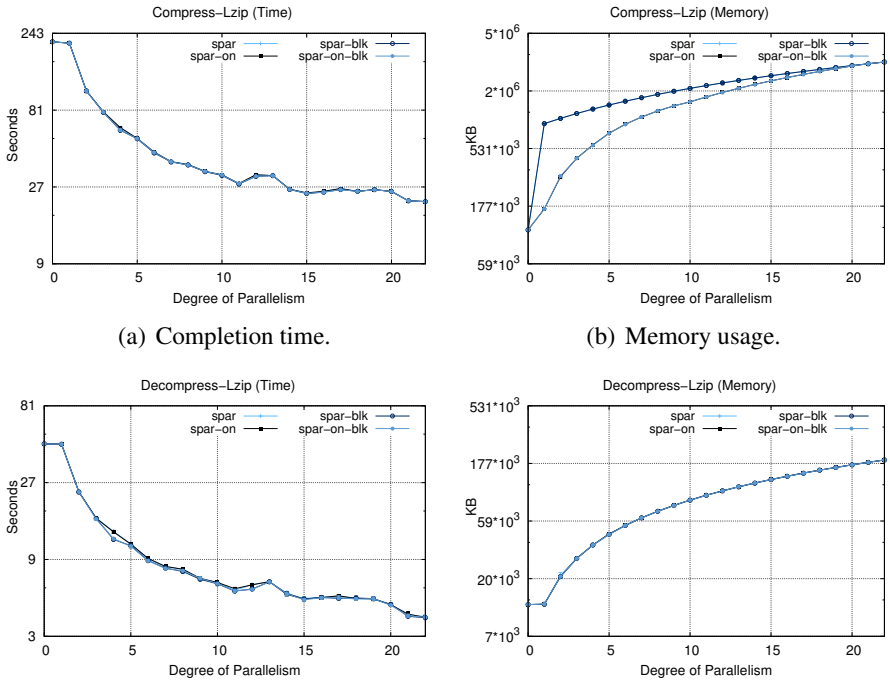
We can observe a similar configuration of SLOC and CCN increase in Figures 2(a) and 2(b) in Lzip parallel versions. Despite FastFlow presenting a higher amount of code needed for decompression compared to SPar, it also presents a lower CCN. That is because FastFlow implements more numerous, smaller, and simpler functions while SPar implements on a single big function, which impacts the CCN negatively. At the same time, FastFlow has a SLOC increase more similar to TBB, since now it takes full advantage of FastFlow's abstractions. One example of this is the abstraction of the producer pipeline stage with a software accelerator that offloads data from the main thread in the decompression function. Also, we highlight that other high-level solutions could be employed by TBB as well (use of `parallal_for` template) for the decompression function, although the need of a different syntax can make this more difficult. Finally, all decompression mode versions achieved a smaller CCN and SLOC increase due to the simpler parallelism strategy employed.

In addition, the fragmentation of the code into `read`, `comp`, and `write` stages is only obtained using parallel programming. This means that the application programmer does not have those details in mind when developing an application. Because of that, when programming FastFlow and TBB over the sequential code, even though low-level details like communication protocols and scheduling options are abstracted, we end up returning to the original POSIX-Threads structure of three separate stages, and need to re-factor the sequential code. SPar on the other hand, does not require the code to be re-shaped/re-written/re-factored in these studied applications, maintaining the sequential code structure.

5.2. Performance

We present the performance results in Figures 3 and 4 obtained for the Lzip application and in Figures 5 and 6 for the Bzip2 application. Our parallelizations were compared with the related works for the Lzip and Bzip2 POSIX-Threads version, and Bzip2 with FastFlow. We evaluated execution time, and memory usage for all parallel versions with SPar (`spar`) combining the `spar_ondemand` (`on`) and `spar_blocking` (`blk`) compiler flags. Also, we plotted the POSIX-Threads (`Plzip` and `PBzip2`), FastFlow (`ff`), and TBB (`tbb`) versions. As we can observe, the standard deviation was negligible in almost all the cases because it is not visible through error-bars. In Figure 3, we present the results obtained for all versions of SPar with the possible combinations of the optimization flags. We highlight that these optimizations modify memory consumption while having almost no impact in the total execution time of the Lzip application.

In Figures 3(b) and 3(d), however, the memory usage variation is caused by the `spar_ondemand` flag. Here, an on-demand scheduling is generated by setting the stages queue size to one, meaning that the workload will be distributed dynamically. Each thread will only receive a new item in its queue once it has already removed the previous one. This way, we reduce the concurrent number of total active items, and the memory demand. Figure 4 depicts the comparison of the best SPar version with respect to the best FastFlow, TBB, and POSIX-Threads implementations. We observe that all versions presented a similar completion time. Concerning memory usage in Figure 4(b), the FastFlow compression mode presented a slightly higher demand up to the fifteenth number of workers. That is because FastFlow does not use the on-demand scheduling, which means that more concurrent items will exist in the queues. This could be improved by enabling the on-demand scheduling within code. The memory usage of the decompression mode presented in Figure 4(d) shows that all versions achieved almost identical results, which is



(a) Completion time. (b) Memory usage.

(c) Completion time. (d) Memory usage.

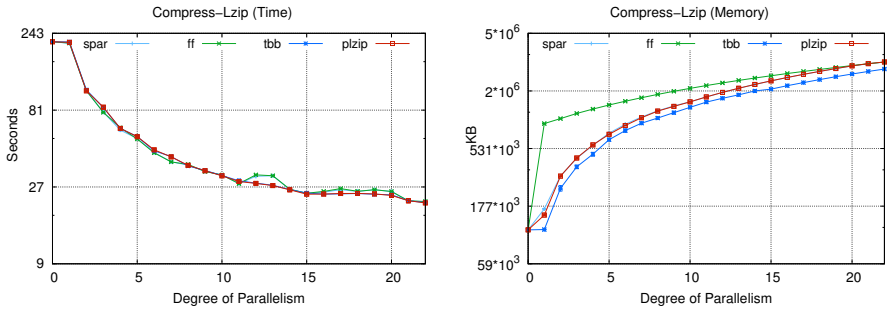
Figure 3. Lzip performance with SPar options.

expected since they all use the same class to manage file splitting and block distribution.

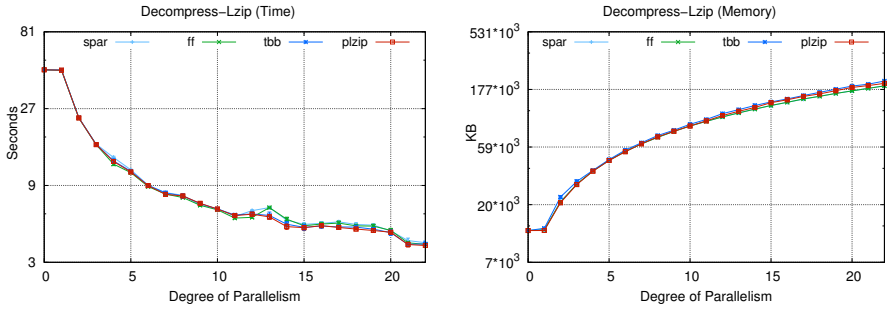
Regarding the Bzip2 application, we can visualize the results of the SPar versions in Figure 5. The completion time achieved a similar result between all SPar versions. However, for the memory consumption we observed similar behavior in Lzip with the SPar versions not using the `spar_ondemand` (on). The same justification of Lzip applies here, the versions with *on* generate less concurrent items in the queues. Even so, in Bzip2 the memory consumption is constant and the disparity between the *on* and non *on* versions is higher. That is because Bzip2 creates larger blocks than Lzip and the whole input file will be in the queues whereas on the *on* versions only part of it will be in the communication queues.

We can compare Bzip2 results in Figure 6. All parallel versions presented very similar performance between them. Our experiments also revealed a pattern of behavior between Lzip and Bzip2 relative to the performance for the parallel versions. In the end, SPar generated efficient FastFlow parallel code, sometimes slightly better than the hand tuned versions such as for the memory usage metric.

Table 1 presents the best speed-ups (***S***) and its respective degree of parallelism need to achieve (Size) for the parallel implementations of Lzip and Bzip2. As expected, SPar presented a slightly lower performance (6% in the worst case) compared to the original Pthreads implementation. Though SPar generates FastFlow code, there small differences between these two versions for both applications. That is because the FastFlow and SPar versions are not necessarily the same. FastFlow was hand-coded whereas SPar generates the code automatically. This indicates that the code generation overhead is negligible and SPar is able to efficiently abstract stream parallelism for Lzip and Bzip2.

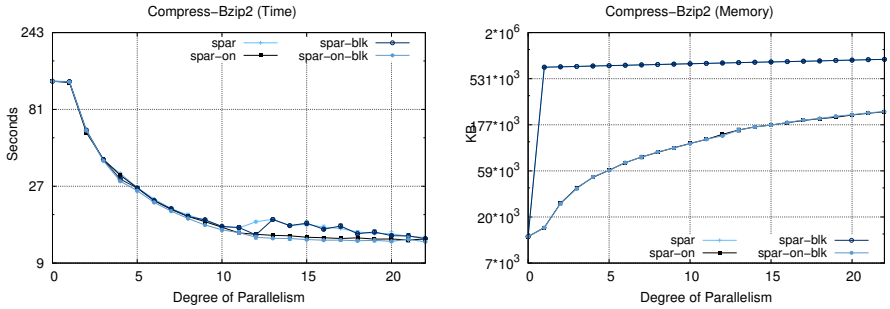


(a) Completion time. (b) Memory usage.

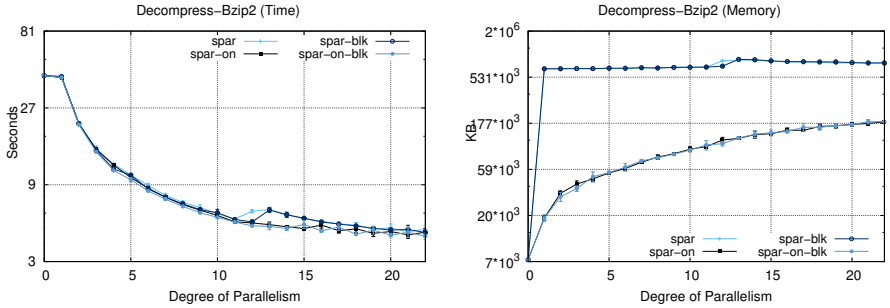


(c) Completion time. (d) Memory usage.

Figure 4. Lzip performance comparison.

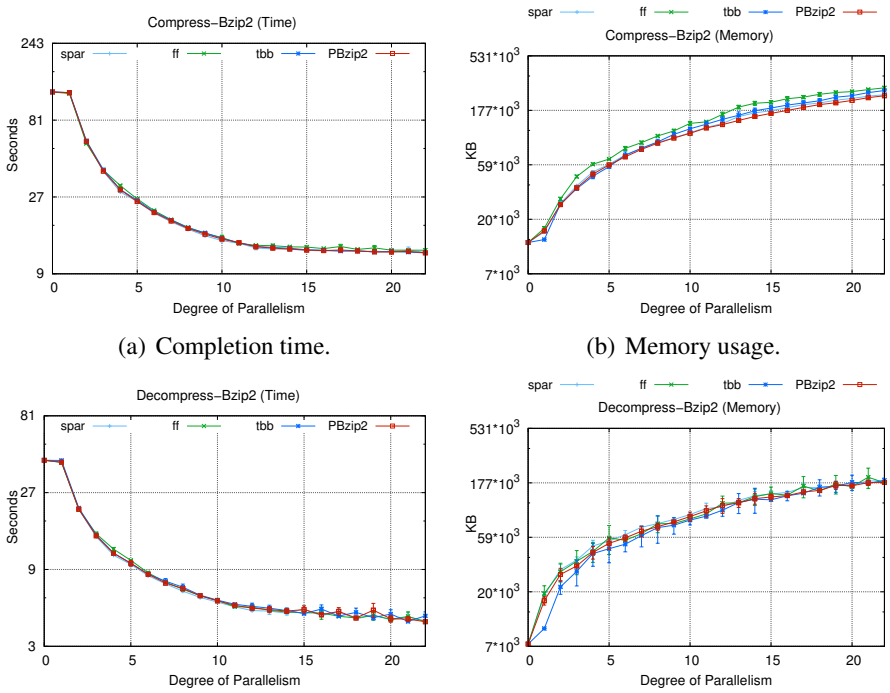


(a) Completion time. (b) Memory usage.



(c) Completion time. (d) Memory usage.

Figure 5. Bzip2 performance with SPar options.



(a) Completion time. (b) Memory usage.

(c) Completion time. (d) Memory usage.

Figure 6. Bzip2 performance comparison.

6. Conclusions

In this paper, we assessed the high-level and efficient stream parallelism of SPar for two real-world data compression applications. Our experiments demonstrated these characteristics through performance and programmability analysis of different parallelizations for the Bzip2 and Lzip with SPar compared to POSIX-Threads, FastFlow, and TBB. Therefore, we concluded that SPar is a suitable alternative for expressing parallelism in these applications. It provides good results on completion time, and memory usage. Also, it requires less code intrusion and the qualitative discussion revealed SPar with a simpler syntax. In the future, we plan to implement parallelism with SPar in other real-world applications, including other fields such as deep-learning, network monitoring, stream processing in the fog computing, and network package inspection.

Acknowledgements

Authors thank the partial financial support from CAPES and FAPERGS Brazilian research institutions as well as FACIN/PUCRS. Also, it was supported by the EU H2020-

Table 1. The best Speed-ups (S) for the parallelized versions.

	(a) Lzip(Com.)		(b) Lzip(Deco.)		(c) Bzip2 (Com.)		(c) Bzip2 (Deco.)	
Version	***Size***	***S***	***Size***	***S***	***Size***	***S***	***Size***	***S***
SPar	22	9.85	22	11.94	23	9.80	23	9.76
FF	22	9.82	22	12.12	23	9.74	22	10.04
TBB	24	9.98	21	12.13	22	9.98	23	10.09
Pthreads	22	10.01	24	12.39	24	9.91	24	10.31

ICT-2014-1 project RePhrase (No. 644235).

References

Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., and Torquati, M. (2011). Accelerating Code on Multi-cores with Fastflow. In *17th International Europen Conference on Parallel and Distributed Computing*, Euro-Par'11, pages 170–181, Bordeaux, France. Springer.

Aldinucci, M., Danelutto, M., Kilpatrick, P., and Torquati, M. (2014). FastFlow: High-Level and Efficient Streaming on Multi-core. In *Programming Multi-core and Many-core Computing Systems*, PDC, page 14. Wiley.

Beard, J. C., Li, P., and Chamberlain, R. D. (2015). RaftLib: A C++ Template Library for High Performance Stream Parallel Processing. In *6th Inter. Works. Progr. Models and App. for Multicores and Manycores*, PMAM' 2015, pages 96–105, San Francisco, USA. ACM.

Benkner, S., Bajrovic, E., Marth, E., Sandrieser, M., Namyst, R., and Thibault, S. (2012). High-Level Support for Pipeline Parallelism on Many-Core Architectures. In *18th International Europen Conference on Parallel and Ditributed Computing*, Euro-Par '12, pages 614–625, Rhodes Island, Greece. Springer.

Diaz, A. D. (2017). Lzip- LZMA Lossless Data Compressor. http://lzip.nongnu.org/.

Gilchrist, J. (2004). Parallel Compression with BZIP2. In *16th IASTED International Conference on Parallel and Distributed Computing and Systems*, PDCS' 04, pages 559–564, MIT, Cambridge, USA. ACTA Press.

Griebler, D. (2016). *Domain-Specific Language & Support Tool for High-Level Stream Parallelism*. PhD thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil.

Griebler, D., Danelutto, M., Torquati, M., and Fernandes, L. G. (2017a). SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):20.

Griebler, D., Filho, R. B. H., Danelutto, M., and Fernandes, L. G. (2017b). High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. In *10th International Symposium on High-Level Parallel Programming and Applications*, HLPP'17, pages 238–256, Valladolid, Spain.

ISO/IEC, . (2014). Information Technology - Programming Languages - C++. Technical report, International Standard, Geneva, Switzerland.

Laird, L. M. and Brennan, M. C. (2006). *Software Measurement and Estimation: A Practical Approach*. Wiley.

Reinders, J. (2007). *Intel Threading Building Blocks*. O'Reilly, USA.

Rixner, S., Dally, W. J., Kapasi, U. J., Khailany, B., Lbpez-Lagunas, A., Mattson, P. R., and Owens, J. D. (1998). A Bandwidth-Efficient Architecture for Media Processing. In *31st ACM/IEEE Inter. Symp. on Microarchitecture*, pages 3–13, Dallas, Texas, USA. 31st Annual ACM/IEEE International Symposium on Microarchitecture.

Seward, J. (2017). A Program and Library for Data Compression. http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.html.

Sutter, H. (2005). The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs journal*, 30(3):202–210.

ILUCTUS: Uma Biblioteca para o Apoio ao Processamento Colaborativo de Dados

Lucas Eduardo Bretana [*], Alana Schwendler [†], Gerson Geraldo H. Cavalheiro

[1]Laboratory of Ubiquitous and Parallel Systems
Programa de Pós-Graduação em Computação
Universidade Federal de Pelotas
Pelotas, RS – Brasil

{lebretana, aschwendler, gerson.cavalheiro}
@inf.ufpel.edu.br

Resumo. *Evoluir grandes volumes de dados requer sítios com capacidade de processamento e armazenamento de informação. O processamento distribuído utiliza tecnologias que permitem o compartilhamento de recursos e custos de processamento, e necessitam de ferramentas que façam a comunicação entre aplicação e processamento. O Espaço de Tuplas é um modelo de programação concebido sobre essas tecnologias que foi retomado neste trabalho como uma alternativa para o desenvolvimento de aplicações em ambiente de nuvem. A biblioteca ILUCTUS é apresentada neste artigo, bem como um estudo dos custos de suas operações elementares e aplicações.*

1. Introdução

Este trabalho está relacionado à manipulação, produção e compartilhamento de grandes volumes de dados. Vários estudos e pesquisas que trabalham manipulando grandes volumes de dados necessitam do processamento distribuído e compartilhamento de custos para a obtenção de resultados conclusivos. A demanda por sítios capazes de produzir e armazenar dados é constantemente notada no meio científico e acadêmico. O surgimento do protocolo HTTP (*The Hypertext Transfer Protocol*) advindo da necessidade do CERN (*Conseil Européen pour la Recherche Nucléaire*) [Berners-Lee et al. 1994] em armazenar dados gerados por pesquisas e que fiquem disponíveis para processamento é um exemplo histórico desta situação.

A demanda por processamento de grandes quantidades de dados pode ser ressaltada, também, no contexto de *Open Data*[Ebrahim and Irani 2005], onde bases de dados públicas são disponibilizadas na Internet. O objetivo é o compartilhamento destes dados para que a comunidade os manipule de forma a gerar resultados e, portanto, novos conhecimentos. No entanto, as políticas usualmente aplicadas nestas bases de dados não preveem que o resultado dos processamentos utilizados sobre os dados disponibilizados sejam agregados a estas. Como consequência, os resultados podem ter um impacto menor que o desejado no crescimento do conhecimento além de haver o claro risco de duplicidade de uso de recursos de processamento quando outra pesquisa/pesquisador visa obter o mesmo resultado, reproduzindo o mesmo trabalho.

[*]Bolsista IC Capes
[†]Bolsista IC Capes

Neste artigo é apresentada uma alternativa à gestão de bases de dados *Open Data*. A contribuição central é permitir que a oferta de dados receba como contrapartida dos pesquisadores beneficiados, os resultados obtidos, permitindo o crescimento da própria base. Outro ponto a ser considerado é que os custos de armazenamento são compartilhados entre os pesquisadores que estão utilizando a base de dados para evolui-la. A proposta segue a linha de pensamento das licenças como o *Creative Commons* [Commons 2016] e *GNU General Public License* (GPL) [Gough 2009] para evolução de dados por uma comunidade de interesse. A proposta oferece um modelo de negócio entre o proprietário da base de dados e seus usuários, que reza que os dados obtidos a partir do processamento sobre os dados originais também serão compartilhados entre todos aqueles que também fazem, ou venham a fazer, uso da base de dados original.

A partir de estudos sobre os trabalhos relacionados como licenças de compartilhamento, entre outros, tornou-se possível o desenvolvimento de uma biblioteca que explora os recursos de nuvem [Armbrust et al. 2010] utilizando o modelo Espaço de Tuplas (TS, do inglês *Tuple Space*) para coordenação e comunicação entre as atividades.

A utilização de nuvem em união com o conceito de Espaço de Tuplas permite construir um modelo para garantir evolução de dados de maneira distribuída. Um dos benefícios que podem ser obtidos é a colaboratividade na evolução de dados em uma aplicação com uma larga escala de informações que necessitam ser processadas. Uma vez que exista uma aplicação com a necessidade de processamento distribuído, ela pode ser compartilhada num espaço em nuvem onde existirão colaboradores intencionados a evoluir os dados da pesquisa para conseguir obter os resultados desejados, dividindo custos de processamento e, eventualmente, de armazenamento. Dentre exemplos de aplicações que fariam uso desse modelo de programação, citam-se o treinamento de redes neurais, cálculo do escavamento de túneis e sequenciamento do DNA.

O restante do artigo está organizado como segue. A próxima seção, seção 2, caracteriza o modelo de negócio concebido para desenvolver o trabalho, considerando o contexto de compartilhamento de informações. Seguidamente, a seção 3 descreve como se deu a implementação da biblioteca sugerida e relata quais ferramentas foram utilizadas em 3.1. Posteriormente, a seção 4 fala sobre o ambiente de armazenamento e compartilhamento dos dados da aplicação. Por fim, seguem as seções 5 que descreve alguns testes que foram realizados para aferir o funcionamento deste trabalho, a seção 6 traz alguns trabalhos e pesquisas semelhantes à apresentada neste artigo e na seção final são apresentadas as conclusões e trabalhos futuros.

2. Modelo de negócio

O modelo de negócio identificado se desenvolve no contexto do processamento e armazenamento colaborativo de grandes quantidades de dados gerados por uma pesquisa e que serão manipulados por uma comunidade de interesse em comum. O modelo de aplicação concebido considera a existência de um ambiente de pesquisa sob o qual são desenvolvidos **Projetos Colaborativos**. Um Projeto Colaborativo requer um ambiente em nuvem que proveja recursos para armazenamento de dados, políticas próprias para autentificação de usuários e compartilhamento de dados e ainda uma interface de programação aplicativa (API) com primitivas que permitam a manipulação dos arquivos na nuvem.

Um Projeto Colaborativo é instanciado por um ator do modelo de negócios iden-

tificado como **Administrador**. Este Administrador, além de criar o Projeto Colaborativo, também disponibiliza o primeiro conjunto de dados, o qual será operado segundo as **Políticas de Colaboração** do sistema. São estas políticas que definem as regras de como os **Colaboradores**, outros atores do modelo, irão participar do projeto acessando os dados compartilhados e oferecendo acesso aos resultados por eles próprios produzidos.

Políticas de Colaboração consistem em um conjunto de regras que definem as relações de compartilhamento de informação entre os participantes bem como operações de compartilhamento dos dados entre os diferentes atores que compõem o projeto. As regras iniciais destas políticas definem como se dá o processo de autentificação na aplicação, e para isso é necessária a troca de *tokens* de identificação. Da parte do Administrador, esses *tokens*, ou chaves, identificam o projeto e autorizam uma aplicação a ter acesso aos dados compartilhados na nuvem. *Tokens* são únicos para cada Colaborador em um Projeto Colaborativo. Pelo lado do Colaborador, o *token* identifica-o dentro do projeto, oferecendo, desta forma, acesso aos seus resultados na nuvem. Nesta troca inicial de informações entre o Administrador e Colaborador, além dos *tokens*, também é feita a troca de quaisquer outras informações necessárias para dar inicio ao processamento dos dados, tais como identificação da base de dados primária, lista de bases já processadas por Colaboradores pré-existentes, permissões sobre estas bases etc.

Além dos processos iniciais de troca de informações, as Políticas de Colaboração também rege o acesso a uma base de dados que já compõe o Projeto, protegendo o sistema de ter as bases de dados alteradas inadvertidamente por qualquer Colaborador. A proposta não inclui sistema de recuperação, considerando que cada Colaborador é responsável pelos dados gerados pela sua contribuição e que este não tem acesso destrutivo aos dados produzido pelos demais atores.

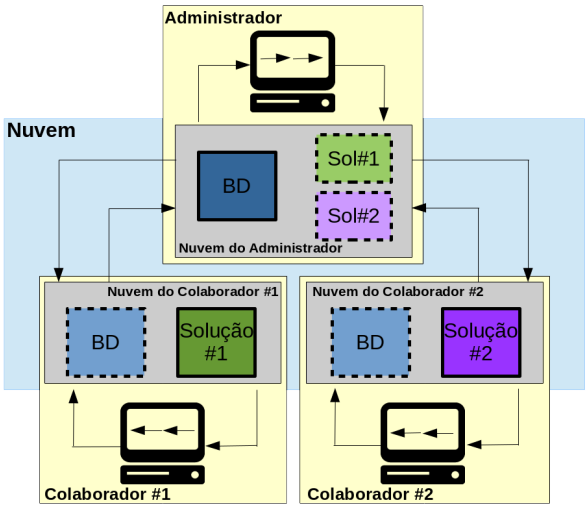


Figura 1. Ilustração do modelo de negócio.

A Figura 1 ilustra o modelo de comunicação entre Administrador e Colaboradores no ambiente de desenvolvimento de Projetos Colaborativos concebidos. Nas bases de dados (BD), a borda contínua representa a base de dados original na nuvem onde foi primeiro formada. O compartilhamento é representado pelas setas entre as nuvens de origem e destino. O conjunto destas BDs representa a nuvem de dados manipulada pelo

Projeto Colaborativo, oferecendo amplo acesso a toda coleção de dados, sem distinção se o dado foi produzido pelo Administrador ou algum Colaborador.

São considerados dois cenários de desenvolvimento de um Projeto Colaborativo. O primeiro é caracterizado como um **Projeto Fechado**, no qual o Administrador disponibiliza uma **aplicação**, que já inclui as Políticas de Colaboração e automatiza os processos de autentificação. Tal aplicação é fechada, e aplica uma determinada heurística para o processamento dos dados, e tem por objetivo explorar o poder de processamento e de armazenamento disponibilizado pelos Colaboradores. O processamento desta aplicação se dá nos recursos próprios aos Colaboradores, sendo, então, na nuvem, disponibilizados os resultados do processamento realizados.

O segundo cenário é descrito por um **Projeto Aberto**. Neste cenário, o Colaborador não apenas oferece seus recursos de processamento, mas também sua *expertise* no tema, oferecendo novas heurísticas, na forma de aplicações por ele próprio desenvolvidas, para manipular os dados. Em um Projeto Aberto, os *tokens* de autentificação recebidos pelo Colaborador permitem que sua aplicação seja conectada a um Projeto Colaborativo.

3. Contexto de Implementação

Para o desenvolvimento deste trabalho foi necessário o estudo de meios comunicação que servissem de apoio para o compartilhamento de dados de modo distribuído entre diferentes sítios de processamento. Outra necessidade era definir uma tecnologia de nuvem que fornecesse a mecânica básica de comunicação por meio de uma API, bem como documentação clara e completa do uso e funcionamento dessa API. Por último, uma linguagem de programação que pudesse fazer uso desta API e fornecesse o necessário para desenvolver o modelo de comunicação escolhido.

3.1. Ferramentas

Uma das necessidades encontradas era definir um mecanismo de comunicação que permitisse a colaboração de tarefas de forma distribuída. Exemplos de mecanismos possíveis para essa comunicação são troca de mensagem [Cáceres et al. 2001], RMI [ORACLE 2016], DSM [Yu and Cox 1997], Espaço de Tuplas (TS, do inglês *Tuple Space*). O modelo de comunicação escolhido foi o Espaço de Tuplas, que, apesar de apresentar altos sobrecustos de comunicação nos ambientes para processamento distribuído utilizados anteriormente ao advento da tecnologia de nuvem, oferece uma camada de abstração para colaboração com uma semântica bastante clara. As questões de desempenho, em um ambiente distribuído de larga escala, diluem-se com os benefícios obtidos pela camada de desenvolvimento.

Um TS consiste basicamente de um espaço de endereçamento compartilhado onde os dados são manipulados como tuplas que seguem o formato <*key*, *content*>. O campo *key* de uma tupla é um identificador e deve ser único dentro do TS. O *content* é o dado da tupla em si, sendo este um dado de tipo genérico. O Espaço de Tuplas prevê a existência de operações simples de leitura, escrita e remoção para manipulação das tuplas. Estas operações devem então ser concretizadas pelas implementações de TS. A escolha do Espaço de Tuplas como o modelo a ser utilizado se deu principalmente por ter uma semântica conhecida e que está apta para o modelo de memória distribuída.

Dentre os modelos que descrevem uma implementação de Espaço de Tuplas o escolhido foi o modelo de Linda [Ahuja et al. 1986]. A escolha se deu pela simplicidade com que os métodos de comunicação descritos por esse modelo se adaptam à solução do problema de compartilhamento de dados. Os meios para manipulação destas tuplas são por operações básicas de de manipulação dos dados. É importante notar que o Linda é um modelo e por isso ao ser implementado pode-se apresentar outras primitivas de comunicação. Linda define quatro primitivas de interação com os dados, sendo elas:

1. `Read:` lê uma tupla do TS com base em um critério de identificação;
2. `In:` lê e retira uma tupla do TS com base em um critério de identificação;
3. `Out:` atribui uma nova tupla no TS para ser computada;
4. `Eval:` recebe um método e uma tupla, aplica o método à mesma e salva o resultado no TS, no formato de uma tupla.

Para a escolha da nuvem se levou em consideração aquelas que dispõem de uma API para gerenciamento de autentificação e manipulação dos dados. Outra característica importante era facilidade da aquisição de uma nuvem particular por um usuário comum, aumentando o número de Colaboradores em potencial. Dentre as nuvens que atendiam aos requisitos necessários a escolha foi pela nuvem do Dropbox [Drago et al. 2012]. Esta decisão se deu pois, na ocasião da implementação, foi aquela que apresentou um processo de autentificação robusto e a melhor documentação sobre a API.

A opção por Java se deu considerando tanto a popularidade desta como pela portabilidade oferecida pelo seu modelo de implementação baseado em máquina virtual. Outro aspecto positivo é a existência de uma API do Dropbox desenvolvida para esta linguagem.

3.2. Implementação

A implementação do modelo de negócio descrito foi feita no formato de uma biblioteca que ganhou o nome de ILUCTUS. Esta biblioteca permite que sejam desenvolvidos Projetos que utilizem a nuvem do Dropbox como memória compartilhada, no formato de Espaço de Tuplas, para o processamento distribuído de dados. Para isso, se desenvolveu métodos para manipular os dados usando operações de leitura, escrita e remoção de tuplas. A ILUCTUS concretiza essas operações em primitivas baseadas nas descritas pelo modelo de Linda.

Para a criação das primitivas foi definida a representação de uma tupla dentro do sistema de arquivos oferecido pelo Dropbox. Para este trabalho definiu-se que o campo *key* de uma tupla é usado como o nome do arquivo que contém os dados referentes àquela tupla. O campo de dados da tupla, *content*, é o conteúdo do arquivo no sistema de arquivos do Dropbox. A escrita e leitura dos objetos Java para arquivos é feita usando recursos internos da Máquina Virtual Java (*Java Virtual Machine* ou JVM). Para isso basta que o tipo do campo *content* implemente a interface *Serializable* da API Java. Usando as ferramentas da JVM um objeto Java pode ser convertido para uma *stream* de *bytes*, o que torna trivial a sua escrita em arquivo. É importante ressaltar que operação inversa também existe e é correspondente, operando uma *stream* de *bytes* válida a JVM retorna um objeto Java.

Nesta biblioteca o meio pelo qual é feita a identificação de uma tupla é utilizando o recurso de expressões *lambda*. Dentro do sistema de Java uma expressão lambda é tipada sobre um Interface que recebe a anotação de *FunctionalInterface* e nesta Interface

se generaliza o funcionamento da expressão *lambda*. Para a expressão *lambda* que faz a identificação de uma tupla dentro do TS a Interface foi desenvolvida de forma a conter um método que recebe como parâmetro uma tupla e retorna um valor *boolean*. O funcionamento do procedimento que identifica a tupla (*match*) fica abstraído e deve ser concretizado em cada uso das primitivas que fazem busca no TS.

As operações de leitura do TS são realizadas pelas primitivas `Read` e `In`. `Read` faz uma busca dentro do TS utilizando uma expressão *lambda* recebido como argumento para identificar a tupla desejada. A implementação do `Read` é não bloqueante, desta forma a primitiva retorna a primeira tupla que for identificada, ou *null* em caso do dados buscado não ser encontrado. O funcionamento da primitiva `In` é semelhante à `Read`, porém uma vez que a tupla é encontrada a sua referência dentro do TS é removida.

As operações para escrita no TS são as primitivas `Out` e `Eval`. A implementação da primitiva `Out` recebe uma tupla como argumento e insere esta tupla para o TS. Para esta implementação a escolha foi por uma abordagem destrutiva, onde se houver uma ocorrência da mesma chave no TS, o dado correspondente será sobrescrito. A primitiva `Eval` recebe uma expressão *lambda* para fazer a identificação de uma tupla dentro do TS e sobre esta tupla é aplicada uma função para modificar o seu estado. A função a ser aplicada pela primitiva também é recebida por argumento como uma expressão *lambda*. Esta *lambda* é tipada sobre uma interface que generaliza um método que tem como parâmetro uma tupla e que retorna, também, uma tupla. Uma vez que tupla é identificada e processada, ela é atribuída novamente ao Espaço de Tuplas usando o mesmo identificador da tupla original.

4. Armazenagem e compartilhamento

A Figura 2 ilustra como ocorrem os processos de armazenagem e compartilhamento das bases de dados. Nesta figura, as bases de dados são representadas com borda pontilhada ou contínua, referenciando respectivamente, à base na sua nuvem original ou uma cópia proveniente do compartilhamento. Já os processos de compartilhamento são ilustrados pelas ações **Requisita acesso**, onde é feita a requisição de uma base de dados, e **Concede acesso**, onde o acesso é garantindo e então a base é copiada para a nuvem do requisitante.

O processamento distribuído dos dados ocorre quando existe o compartilhamento das bases de dados que são então processadas separadamente. Neste compartilhamento é feita a cópia dos dados requisitados para a nuvem do Colaborador. A base de dados que fornece os materiais para processamento fica na nuvem do Colaborador até o fim da execução da aplicação e então é removida, de forma a não onerar sobrecustos de armazenagem, uma vez que o programa terminou. Na figura isso fica claro ao notar a cópia da base de dados *init* na nuvem dos Colaboradores.

Por outro lado a base de dados criada a partir dos resultados da computação no Colaborador existe unicamente na nuvem deste Colaborador até que a aplicação termine. Desta forma temos, além do processamento distribuído, a divisão dos custos de armazenamento dos dados. Na figura, essa divisão dos custos é expressa pela ausência da base de dados *Solução 2* na nuvem do Administrador pois o Colaborador 2 ainda não terminou a execução e desta forma seus resultados não estão compartilhados.

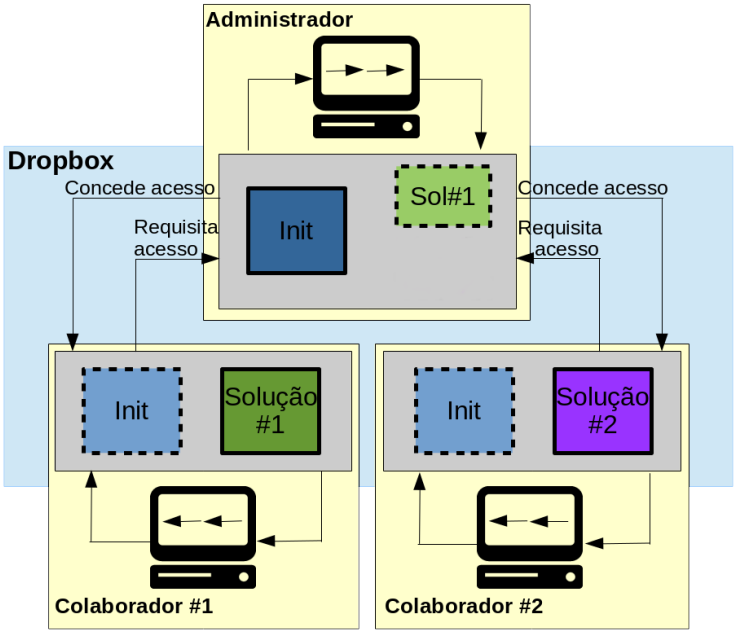


Figura 2. Distribuição do armazenamento e fluxo de comunicação para compartilhamento de BDs.

5. Caso de Teste

Para fazer a verificação da funcionalidade desta biblioteca apresentada no artigo, foram realizados dois tipos de testes. No primeiro caso, 5.1, foram feitas execuções repetidas das operações básicas com as quais a biblioteca opera, de forma a se obter os tempos médios de execução das mesmas. Em um segundo momento, 5.2 foi realizado um caso de teste para estressar a aplicação, utilizando a biblioteca desenvolvida em duas abordagens diferentes. Neste caso de teste também foi avaliado o tempo médio de execução com o desvio padrão.

5.1. Teste das primitivas

A manipulação das tuplas é feita pelas primitivas descritas. Logo, foram feitas execuções das primitivas `Read`, `In` e `Out` de modo a fazer a coleta dos seus tempos de execução em alguns cenários variados. Para `Read` e `In` foram realizados buscas no TS com poucas tuplas (dez), uma quantidade média (cinquenta) e uma quantidade grande (cem). Para `Out` não se fazem necessárias essas variações, já que sua ação é destrutiva. Os testes com a primitiva `Eval` foram dispensados pois a mesma se trata de uma execução sequencial de outras primitivas.

Tabela 1. Tempos médios e desvio padrão de cada cenário de execução

	Out	Read			In		
	-	P	M	G	P	M	G
μ	1.09s	3.78s	16.3s	32.33s	7.27s	16.29s	33.15s
δ	0.36s	0.37s	0.65s	1.57s	0.77s	0.49s	0.99s

Deve ser levado em consideração que o Dropbox exige um tempo de espera entre requisições feitas pela API e que os resultados apresentados na Tabela 1 são referentes à 30 execuções de cada primitiva feitas a partir de computador na rede da UFPel. É possível observar que as primitivas de leitura, i.e., `Read` e `In`, tem um tempo médio maior que a primitiva `Out`. Isto coincide com o esperado, uma vez que na leitura é feito um número maior de requisições para a nuvem do Dropbox.

5.2. Caso de teste Fibonacci

Para representar a produção de informação, foi implementada uma aplicação sintética reproduzindo o cálculo recursivo de Fibonacci dado pela equação 1. Opta-se pelo método recursivo para que seja possível gerar uma maior carga de processamento e assim calcular de maneira mais fiel o desempenho da aplicação. Com as execuções feitas, foram coletados os tempos em que os cálculos foram realizados.

$$F_{(n)} = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{(n-1)} + F_{(n-2)}, & \text{if } n > 1 \end{cases} \quad (1)$$

Na execução dos casos de teste, a chave da tupla representa a posição da série de Fibonacci desejada, e o conteúdo da tupla é o respectivo valor da posição identificada na série. Se a tupla estiver apenas com a identificação preenchida e seu conteúdo estiver nulo, significa que seu valor ainda não foi calculado e que ela está disponível para o processamento. Caso o valor da tupla esteja preenchido, a posição da série correspondente àquele valor já foi calculada e disponibilizada para o resto da aplicação.

Inicialmente o Administrador disponibiliza diversos valores a serem calculados utilizando da primitiva `Out` da biblioteca. Uma vez que temos a base inicial, agora um Colaborador, ou o próprio Aministrador, utiliza de uma aplicação que fará o calculo para realizar as tarefas que estão no TS. Dentro das execuções do caso de teste foram propostas duas heurísticas:

1. Utilizar a primitiva `In` passando uma expressão lambda que retorne apenas as tuplas com o valor nulo, chama novamente a primitiva `In` para verificar se o valor do índice anterior já foi calculado e chama também para verificar o índice precedente ao anterior já foi calculado, i.e., verifica se os valores anteriores de Fibonacci já estão calculados. Caso estes valores estejam nulos, i.e. ainda não foram calculados, são realizados primeiro os seus cálculos, fazendo sempre esta checagem recursivamente, até que seja possível calcular o valor real desejado. Com o detalhe de que cada vez que a primitiva `In` retira uma tupla do ambiente compartilhado e calcula o valor, a primitiva `Out` é chamada para escrever o valor que foi processado ou o mesmo valor, caso a aquela estivesse com o seu valor calculado;
2. Utilizar a primitiva `In` passando uma expressão lambda que retorne apenas as tuplas com o valor nulo, chamar a primitiva `Read` para verificar se o valor anterior e precedente ao anterior já foram calculados. Caso estes valores não tenham sido processados ainda, é chamado o método de cálculo, porém estes valores não são escritos no ambiente compartilhado. O único valor que será escrito é o valor real que deseja ser calculado.

As duas heurísticas listadas foram implementadas e os resultados obtidos podem ser observados na tabela 2. Na tabela, o tempo médio é representado pelo símbolo *mu* (μ) e o desvio padrão é representado pelo símbolo *delta* (δ). Percebe-se que a segunda heurística obteve melhor desempenho que a primeira heurística, já que é realizada a escrita do resultado apenas para o valor da série que é desejado, e não de todos os valores calculados como faz a primeira heurística. Para os dados apresentados na tabela deve-se considerar que o Dropbox exige um tempo mínimo entre as requisições e que os presentes resultados correspondem a média de 10 chamadas a cada primitiva a partir de um computador localizado na rede da UFPel.

Tabela 2. Tempos de execução das heurísticas

Local	H#1		H#2	
-	μ	δ	μ	δ
Sítio#1	203.19min	3.46	170.70min	4.02
Sítio#2	207.35min	1.38	169.90min	4.36

Como foi mencionando anteriormente as primitivas retornam a primeira tupla cujo a expressão *lambda* faça *match* e a ordem na qual elas são verificas depende da organização dos arquivos dentro da nuvem. Na execução destes testes verificou-se que a ordem na qual a nuvem coloca os arquivos é de acordo com o *timestamp* da criação. Por isso as primeiras tuplas a serem verificadas foram as com maior índice na sequencia Fibonacci e desta forma a primeira heurística apresenta um sobre custo de reescrever as tuplas que já foram calculadas, o que explica porque seus tempo médio foi maior.

6. Trabalhos Relacionados

Após uma pesquisa bibliográfica sobre o assunto foram encontradas trabalhos muito bem consolidados que serviram de embasamento para o desenvolvimento da proposta apresentada, destacando-se o Seti@Home, BOINC e o modelo EvoSpace. Vale destacar que uma das dificuldades deste trabalho foi encontrar trabalhos relacionados mais recentes.

O Seti@Home [Korpela et al. 2001] é uma tecnologia desenvolvida para realizar processamento distribuído de informações coletadas em sua pesquisa. Esta pesquisa é voltada para coleta de dados vindos do espaço. Esta tecnologia tem o seguinte funcionamento: o usuário que desejar ceder seu poder de processamento para a aplicação pode se cadastrar como colaborador, e sempre que seu computador estiver ocioso, alguns dados serão enviados para o seu sítio de processamento e a aplicação vai ser executada em segundo plano, enquanto houver processamento disponível para evoluir os seus dados. Uma vez que respostas são obtidas, elas são enviadas para a aplicação original, para seguir com o desenvolvimento dos dados da base original. O processo que está sendo executado no sítio de processamento fica como uma espécie de proteção de tela, permitindo que todo o poder de processamento ocioso se foque em trabalhar em cima da aplicação [Anderson et al. 2002]. Neste sentido, a base inicial da ILUCTUS foi dada pelo Seti@Home, pela sua ideia de processar os dados de maneira distribuída com vários colaboradores possíveis para a mesma aplicação. A biblioteca ILUCTUS se diferencia apresentando um modelo de Projeto Aberto onde que cada Colaborador pode aplicar sua própria heurística de evolução.

BOINC (Berkeley Open Infrastructure for Network Computing) [Anderson 2004] é uma ferramenta de gerenciamento de aplicações distribuídas, desenvolvida na Universidade de Berkeley - Califórnia. Assim como o Seti@Home, existem outras aplicações que fazem o mesmo tipo de processamento de dados porém com outros focos de pesquisa. O BOINC realiza uma espécie de ponte entre a aplicação e o sítio de colaboração. A partir dele, é possível escolher para qual aplicação o poder de processamento será cedido. Em união com a ideia do Seti@Home, o BOINC é uma interface de gerenciamento de aplicações distribuídos, o que baseia interface planejada para esta aplicação.

O modelo EvoSpace é uma plataforma evolutiva e distribuída baseada no modelo de Espaço de Tuplas [García-Valdez et al. 2013]. Esta plataforma é utilizada para o armazenamento de algoritmos evolutivos, utilizando o recurso da nuvem para isto, podendo ser utilizado como modelo de *Platform as a Service* (PaaS). Este modelo também trata de alguns problemas como redundâncias de trabalho, *starvation, starvation of the population pool*, insegurança de usuários conectados e um grande espaço de parâmetros.

Uma das aplicações de uso citadas é a de *Open Data*. As bases de dados abertas operam de maneira a serem preenchidas a partir de pesquisas de campo [Bouguettaya et al. 2001], onde os dados são coletados através de analises de comunidades precárias, censos, pesquisas demográficas etc. Pesquisas desta natureza geram grandes volumes de dados e, geralmente, precisam ser processadas para gerar estatísticas e resultados para trazer a melhoria das comunidades estudadas. Estas bases de dados são carregadas com quantidades exorbitantes de dados que precisam ser processados. Já que os dados estão disponíveis abertamente, podem ser usados por vários colaboradores para que os resultados necessários sejam alcançados.

A proposta deste artigo prevê a Colaboração e compartilhamento de custos de armazenamento em mais de um sítio. Estes trabalhos apresentam funcionalidades características que podem ser incorporadas à ILUCTUS. A proposta deste artigo utiliza os mesmos conceitos mas com tecnologias mais atuais, como o ambiente de nuvem servindo de ambiente de compartilhamento e a linguagem de programação Java em uma versão com mais funcionalidades agregadas.

7. Conclusões e trabalhos futuros

Neste trabalho foi apresentada ILUCTUS, uma biblioteca que permite a competência no processamento de dados em larga escala. Esta ferramenta foi concebida para atender uma crescente demanda do uso de nuvens computacionais para provisionamento de recursos para armazenamento de grandes coleções de dados. ILUCTUS foi implementada de forma a explorar a tecnologia de nuvem provida pelo DROPBOX como um Espaço de Tuplas distribuído em larga escala. Uma política de controle de acesso garante a integridade da base de dados construída na forma de um Projeto Colaborativo, no qual os papéis dos atores, Administrador e Colaborador, são claramente definidos. Nos cenários projetados, a adesão de Colaboradores aos Projetos Colaborativos permite racionalizar o uso de recursos computacionais, além de repartir custos de processamento e armazenamento dos dados.

Alternativas desta natureza de processamento distribuído significam um avanço científico, pois ocorre o compartilhamento de custos de processamento e armazenamento. A biblioteca desenvolvida neste trabalho traz estas possibilidades de compartilhamento de

custos. As tecnologias estão em evolução e, com isto, é dada a motivação para explorar os serviços de nuvem como um meio para compartilhar dados de pesquisas, garantindo com que novos avanços tecnológicos e pesquisas se beneficiem destes serviços.

Para trabalhos futuros serão desenvolvidos variações das primitivas. Podemos destacar a promoção de escritas não destrutivas que podem tirar vantagem do sistema de versionamento de arquivos disponibilizado pelo Dropbox. Além desta, tem-se a possibilidade de implementar leituras que retornam todas as tuplas que satisfazem os requisitos de assimilação e leitura bloqueantes. Além dos avanços para novas primitivas pretende-se desenvolver novas aplicações que usufruam da biblioteca para evolução de dados, tornando-se melhores métricas de avaliação da biblioteca.

8. Agradecimentos

Agradecemos à Universidade Federal de Pelotas (UFPel), aos órgãos de fomento CNPq e Fapergs e ao grupo de pesquisa LUPS.

Referências

Ahuja, S., Curriero, N., and Gelernter, D. (1986). Linda and friends. *Computer;(United States)*, 19(8).

Anderson, D. P. (2004). Boinc: A system for public-resource computing and storage. pages 4–10.

Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. (2002). Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. (2010). A view of cloud computing. *Communications of the ACM*, 53(4):50–58.

Berners-Lee, T., Dimitroyannis, D., Mallinckrodt, A. J., McKay, S., et al. (1994). World wide web. *Computers in Physics*, 8(3):298–299.

Bouguettaya, A., Ouzzani, M., Medjahed, B., and Cameron, J. (2001). Managing government databases. *Computer*, 34(2):56–64.

Cáceres, E. N., Mongelli, H., and Song, S. W. (2001). Algoritmos paralelos usando cgm/pvm/mpi: uma introduçao. In *XXI Congresso da Sociedade Brasileira de Computaçao, Jornada de Atualizaçao de Informática*, pages 219–278.

Commons, C. (2016). Creative commons.

Drago, I., Mellia, M., M Munafo, M., Sperotto, A., Sadre, R., and Pras, A. (2012). Inside dropbox: understanding personal cloud storage services. pages 481–494.

Ebrahim, Z. and Irani, Z. (2005). E-government adoption: architecture and barriers. *Business process management journal*, 11(5):589–611.

García-Valdez, M., Trujillo, L., de Vega, F. F., Guervós, J. J. M., and Olague, G. (2013). Evospace: a distributed evolutionary platform based on the tuple space model. pages 499–508.

Gough, B. (2009). *GNU scientific library reference manual*. Network Theory Ltd.

Korpela, E., Werthimer, D., Anderson, D., Cobb, J., and Lebofsky, M. (2001). Seti@ home—massively distributed computing for seti. *Computing in science & engineering*, 3(1):78–83.

ORACLE (2016). Trail: Rmi. java documentation. tutorials. Disponível em: <http://docs.oracle.com/javase/tuorial/rmi>. Acesso em: Julho de 2017.

Yu, W. and Cox, A. (1997). Java/dsm: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224.

Compactação do Algoritmo de Comparação de *Strings* do Snort para o uso na Memória Compartilhada de GPUs

José Bonifácio da Silva Júnior[1], Edward David Moreno[1], Ricardo Ferreira dos Santos[2]

[1]Departamento de Computação da Universidade Federal de Sergipe – DCOMP/UFS

[2]Departamento de Computação da Universidade Federal de Viçosa

boni14_gto@hotmail.com, {edwdavid,cacauvicosa}@gmail.com

Resumo. *A tarefa de comparar assinaturas de ataques com pacotes de redes em um Intrusion Detection System (IDS) consome bastante tempo de CPU. Para amenizar esse problema, tem-se tentado paralelizar o motor de comparação dos IDSs transferindo sua execução da CPU para a GPU. Este artigo mostra o processamento em paralelo dos dados no algoritmo de comparação de string Aho-Corasick e propõe compactar a Tabela de Transição de Estados desse algoritmo a fim de possibilitar o uso dele na memória compartilhada. A paralelização foi feita através da plataforma CUDA da NVIDIA e executada nas diversas memórias da GPU. O algoritmo AC foi compactado e executado na memória compartilhada, alcançando, em seu melhor resultado, um ganho de desempenho de 73% em relação às outras memórias da GPU e o algoritmo compactado chegou a ser 56 vezes mais rápido que sua versão serial. Com isso, pode-se perceber que o uso da compactação na memória compartilhada torna-se uma solução adequada para acelerar o processamento de IDSs que necessitem de agilidade na busca por padrões.*

1. Introdução

A tecnologia da informação (TI) tem sido cada vez mais determinante para o sucesso de empresas e organizações ao redor do mundo, sendo as redes de computadores como um dos seus principais meios de transmissão de dados. Juntamente com o aumento da importância da TI, cresceu também a necessidade de proteção do seu maior patrimônio, a informação.

Uma das formas de proteção passa pelo uso do IDS, uma ferramenta baseada em assinaturas que analisa os cabeçalhos dos pacotes e inspeciona as cargas, comparando-os com um grande conjunto de regras, ou seja, uma coleção de assinaturas de ataques conhecidos, tais como: vírus, *worms*, *spyware* ou código malicioso [Jaiswal 2014].

O Snort é um dos IDSs mais utilizados, sendo um software *open source* para UNIX e Windows. Ele é capaz de detectar quando um ataque está sendo realizado e, baseado nas características do ataque, alterar ou remodelar a configuração do sistema de acordo com as necessidades, e alertar o administrador do ambiente sobre esse ataque [Santos 2005]. Ele monitora o tráfego da rede pacote a pacote e em tempo real para verificar se o pacote que chega na interface coincide com algumas das suas assinaturas pré-configuradas.

Para alcançar isso, o Snort v2.9.7.3 usa o algoritmo de Aho-Corasick (AC) [Snort

Team 2015] para fazer a comparação do *payload* do pacote com a coleção de assinaturas. Isto é um problema, pois só essa atividade consome cerca de 70 a 80% do tempo de processamento [Jaiswal 2014]. Em redes de alta velocidade essa comparação pode sobrecarregar a CPU, fazendo-a deixar de executar os outros processos necessários para a execução do Snort ou de outra aplicação que estiver em execução no host.

Essas falhas fazem da paralelização da comparação de *strings* utilizando as *Graphic Processing Unit* (GPUs) uma solução adequada para o problema, já que as GPUs têm um maior poder de computação do que as CPUs, como pode ser visto em alguns trabalhos. Por exemplo, Lin, C. et al. (2013) paralelizaram o processamento do algoritmo de comparação de *string* Aho-Corasick (AC) e alcançaram uma velocidade 74,95 vezes maior que a versão serial do mesmo algoritmo. Tran, N. et al. (2012) paralelizaram os dados de entrada do algoritmo AC e obtiveram uma melhoria de 15,72 vezes na velocidade de comparação de *strings* em relação à versão serial. Já o trabalho de Thambawita, D. et al. (2014) mostrou que se o texto onde serão procurados os padrões for maior que 40000 bytes (o que tende a acontecer quando um *host* IDS é colocado em uma rede de alta velocidade) o desempenho da GPU supera o da CPU. Os resultados desses trabalhos mostraram que houve ganho significativo da GPU em relação à CPU, o que, por si só, serve como justificativa para continuar as pesquisas nesta linha de estudo.

A paralelização do processamento de dados no algoritmo AC já mostra ser um caminho natural para acelerar a comparação de *strings* do Snort. Porém, com o passar do tempo, novos ataques vão surgindo e consequentemente o número de assinaturas tende a aumentar. Isso leva a uma preocupação a mais com a limitação da quantidade de memória das GPUs, principalmente das memórias mais velozes, como é o caso da memória compartilhada, por exemplo. Sendo assim, a compactação do algoritmo AC na GPU é uma alternativa para superar esse problema.

O presente artigo tem como objetivo principal realizar a paralelização do processamento dos dados do algoritmo AC, dividindo os dados a serem analisados em partes e processando-os em paralelo, explorar a hierarquia de memórias da GPU a fim de verificar onde melhor se encaixam os dados e o algoritmo e, também, aplicar uma compactação na STT do algoritmo a fim de possibilitar sua execução na memória compartilhada.

O artigo está organizado da seguinte forma: a Seção 2 mostra a hierarquia de memórias de uma GPU genérica. A Seção 3 traz o funcionamento do algoritmo AC e como o mesmo pode ser implementado em uma GPU. A Seção 4 mostra os trabalhos relacionados à paralelização do processamento de dados do algoritmo AC em GPUs. A Seção 5 explica a compactação eficiente desenvolvida neste trabalho. Os resultados experimentais utilizando a compactação são detalhados na Seção 6, seguidos da conclusão na Seção 7.

2. *Graphic Processing Unit* (GPU)

Nesta seção é dada uma breve descrição da arquitetura de uma GPU, com foco na sua hierarquia de memórias.

As Unidades de Processamento Gráfico são dispositivos de processamento de elementos gráficos introduzidos na década de 1980 para descarregar os processamentos gráficos relacionados às *Central Processing Units* (CPUs) [Kouzinopoulos e Margaritis

2008]. As GPUs modernas são programáveis e possuem processadores de *streams* capazes de fazerem computação de alto desempenho.

Pode-se ver pela Figura 1 que uma GPU possui alguns tipos de memórias que podem ser utilizadas em uma aplicação paralela. A memória global é uma memória localizada no *off-chip* DRAM e é através dela que a CPU se comunica com a GPU. A memória compartilhada fica dentro de cada bloco de *thread* e é compartilhada entre os *threads* em execução no bloco. O tempo de acesso coincide de perto com o tempo de acesso de um registrador, portanto, é uma memória muito rápida [Tran, et al. 2012].

Existem também as memórias constante e de textura localizadas na área *off-chip* DRAM. Estas memórias possuem, respectivamente, a cache constante e a cache de textura no chip, que podem armazenar dados somente leitura [Tran, et al. 2012].

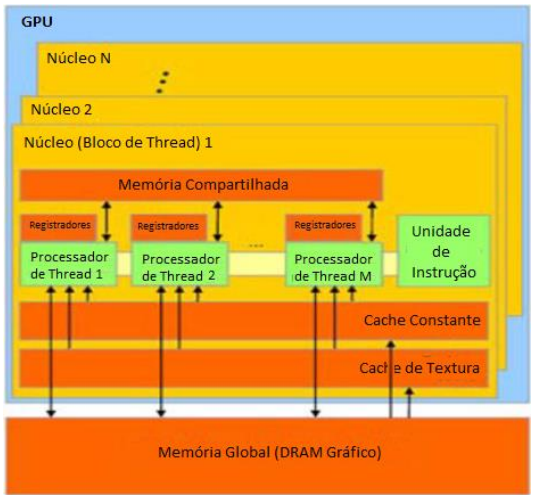


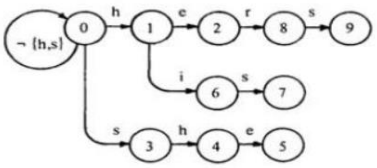
Figura 1: Arquitetura geral de uma GPU [Adaptada de Tran, et al. 2012].

3. Algoritmo Aho-Corasick e sua Implementação em GPUs

Nesta seção é dada uma breve descrição do algoritmo original desenvolvido por Alfred Aho e Margaret Corasick.

O algoritmo invoca três funções [Tran, et al. 2012]: a função *goto* (g), a função de falha (f), e a função de saída (*output*). A Figura 2 mostra as funções utilizadas pela máquina AC para o conjunto de padrões {"he", "she", "his", "hers"}:

- O grafo orientado da Figura 2 (a) representa a função *goto* (onde ¬ ('h', 's') denota todos os outros símbolos de entrada diferentes de 'h' e 's'). A função *goto* mapeia um par consistindo de um estado e um símbolo de entrada dentro de um estado ou uma mensagem de falha. Por exemplo, a borda marcada como h do estado 0 para o estado 1 indica que o g (0, 'h') = 1. A ausência de uma seta indica falha. A máquina AC tem a propriedade que g (0, σ) ≠ falha para qualquer símbolo de entrada σ.
- A função de falha mapeia um estado para outro estado. Ela é consultada sempre que a função *goto* relata uma "falha".
- A função de saída mapeia um conjunto de palavras-chave para a saída de acordo os estados finais alcançados.



(a) A função *goto*

i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

(b) A função de falha

i	$output(i)$
2	{he}
5	{she, he}
7	{his}
9	{hers}

(c) A função de saída

Figura 2: Funções usadas no algoritmo AC [Adaptada de Tran, et al., 2012].

Como exemplo, assuma o texto "ushers". A máquina AC trabalha da seguinte maneira: iniciando com o estado 0, a máquina volta para o estado 0 uma vez que g(0,'u')=0. Pela mesma razão, a máquina entra nos estados 3, 4 e 5 sequencialmente enquanto processa os caracteres 's', 'h' e 'e' (g(0,'s')=3, g(3,'h')=4, g(4,'e')=5) e emite a saída, indicando que ela encontrou as palavras "she" e "he". Depois a máquina avança para o próximo caractere de entrada 'r'. Uma vez que g(5,'r')=falha, a máquina entra no estado 2, já que f(5)=2 (Figura 2b). Então, uma vez que g(2,'r')=8 e g(8,'s')=9, a máquina AC entra no estado 9 e emite a saída "hers".

Para finalizar a discussão sobre o algoritmo AC, Aho e Corasick (1975) afirmam que a função *goto* pode ser armazenada das seguintes formas: em um vetor bidimensional (muitas vezes chamada de *State Transition Table* ou pela sigla STT), onde o acesso é em tempo constante, mas exige mais espaço de armazenamento, ou em uma lista linear, que necessita de menos espaço de armazenamento, porém o tempo de acesso para *g(s,a)* é proporcional ao número de valores que não levam a máquina a uma falha no estado *s*. Eles também sugerem uma solução mista, onde os estados mais frequentemente usados, tal como o estado 0, sejam armazenados na tabela e os menos usados, na lista linear.

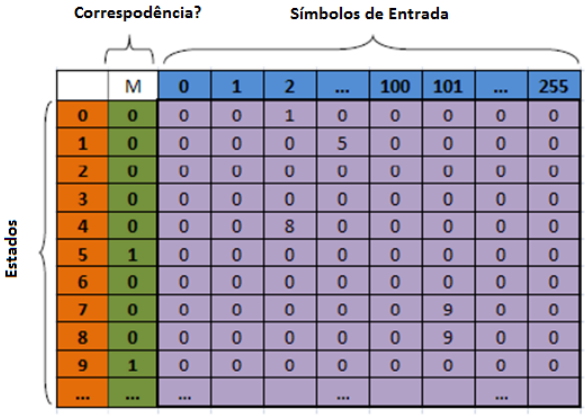
No entanto, em se tratando de GPU, a estrutura de dados utilizada tende a ser o vetor bidimensional, já que o modelo de memória da GPU é restritivo quando se trata de implementar estruturas de dados bem conhecidas, tais como listas ligadas e árvores [Jacob e Brodley 2006].

4. Trabalhos Relacionados

Alguns trabalhos desenvolvidos recentemente são apresentados nesta seção a fim de mostrar como andam as pesquisas na área da paralelização de dados no algoritmo AC.

Tran, N. et al. (2012) apresentaram uma nova técnica de paralelização na qual armazenam de forma eficiente os dados do texto de entrada e os dados de referência (padrões de comparação) nas memórias da GPU. A partir dos dados de referência eles criaram uma STT (Figura 3) e colocaram esses dados na memória de textura de modo que a parte ativamente usada da STT pôde ser armazenada na cache de textura. A abordagem reduziu significativamente as latências médias de acesso à memória para

carregar ambos os dados de entrada e os dados de referência, e levou a melhorias de desempenho para o algoritmo AC, fazendo com que o algoritmo tivesse uma velocidade de processamento de 15,75 vezes maior se comparado com a versão serial em um de seus experimentos.



	M	0	1	2	...	100	101	...	255
0	0	0	0	1	0	0	0	0	0
1	0	0	0	0	5	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	8	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	9	0	0
8	0	0	0	0	0	0	9	0	0
9	1	0	0	0	0	0	0	0	0
...	...	...			...			...	

Figura 3: Ilustração da Tabela de Transição de Estado [Tran et al. 2012].

Lin, C. et al. (2013) criaram o PFAC, uma versão que faz a comparação de *strings* de forma paralela utilizando o algoritmo AC. O PFAC não utiliza as transições de falhas existentes no algoritmo original. Sua STT foi colocada na memória de textura, porém eles afirmaram que como o estado zero é o mais acessado, as transições referentes esse estado foram inseridas na memória compartilhada. O algoritmo alcançou um *speedup* de 74,95 vezes se comparado com a versão serial do AC.

Um ponto interessante a se observar é que nem sempre é possível colocar a STT completa na memória devido à limitação de espaço de armazenamento. LIN et al. (2010), que implementaram a STT na memória compartilhada, dividiram a STT por grupos de ataques. Villa et al. (2012), implementaram na memória de textura, mas se a STT for muito grande ela é lançada por partes.

Além disso, Villa et al. (2012) aproveitaram a própria STT para dizer se o próximo estado era um estado final ou não. Para isso, cada posição da STT tem 32 bits, sendo que os 31 primeiros indicam o próximo estado e o último indica um *flag* de estado final. Dessa forma retira-se a necessidade de construir outra tabela para informar se é um estado final (ou estado de aceitação).

O espaço de armazenamento limitado também fez com que alguns autores compactassem a STT. Tanto Pungila (2013, 2015) como Bellekens (2014) fizeram uma compressão no DFA e usaram a técnica de mapeamento de bits para representá-lo. Pungila (2013) usou comparação de prefixos afirmando que um prefixo de profundidade igual a 8 é suficiente para produzir uma taxa de falso positivo de apenas 0,0001%. Já Pungila (2015) usou o algoritmo de compressão chamado Lempel-Ziv-Welch, conhecido como LZW.

Na técnica de mapeamento de bits cada nó do DFA tem um bitmap associado (um *array* de 8 células de 32 bits cada) representando os 256 valores do alfabeto ASCII. Cada bit no bitmap do nó que é setado como 1 representa uma transição válida para aquele nó. Com isso, os autores não necessitaram mais representar todas as combinações possíveis entre estados e caracteres de entrada, conforme é feito na STT

comum.

5. Compactação da *State Transition Table*

Nesta seção a estrutura da compactação é apresentada juntamente com os passos necessários para compactar uma STT.

Como pôde ser visto na Figura 3 uma STT comum possui poucos valores diferentes de zero. A fim de retirar os zeros desnecessários da STT, a compactação descrita a seguir foi desenvolvida e executada em pacotes de redes reais, mostrando que seu poder de compactação possibilita que milhares de regras sejam armazenadas na memória compartilhada da GPU, aumentando a velocidade da procura por padrões.

A tabela de transição passará a ser representada por três vetores, conforme a explicação seguinte (ver Figura 4):

a) Vetor de Índices (VI): O valor armazenado neste vetor, ou seja, VI[i], significa o índice inicial no vetor VE correspondente ao estado i. O tamanho de VI será igual à quantidade de estados, sendo que o índice i de VI corresponde ao estado i da máquina AC. Além disso, se ∀ caractere de entrada α, o estado i levar a máquina a uma falha, VI[i] é igual a -1. Possui relação 1 para N com VE, sendo que um estado pode ter várias entradas que o levem a outro estado válido, mas uma entrada leva a máquina apenas a um estado.

b) Vetor de Entrada (VE): armazena todas as entradas que levam um estado qualquer para outro estado válido em conformidade com os valores de VI. Possui relação 1 para 1 com VS.

c) Vetor de Saída (VS): armazena o estado de saída devido à entrada VE de mesmo índice.

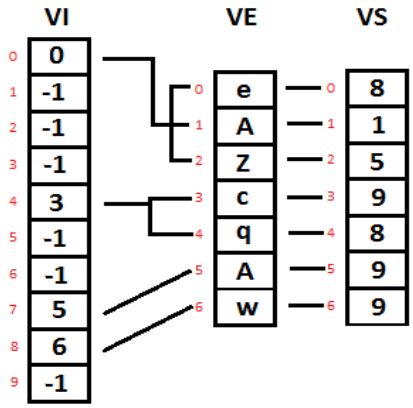


Figura 4: STT compactada

Para exemplificar como a transição de estado ocorre nos vetores de compactação, suponha que entre na máquina AC o caractere ‘q’ e o estado atual seja 4. VI[4] indica que os caracteres de entrada que podem levar a máquina a um estado válido se iniciam no índice 3 de VE. Agora, deve-se obter o valor posterior a VI[4] diferente de -1, que nesse caso é 5 (armazenado em VI[7]). Diminuindo 5 de 3 obtém-se a quantidade de caracteres de entradas que deve ser analisada em VE, ou seja, duas entradas. Portanto, a análise em VE deve ser feita iniciando no índice 3 até o índice 4. Partindo do índice 3

de VE, nota-se que ‘c’ é diferente de ‘q’. Avançando para o índice 4, encontra-se o caractere ‘q’. Como estamos no índice 4 de VE, o valor correspondente em VS, ou seja, VS[4], é o estado 8.

Caso no exemplo anterior em vez de ‘q’ a entrada fosse ‘y', não haveria uma saída correspondente em VS. Neste caso ocorreria uma falha na função *goto* e o vetor de falhas deveria ser consultado para o estado 4.

Como o Snort possui milhares de regras, uma máquina AC real pode chegar a ter milhares de estados, fazendo com que a compactação seja bastante necessária, quer seja para colocar a máquina AC na memória compartilhada, quer seja para colocá-la em memórias maiores que não suportem uma STT tão grande, necessidade que ocorreu nos experimentos de Villa et al. (2012), que apesar de implementarem a STT na memória de textura, tiveram a necessidade de lançá-la por partes caso ela fosse muito grande.

A compactação descrita acima é bem semelhante às Matrizes Esparsas CSR (*Compressed Sparce Row*). VE e VS são idênticos. A diferença está em VI, que nas matrizes esparças

Através das regras *free* do Snort, foi construída uma STT a partir de mais de cinco mil regras, gerando um autômato de 48317 nós. Podemos ver na Figura 5 que a compactação proposta neste artigo reduz significativamente a quantidade de KB necessários para armazenar a STT.

Abordagem	Quantidade de kilobytes necessários
STT comum	24738
Bitmapeamento	1540
Compactação desse trabalho	145

Figura 5: Redução da quantidade de dados através da compactação.

Para construir os vetores VI, VE e VS a partir de uma determinada STT, o seguinte algoritmo deve ser seguido:

1) Declare a variável *contador_de_entradas* e atribua zero a essa variável;
2) Inicie um loop a partir da linha zero da STT;
3) Percorra todas as células da linha *i* da STT da esquerda para a direita;
4) Se o valor da célula for diferente de -1, armazene o valor da célula em *VS*, armazene o valor da coluna *j* em *VE* e incremente a variável *contador_de_entradas*;
5) Depois de percorrer a linha *i* completamente, armazene *contador_de_entradas* em *VI[i]* (caso este contador não tenha sido incrementado para a linha *i*, armazene -1 em *VI[i]*);
6) Vá para a próxima linha da STT;
7) Volte para o passo 2 até que toda STT tenha sido percorrida.

6. Experimentos e Resultados

Esta seção mostrará como os experimentos foram realizados e quais resultados foram obtidos.

Foram utilizadas duas GPUs e uma CPU para executar o algoritmo AC a fim de comparar o desempenho em cada uma das plataformas, conforme as seguintes descrições:

- GPU TITAN X: GPU com 3584 cores CUDA e memória global de 12 GB. Seu host possui um processador Intel Core i5 1,2 GHz da 6ª geração, memória RAM de 8 GB em um sistema operacional Linux 3.19.0-80-generic #88~14.04.1-Ubuntu;
- GPU TESLA K20: GPU com 2496 cores CUDA e memória global de 5 GB. Seu host possui um processador Intel Xeon Ten-Core E5-2660v2 de 2.2 GHz, memória RAM de 64 GB em um sistema operacional RedHat 6.4;
- CPU: Intel Core i3-4005U 1,70 GHz. Opera com uma memória RAM de 4 GB em um sistema operacional Windows 8.1 de 64 bits.

Os dados de entrada foram divididos em pacote sintético e pacote real. O pacote sintético é composto por 1000 caracteres de um texto em inglês. Por ter 1000 caracteres, seu tamanho bruto é de 1 KB. Para simular dados maiores, este texto foi replicado a fim de obter o tamanho desejado. Por exemplo, no caso do experimento de 1MB de texto de entrada, o texto bruto foi replicado 1000 vezes. Cada thread processa um dos textos replicados, ou seja, no caso de 1 MB serão necessárias 1000 threads para processar o texto completo. O pacote real foi formado pelos caracteres dos pacotes obtidos na rede da Universidade Federal de Sergipe, com o auxílio do software Wireshark. Seu tamanho bruto de 1 MB. Esse texto foi dividido em 10 partes de tamanhos iguais para simular a chegada de vários pacotes de dados para serem processados paralelamente. Neste texto também houve replicações para se alcançar o tamanho de texto desejado. Para o experimento de 10 MB, por exemplo, esse texto foi replicado 10 vezes e processado por 100 *threads*.

Algumas configurações de bloco e grid da GPU foram testadas e, para a placa TESLA K20, a melhor configuração foi um bloco de tamanho 100 e grid de tamanho 10000. Já para a placa TITANX, a melhor configuração foi bloco igual a 1000 e grid igual a 1000.

Quatro versões do algoritmo AC foram criadas, sendo uma versão serial e três versões com processamento paralelo. Das versões paralelas, em uma versão a STT é colocados na memória global, em outra versão a STT é colocada na memória de textura e outra onde a STT é compactada com a técnica descrita na Seção 5 e colocada na memória compartilhada da GPU. O pacote de entrada foi colocado na memória global em todas as versões paralelas.

As métricas utilizadas nos experimentos são descritas abaixo:

- Tempo de Execução do Sistema (TE): É a soma do tempo de execução do *kernel* com o tempo de transferência de dados entre CPU e GPU medido em milissegundos;
- *Throughput* (TT): É a taxa de transferência do *kernel* medida em caracteres por segundo (Mcps);
- Percentual de Execução do *Kernel* (% TE): É a porcentagem do Tempo de Execução do Sistema necessária para a execução do kernel.

Pode-se ver na Tabela 1 que as três versões da GPU executaram de forma mais rápida que a CPU em todos os ensaios com pacotes sintéticos na TESLA K20. Nota-se também que entre pacotes de 10 MB e 50 MB a versão de memória compartilhada já consegue superar as outras duas versões paralelas.

Tabela 1: Resultados com pacote sintético - Tesla K20.

Tamanho do Texto de Entrada	Serial		Global			Textura			Compartilhada		
	TE	TT	TE	TT	% TE	TE	TT	% TE	TE	TT	% TE
1 MB	31	32,26	2,5	500,00	80	2,5	500,00	80	2,5	500,00	80
10 MB	250	40,00	9	3333,33	33	9	3333,33	33	9	3333,33	33
50 MB	1328	37,65	68	1470,59	50	67	1515,15	49	62	1785,71	45
125 MB	3453	36,20	171	1453,49	50	171	1453,49	50	155	1785,71	45
250 MB	6750	37,04	346	1461,99	49	342	1461,99	50	307	1838,24	44
500 MB	13297	37,60	684	1461,99	50	699	1461,99	49	613	1845,02	44
1000 MB	26818	37,29	1368	1459,85	50	1371	1461,99	50	1224	1855,29	44

Além disso, nota-se que até 10 MB o GCT (Ganho da memória Compartilhada em relação à memória de Textura) e o GCG (Ganho da memória Compartilhada em relação à memória Global) – ambos calculados através do Tempo de Execução - foram iguais a zero e as abordagens poderiam ser usadas sem distinção. Porém se o texto de entrada tiver um tamanho maior ou igual a 50 MB é válido usar a abordagem de memória compartilhada para ter uma execução mais veloz, podendo chegar a um ganho de velocidade acima de 14% em relação às outras duas abordagens.

Já na placa TITANX o desempenho da abordagem compactada se mostrou ainda mais eficiente. Diferentemente da placa Tesla K20, nesta placa o ganho da versão compartilhada em relação às outras abordagens já começa a aparecer a partir de 1 MB e a partir daí vai aumentando. Com 1 GB de texto de entrada, a memória compartilhada obteve um ganho de desempenho de 73% em relação às versões de memória de textura e memória global. Além disso, em termos de tempo de execução, a versão compactada foi 56 vezes mais rápida que a versão serial quando o tamanho do texto de entrada foi 1 GB, como pode ser visto na Tabela 2.

Tabela 2: Resultados com dados sintéticos - Titanx.

Tamanho do Texto de Entrada	Serial		Global			Textura			Compartilhada		
	TE	TT	TE	TT	% TE	TE	TT	% TE	TE	TT	% TE
1 MB	31	32,26	1,87	729,93	73	1,85	740,74	73	1,52	980,39	67
10 MB	250	40,00	5,35	7407,41	25	5,46	6849,32	27	5,16	8620,69	22
50 MB	1328	37,65	42	2173,91	55	42	2173,91	55	26	7142,86	27
125 MB	3453	36,20	97	2450,98	53	96	2500,00	52	61	8333,33	25
250 MB	6750	37,04	201	2293,58	54	200	2314,81	54	122	8620,69	24
500 MB	13297	37,60	402	2252,25	55	402	2314,81	54	238	8928,57	24
1000 MB	26818	37,29	823	2183,41	56	823	2217,29	55	476	9090,91	23

Ao se usar os pacotes reais o desempenho de todas as versões diminuíram se comparado aos pacotes sintéticos devido aos dados apresentarem uma assimetria maior, fato que é ruim para o sincronismo das *threads* da GPU. Porém, os resultados obtidos continuaram satisfatórios, como pode ser visto no experimento com a placa TESLA K20 na Tabela 3 e com o TITAN X na Tabela 4. Em todos os ensaios as GPUs superaram a CPU. Na placa TESLA K20 a versão compartilhada foi mais veloz que a versão global em todos os ensaios, porém só foi mais rápida que a versão de textura quando o texto de entrada teve um tamanho entre 200MB e 500MB.

Tabela 3: Resultados com dados reais – Tesla K20.

Tamanho do Texto de Entrada	Serial		Global			Textura			Compartilhada		
	TE	TT	TE	TT	% TE	TE	TT	% TE	TE	TT	% TE
10 MB	375	26,67	268	38,31	97	246	41,84	97	265	38,76	97
50 MB	1906	26,23	296	190,84	89	275	208,33	87	293	193,05	88
100 MB	3828	26,12	331	380,23	79	310	414,94	78	327	386,10	79
200 MB	7641	26,17	449	729,93	61	388	793,65	65	399	760,46	66
500 MB	19062	26,23	736	1333,33	51	670	1533,74	49	656	1607,72	47
1000 MB	38329	26,09	1527	1182,03	55	1610	1128,67	55	1506	1283,70	52

Na placa TITANX, a versão compartilhada superou as outras duas versões paralelas em todos os ensaios, chegando a ter um ganho de 12,1% em relação à memória de textura e 14,9% em relação à memória global. Além disso foi 48 vezes mais rápida que sua versão serial.

Tabela 4: Resultados com dados reais – Titanx.

Tamanho do Texto de Entrada	Serial		Global			Textura			Compartilhada		
	TE	TT	TE	TT	% TE	TE	TT	% TE	TE	TT	% TE
10 MB	375	26,67	105	99,01	96	98	106,38	96	92	113,64	96
50 MB	1906	26,23	120	495,05	84	113	531,91	83	108	561,80	82
100 MB	3828	26,12	140	970,87	74	134	1041,67	72	128	1098,90	71
200 MB	7641	26,17	177	1923,08	59	170	2061,86	57	165	2173,91	56
500 MB	19062	26,23	456	1838,24	60	445	1923,08	58	397	2369,67	53
1000 MB	38329	26,09	905	1883,24	59	897	1926,78	58	831	2202,64	55

Os códigos-fontes desenvolvidos neste trabalho podem ser acessados através do link https://gist.github.com/anonymous/4957a9bc119be1c0514f26eca6e63266.

7. Conclusões

Este artigo mostrou que a comparação serial de padrões utilizada em softwares de segurança da informação como o IDS Snort tem se tornado cada vez mais problemática para a CPU devido a grande quantidade de dados para processar. Como uma solução viável, as GPUs fornecem uma estrutura de processamento de dados paralela e uma hierarquia de memórias que fazem aumentar a velocidade do processamento dos dados.

Sendo assim, foram extraídos 18 ataques das regras free do Snort a fim de construir uma STT do algoritmo Aho-Corasick para ser executada nas memórias das GPUs e, particularmente, compactá-la para ser usada na memória compartilhada.

Nos testes com dados sintéticos, as duas placas GPUs utilizadas tiveram ganhos de velocidades consideráveis podendo chegar a um ganho de 73% com a placa TITANX na abordagem compactada na memória compartilhada se comparada com as abordagens não compactadas nas memórias global e de textura. Se comparado com a CPU, o ganho da versão compactada é maior ainda, podendo ser 56 vezes mais rápida. Nos teste com dados reais, as abordagens tiveram ganhos menores (muito devido à aleatoriedade dos dados) se comparados com os dados sintéticos, mas evidenciou-se que mesmo assim os

ganhos são suficientes para justificar o uso da versão compactada, podendo chegar a 14,9% também na placa TITANX. Se comparada com a CPU, a versão compactada executou 48 vezes mais rápida.

Como trabalhos futuros, sugere-se que sejam retirados os conflitos de banco da versão compartilhada, já que isto deve aumentar consideravelmente o desempenho desta versão. Por fim, pode-se também fazer a integração do algoritmo desenvolvido com o Snort. Uma sugestão seria criar dois *buffers* de 200 MB, por exemplo. Quando o primeiro *buffer* estiver com os 200 MB das *strings* onde serão procurados os padrões, ele seria enviado para a GPU processar. Enquanto isso, o segundo *buffer* começaria a ser preenchido e quando estivesse completo (o processamento do primeiro já deveria ter acabado) seria enviado para a GPU e o processo recomeçaria no primeiro buffer.

Referências

Aho, A.; Corasick, M. (1975). "Efficient string matching: an aid to bibliographic search", Communications of the ACM, v.18 n.6, p.333-340, June.

Bellekens, X. J. A.. et al. (2014). "A Highly-Efficient Memory-Compression Scheme for GPU-Accelerated Intrusion Detection Systems". ACM: SIN '14 Proceedings of the 7th International Conference on Security of Information and Networks. [s. L.], p. 302-310. September.

Cuda. "CUDA: Programación Paralela Facilitada". http://www.nvidia.com.br/object/cuda_home_new_br.html.

Jacob, N.; Brodley, C. (2006). "Offloading IDS Computation to the GPU". The 22nd Annual Computer Security Applications Conference, pp 371-380, December.

Jaiswal, M. (2014). "Accelerating Enhanced Boyer-Moore String Matching Algorithm on Multicore GPU for Network Security". International Journal of Computer Applications, Vol. 97 – No. 1, 2014. http://research.ijcaonline.org/volume97/number1/pxc3896934.pdf.

Kouzinopoulos, C. S.; Margaritis, K. G. (2008). "String Matching on a multicore GPU using CUDA". The 13th Panhellenic Conference on Informatics, pp 14-18, September.

Lee, C.; Lin, Y.; Chen, Y. (2015). "A Hybrid CPU/GPU Pattern-Matching Algorithm for Deep Packet Inspection". PLoS ONE 10(10): e0139301, October.

Lin, C. et al. (2013) "Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs". IEEE Transactions on Computers, Vol. 62, pp 1906-1916, October.

Lin, C. et al. (2010). "Accelerating String Matching Using Multi-threaded Algorithm on GPU". 2010 IEEE Global Telecommunications Conference (GLOBECOM 2010), pp 1-5, December.

Pungila, C. (2013). "Hybrid Compression of the Aho-CorasickAutomaton for Static Analysis in Intrusion Detection Systems". Springer. [s. L.], p. 1-10, January.

Pungila, C.; Negru, V. (2015). "Real-Time Hybrid Compression of Pattern Matching Automata for Heterogeneous Signature-Based Intrusion Detection". Springer Link. Berlim, p. 65-74, May.

Pungila, C.; Reja, M.; Negru, V. (2014). “Efficient parallel automata construction for hybrid resource-impelled data-matching”. Future Generation Computer Systems, Vol. 36, pp 31-41, July.

Santos, B. R. (2005). “Detecção de Intrusos Utilizando o Snort”. 83 f. Monografia (Especialização) - Curso de Pós-Graduação Latu Sensu em Administração de Rede Linux, Departamento de Computação, Universidade Federal de Lavras, http://www.ginux.ufla.br/files/mono-BrunoSantos.pdf, September.

Snort Team. (2015). “SNORT Users Manual 2.9.7: The Snort Project”. 265 p, https://www.snort.org/#documents, June.

Thambawita, D. R. V. L. B.; Ragel, R.; Elkaduwe, D. (2014). “To Use Or Not To Use: Graphics Processing Units (GPUs) For Pattern Matching Algorithms”. The 7th International Conference on Information and Automation for Sustainability (ICIAfS), pp 1-4.

Tran, N.; Lee, M.; Hong, S.; Shin, M. (2012). “Memory Efficient Pararellelization for Aho-Corasick Algorithm on a GPU”. The 14th International Conference on High Performance Computing and Communications, pp 432-438.

Villa, O.; Chavarría-miranda, D. G.; Tumeo, A. (2012). “Aho-Corasick String Matching on Shared and Distributed-Memory Parallel Architectures”. IEEE Transactions on Parallel and Distributed Systems, Vol. 23, pp 436-443.

Execução Energeticamente Eficiente de Aplicações Estêncil com o Processador *Manycore* MPPA-256

Emmanuel Podestá Jr.[1], **Alyson D. Pereira**[1], **Rodrigo C. O. Rocha**[2], **Márcio Castro**[1], **Luís F. W. Góes**[2]

[1] Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD)
Universidade Federal de Santa Catarina (UFSC) – SC, Brasil

[2] Grupo de Computação Criativa e Paralela (CreaPar)
Pontifícia Universidade Católica de Minas Gerais (PUC Minas) – MG, Brasil

emmanuel.podesta@grad.ufsc.br, alyson.pereira@posgrad.ufsc.br,
rcor@pucminas.br, marcio.castro@ufsc.br, lfwgoes@pucminas.br

Resumo. *Neste artigo é proposta uma adaptação do framework PSkel para o processador manycore de baixa potência MPPA-256. O framework permite simplificar o desenvolvimento de aplicações estêncil iterativas para o MPPA-256, escondendo do desenvolvedor detalhes de implementação. Os resultados obtidos no MPPA-256 mostraram uma redução do consumo de energia de aplicações estêncil iterativas de até 1.45x em comparação com um processador multicore Intel Broadwell.*

1. Introdução

Plataformas de Computação de Alto Desempenho (CAD) tem sido avaliadas quase que exclusivamente pela suas capacidades de processamento. Contudo, o consumo excessivo de energia é uma barreira para o aumento de desempenho de forma escalável nestas plataformas. Por essa razão, o estudo de técnicas que melhorem a eficiência energética em plataformas de CAD está se tornando muito importante. Recentemente, uma nova classe de processadores *manycore* de baixa potência tais como o Sunway SW26010 [Fu et al. 2016] e o Kalray MPPA-256 [Castro et al. 2013] foram desenvolvidos. Esses processadores possuem centenas de núcleos de processamento capazes de lidar com paralelismo de dados e tarefas com baixo consumo de energia.

Processadores *manycore* de baixa potência (*low-power manycore processors*) apresentam uma melhor eficiência energética em comparação com processadores de propósito geral presentes atualmente [Francesquini et al. 2014], contudo as suas características arquiteturais tornam o desenvolvimento de aplicações uma tarefa desafiadora [Varghese et al. 2014, Castro et al. 2016, Castro et al. 2014]. Geralmente, núcleos de processamento sem coerência de *cache* são distribuídos em uma arquitetura organizada em *clusters*, onde cada *cluster* possui uma memória local (compartilhada somente entre os núcleos do *cluster*). Dessa forma, a comunicação entre *clusters* deve que ser efetuada através de uma *Network-on-Chip* (NoC) de maneira distribuída. Por essa razão, o tempo de comunicação pode variar entre os núcleos que estão se comunicando.

Uma possível abordagem para facilitar o desenvolvimento de aplicações paralelas para processadores *manycore* de baixa potência é através do uso de abstrações de mais alto nível fornecidas por padrões paralelos ou esqueletos algorítmicos [Cole 2004]. Esses padrões permitem que desenvolvedores foquem na construção de algoritmos, sem a preocupação com problemas de sincronização ou escalonamento de tarefas. Esses problemas são resolvidos de forma transparente pelo *framework* do padrão adotado.

Dentre os diversos padrões paralelos existentes (e.g., *map*, *reduce*, *pipeline* e *scan*), o padrão estêncil tem sido muito utilizado em várias áreas importantes, como física quântica, previsão do tempo e processamento de imagens [Gonzalez and Woods 2006, Holewinski et al. 2012, Lutz et al. 2013]. No padrão estêncil, para cada elemento de uma estrutura n-dimensional de entrada é computado um novo valor para o respectivo elemento em uma estrutura n-dimensional de saída, utilizando-se como base os valores dos elementos vizinhos ao elemento de entrada. A quantidade de vizinhos e a computação a ser realizada em cada elemento é definida por uma função (ou *kernel*) estêncil. Em aplicações estêncil iterativas, os valores produzidos na estrutura n-dimensional de saída em uma iteração i são utilizados como entrada da iteração $i + 1$.

Alguns *frameworks* foram propostos para o desenvolvimento de aplicações paralelas com base no padrão estêncil, tais como SkelCL [Steuwer et al. 2011], SkePU [Enmyren and Kessler 2010] e PSkel [Pereira et al. 2015]. Em especial, o *framework* PSkel provê uma abstração de alto nível para o desenvolvimento de aplicações estêncil em ambientes heterogêneos compostos por processadores *multicore* e *Graphical Processing Units* (GPUs). Todavia, nenhum desses *frameworks* possui suporte para processadores *manycore* de baixo potência de energia emergentes tais como o MPPA-256.

Portanto, nesse artigo é proposta uma adaptação completa do *framework* PSkel para o processador MPPA-256, a qual permite simplificar significativamente o desenvolvimento de aplicações estêncil nesse processador. A adaptação permite eliminar as dificuldades de desenvolvimento intrínsecas do processador, fazendo com que as aplicações já implementadas em PSkel possam ser executadas no MPPA-256 sem a necessidade de nenhuma modificação em seus códigos. Os resultados obtidos mostram que o MPPA-256 apresenta uma melhor eficiência energética que um processador Intel Broadwell com 10 núcleos físicos ao executar três aplicações estêncil implementadas no PSkel.

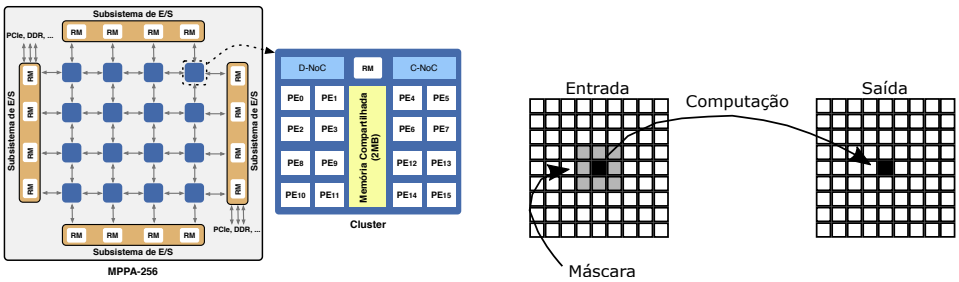
O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta os principais conceitos do processador *manycore* MPPA-256 e do *framework* PSkel. A Seção 3 discute a adaptação do *framework* PSkel para fornecer suporte ao MPPA-256. Os resultados obtidos com a adaptação do *framework* PSkel para o MPPA-256 são apresentados na Seção 4. Por fim, a Seção 5 apresenta os trabalhos relacionados ao tema abordado por esse artigo e Seção 6 apresenta as conclusões deste trabalho.

2. Fundamentação Teórica

2.1. MPPA-256

O MPPA-256 é um processador *manycore* desenvolvido pela empresa francesa Kalray, o qual possui 256 núcleos de processamento de 400 MHz denominados *Processing Element*s (PEs). Além dos PEs, o processador possui 32 núcleos dedicados a gerência de recursos denominados *Resource Managers* (RMs). PEs e RMs são distribuídos fisicamente no *chip* em 16 *clusters* e 4 subsistemas de Entrada/Saída (E/S), contendo cada *cluster* 16 PEs e 1 RM. Além dos *clusters*, o MPPA-256 possui 4 subsistemas de E/S contendo, cada um, 4 RMs. Toda a comunicação entre *clusters* e/ou subsistemas de E/S é feita através de uma NoC *torus* 2D. A arquitetura do MPPA-256 pode ser vista na Figura 1a.

A finalidade principal dos PEs é executar *threads* de usuário de forma ininterrupta e não preemptível para realização de computação. PEs de um mesmo *cluster* compartilham uma memória de 2 MB, a qual é utilizada para armazenar os dados a serem processados pelos PEs. Cada PE possui também uma memória *cache* associativa 2-*way* de 32KB para dados e uma para instruções. Porém, o processador não dispõe de coerência



(a) MPPA-256. (b) O padrão estêncil.

Figura 1. Visão geral do MPPA-256 (esquerda) e uma ilustração do padrão estêncil oferecido pelo PSkel (direita).

de *caches*, o que dificulta o desenvolvimento de aplicações para esse processador. Por outro lado, a finalidade dos RMs é gerenciar E/S, controlar comunicações entre *clusters* e/ou subsistemas de E/S e realizar comunicação com uma memória RAM. Na arquitetura utilizada neste artigo, um dos subsistemas de E/S está conectado a uma memória externa *Low Power Double Data Rate 3* (LPDDR3) de 2 GB.

Trabalhos anteriores mostraram que desenvolver aplicações paralelas otimizadas para o MPPA-256 é um grande desafio [Francesquini et al. 2014] devido a alguns fatores importantes. O primeiro deles está relacionado ao **modelo de programação híbrido** exigido pelo processador: *threads* em um mesmo *cluster* se comunicam através de uma memória compartilhada local, porém a comunicação entre *clusters* é feita explicitamente via NoC, em um modelo de memória distribuída. Mais especificamente, aplicações desenvolvidas para o MPPA-256 precisam utilizar duas bibliotecas de programação paralela para utilizar os recursos do processador: OpenMP, baseado em um modelo de memória compartilhada, utilizada para paralelizar a computação dentro de cada *cluster* e uma *Application Programming Interface* (API) proprietária, que segue um modelo de memória distribuída, sendo utilizado na comunicação entre os *clusters* e o subsistema de E/S por meio da NoC. O segundo fator importante está relacionado a **capacidade limitada de memória** no *chip*: cada *cluster* possui apenas 2 MB de memória local de baixa latência. Portanto, aplicações reais precisam constantemente realizar comunicações com o subsistema de Entrada e Saída (E/S) conectado à memória LPDDR3. Por fim, o último fator está diretamente relacionado à **ausência de coerência de *cache***: cada PE possui uma memória *cache* privada sem coerência de *cache*, sendo necessário o uso explícito de instruções do tipo *flush* para atualizar a *cache* de um PE quando necessário.

2.2. PSkel

O PSkel é um *framework* de programação em alto nível para o padrão estêncil, baseado no conceito de esqueletos paralelos. Ele oferece suporte à execuções paralelas em arquiteturas heterogêneas incluindo CPU e GPU. Utilizando uma única interface de programação escrita em C++, o usuário é responsável por definir o *kernel* principal da computação estêncil, enquanto o *framework* se encarrega de gerar código executável para as diferentes plataformas paralelas, realizando todo o gerenciamento de memória e transferência de dados entre dispositivos de forma transparente [Pereira et al. 2015].

A API do PSkel possibilita a definição de *templates* para a manipulação de estruturas n-dimensionais, denominadas `Array` (1 dimensão), `Array2D` (2 dimensões) e `Array3D` (3 dimensões). Além disso, o *framework* provê abstrações para a definição da

vizinhança do estêncil (`Mask`) e o *kernel* da computação estêncil (`stencilKernel()`). O `stencilKernel()` é um método a ser implementado pelo usuário que descreve, especificamente, a computação que será executada para cada célula do `Array` de entrada com base nos valores de sua vizinhança (`Mask`).

Em uma aplicação estêncil iterativa, cada iteração utiliza a máscara de vizinhança (`Mask`) sobre o `Array` de entrada para determinar o valor de cada célula do `Array` de saída. No exemplo da Figura 1b, o valor de cada célula do `Array` de saída é determinado em função dos valores de cada uma das células vizinhas adjacentes. Esse processo é realizado para todas as células do `Array` de entrada, produzindo um `Array` de saída da computação estêncil. Ao final de uma iteração, o `Array` de saída será considerado como `Array` de entrada para a próxima iteração no caso de uma aplicação estêncil iterativa.

3. Adaptação do *framework* PSkel para o MPPA-256

A adaptação do *framework* PSkel para o processador MPPA-256 proposta neste artigo segue um modelo mestre/escravo. Um processo mestre é executado no subsistema de E/S conectado à memória LPDDR3 de 2 GB, sendo responsável por alocar o `Array` de entrada e por distribuir os dados entre os processos escravos. Em cada *cluster* é instanciado um único processo escravo que é responsável por gerenciar a computação no seu *cluster*. Devido às limitações de memória dos *clusters* (apenas 2 MB por *cluster*), o processo mestre deve subdividir o `Array` de entrada em blocos denominados *tiles* e, então, gerenciar as comunicações dos mesmos com os processos escravos.

O processo mestre particiona o `Array` de entrada com dimensão n em b blocos, onde b é o número de *clusters* utilizados na computação. Então, cada bloco é particionado em *tiles* de tamanho fixo definidos pelo usuário. Quando são feitas computações estêncil sobre o *tile*, dependências de vizinhança, inerentes ao padrão paralelo do estêncil, precisam ser consideradas durante o particionamento dos dados. Uma das principais soluções para satisfazer essas dependências é via blocos sobrepostos, resultando em dados redundantes e computação por *tile* [Meng and Skadron 2011, Holewinski et al. 2012, Rocha et al. 2017]. Essa técnica é muito importante em *manycores* de baixa potência como o MPPA-256, onde o sobrecusto de comunicação pode ser elevado. O impacto dos custos de comunicação será analisado posteriormente na Seção 4.

Portanto, foi implementada uma técnica de *tiling* trapezoidal. Para ilustrar e detalhar como essa técnica de *tiling* pode ser aplicada na computação estêncil, utilizamos a definição a seguir. Seja A um `Array2D`, com dimensões $\dim(A) = (w, h)$, onde w e h são, respectivamente, a largura e a altura. Utilizando *tiles* de dimensões (w', h') produz $\lceil \frac{w}{w'} \rceil \lceil \frac{h}{h'} \rceil$ *tiles* possíveis de A. Seja $A_{i,j}$ um *tile*, onde $0 \leq i < \lceil \frac{w}{w'} \rceil$ e $0 \leq j < \lceil \frac{h}{h'} \rceil$. $A_{i,j}$ possui *offset* (iw', jh') relativo ao canto superior esquerdo de A e $\dim(A_{i,j}) = (\min\{w', w - iw'\}, \min\{h', h - jh'\})$. O *offset* é uma indexação de deslocamento necessário para acessar os elementos do *tile* (Figura 2). Essa técnica pode ser facilmente estendida para mais dimensões.

Aplicar um estêncil em A envolve aplicar a função de vizinhança (máscara) contendo o deslocamento de cada vizinho de um dado elemento central. Por causa da dependência entre vizinhos, para computar a função estêncil, de acordo com as limitações necessárias pelos *tiles*, se torna necessário obter valores de *tiles* adjacentes. Seja r o *range* da máscara de vizinhos, i.e., r é o deslocamento mais distante necessário para a vizinhança definida pela máscara. A área de r envolvendo a vizinhança é denominada região *halo*. Se a função estêncil é aplicada iterativamente sobre A, para t iterações,

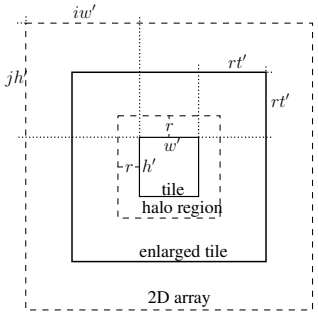


Figura 2. Diagrama do *tiling* 2D. Um *tile* lógico (linha interna sólida) é contido dentro do Array 2D (linha externa pontilhada) com *offsets* verticais e horizontais dado por jh' e iw'. Computar t' consecutivas iterações estêncil no *tile* requer um aumento no *tile* lógico com uma *ghost zone* (área entre a linha interna sólida e a linha externa sólida), que é constituída de regiões *halo* (área entre a linha interna sólida e a linha interna pontilhada).

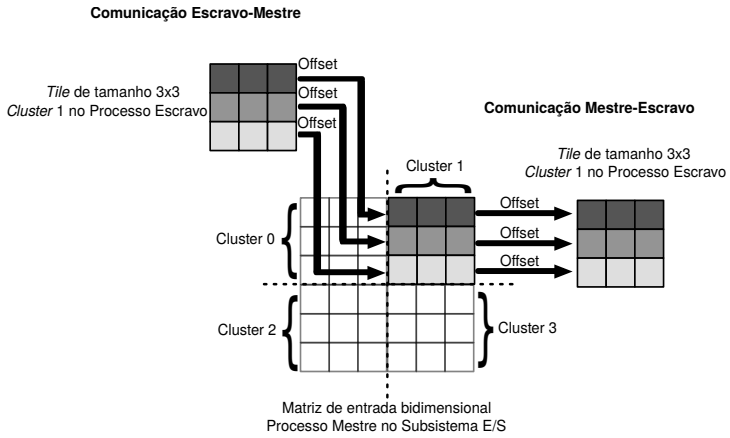


Figura 3. Exemplo do funcionamento do método *strides* do MPPA-256.

a dependência da vizinhança entre os *tiles* limita o número de iterações que podem ser computadas consecutivamente sem a necessidade de realizar comunicações entre *tiles*.

Além disso, devido à restrição da API e NoC no MPPA-256, os dados armazenados em cada *tile* precisam ser contíguos para serem transferidos pela NoC. A fim de se evitar cópias locais de dados, o que desperdiçaria memória e tempo de processamento, utiliza-se o conceito de comunicação por *strides*. Cada *stride* é uma parte contígua do `Array` original, sendo determinado por deslocamentos (*offsets*) especificados durante a execução. Então, cada *stride* é enviado para o `Array` de entrada em um processo escravo na posição determinada de acordo com o tamanho dos *tiles* e do `Array` original. O método de *strides* possibilita a definição de algumas variáveis para o gerenciamento do `Array`, sendo a mais importante os *offsets* que serão utilizados. A partir deles, o método irá efetuar a comunicação de maneira direta para o`Array` destino. Como dito anteriormente, a utilização de *tiles* aumentados permite reduzir a quantidade de comunicações necessárias entre os processos mestre e escravos. Fazendo o aumento dos *tiles* em uma dimensão temporal, os processos escravos podem executar múltiplas iterações sem a necessidade de comunicação ou sincronização com o processo mestre.

O escalonamento dos *tiles* nos *clusters* é feito de maneira circular (*round-robin*). Devido a isso, alguns *clusters* podem receber mais *tiles* que outros, dependendo do número de *tiles* e *clusters* usados na computação. Toda a comunicação entre o mestre e os escravos é feita utilizando-se a API de comunicação assíncrona oferecida pelo processador MPPA-256. A comunicação com cada processo escravo é feita de forma individual, mapeando diretamente áreas contíguas de memória de seus respectivos *tiles*. Além disso, a implementação atual permite a execução de aplicações estêncil iterativas. Nesse caso, o escalonamento dos *tiles* e a computação dos mesmos pelos *clusters* é repetida a cada iteração da aplicação.

Cada processo escravo realiza a computação do *tile* recebido no *cluster* utilizando o *kernel* de computação estêncil definido pelo usuário. A paralelização da computação dentro do *cluster* é feita com auxílio da API OpenMP. Em cada *cluster* podem ser criadas até 16 *threads* (uma para cada PE), onde cada uma é responsável por executar o *kernel* estêncil em um subconjunto de elementos dos *tiles*. Quando a computação do *kernel* estêncil é finalizada, os *tiles* resultantes são enviados pelos processos escravos para o processo mestre, onde são agrupados em um único `Array`, constituindo o resultado final da computação estêncil em uma iteração. Tendo em vista que os *tiles* são regiões contíguas na memória, processos escravos precisam gerenciar os *offsets* de dados para escrevê-los nas posições corretas no `Array` no processo mestre. Esse gerenciamento é efetuado pela API do MPPA-256, mais especificamente, pelo método de *strides* mencionado anteriormente. A Figura 3 ilustra esse procedimento para o caso de um `Array2D`.

Para fornecer uma maior facilidade ao usuário, todas as tarefas complexas relacionadas com a técnica de *tiling*, comunicações NoC e adaptações discutidas nessa seção são abstraídas, pois elas são incluídas no *back-end* do PSkel. Isso significa que aplicações desenvolvidas com o *framework* PSkel podem executar no MPPA-256 sem a necessidade de modificações no código fonte.

4. Resultados Experimentais

Nesta seção será avaliado o desempenho e o consumo energético de aplicações estêncil do PSkel quando executadas no MPPA-256 e em um processador Intel Xeon E5-2640 v4 com 10 núcleos de 2.4GHz (Broadwell). As medições de energia no MPPA-256 foram feitas considerando todos os *clusters*, a memória, os subsistemas de E/S e a NoC, onde foram coletadas por meio dos sensores de potência e energia disponíveis no MPPA-256. Como o MPPA-256 possui características intrínsecas do próprio processador que garante baixa variabilidade entre as execuções, foram realizadas somente 5 repetições de cada experimento, computando-se a média aritmética dos valores. Todos os experimentos no MPPA-256 consideraram 16 PEs por *cluster*. Por outro lado, as medições de consumo de energia foram feitas no processador Intel com uso do *Running Average Power Limit* (RAPL) através da biblioteca PAPI [Weaver et al. 2012]. Em cada experimento foram utilizadas 10 *threads* (uma *thread* por núcleo) sem uso de *hyperthreading*. Cada experimento foi repetido 30 vezes e a média aritmética dos resultados foi calculada. Todos os resultados (MPPA-256 e Intel) apresentaram um desvio-padrão menor que 1%.

É possível reduzir a quantidade de sincronizações e comunicações realizadas na solução para o MPPA-256 de duas maneiras: aumentando o tamanho dos *tiles* ou aumentando a quantidade de iterações sobre cada *tile*. Como podemos ver na Figura 2, ao aumentarmos a quantidade de iterações sobre o *tile*, precisamos aumentar o *tile* lógico, formando um *tile* aumentado que será enviado para o *cluster*. Desta forma, devido às limitações de memória em cada *cluster*, ao ser especificado uma quantidade muito

grande de iterações, o *tile* aumentado enviado para o escravo pode ser maior que o limite de memória de cada *cluster*. Portanto, foi fixado uma quantidade de 10 iterações sobre cada *tile*. Além disso, para os experimentos terem uma quantidade significativa de sincronizações sobre aplicações estêncil iterativas, foi adotado 30 iterações para cada aplicação.

4.1. Aplicações Estêncil

Para a realização dos experimentos foram utilizadas as seguintes aplicações estêncil:

Fur: modela a formação de padrões sobre a pele de animais[1]. Nessa aplicação, a pele do animal é modelada por uma *array* bidimensional de células de pigmento que podem estar em um dos dois estados: colorida ou não-colorida. As células coloridas secretam ativadores e inibidores. Ativadores fazem uma célula central se tornar colorida; inibidores, por outro lado, fazem uma célula central se tornar não colorida. A diferença entre as potências dos ativadores e inibidores é responsável por decidir a coloração da célula central, onde mais ativadores resulta em uma célula colorida e mais inibidores resulta em uma célula não colorida. Nos casos em que as potências dos ativadores e inibidores forem iguais, a cor da célula permanece inalterada. A máscara contém células adjacentes à célula central e seu tamanho é parametrizável. Neste trabalho foi utilizado 2 vizinhos adjacentes em cada direção.

Jacobi: método iterativo para resolver equações matriciais [Demmel 1997]. O método converge garantidamente se a matriz de entrada é restrita ou irredutívelmente dominante diagonalmente, i.e., $|u_{i,i}| > \sum_{j \neq i} |u_{i,j}|$, para todo i. A Equação 1 define a computação em cada passo do método iterativo de Jacobi para resolver a equação discreta elíptica de Poisson [Demmel 1997]. A solução aproximada é computada discretizando o problema na matriz em pontos espaçados de forma equivalente por $n \times n$.

$$u'_{i,j} = \frac{u_{i\pm1,j} + u_{i,j\pm1} + h^2 f_{i,j}}{4} \tag{1}$$

A cada passo, o novo valor de $u_{i,j}$ é obtida fazendo a média $h^2 f_{i,j}$ dos seus vizinhos, onde $h = \frac{1}{n+1}$ e $f_{i,j} = f(ih, jh)$, para uma dada função f.

GoL: autômato celular que implementa o Jogo da Vida de Conway [Gardner 1970]. O autômato é representado por um *array* bidimensional, onde cada elemento representa um indivíduo vivo ou um indivíduo morto. A máscara do estêncil, a qual determina a interação entre o indivíduo e seus vizinhos, considera as 8 células vizinhas adjacentes à célula central. Dependendo dos valores dos vizinhos, o elemento pode modificar seu estado entre vivo e morto.

4.2. Impacto do Tamanho do *Tile* no Desempenho do MPPA-256

O primeiro experimento tem por objetivo verificar o impacto do tamanho dos *tiles* no desempenho e consumo energético das aplicações. As Figuras 4 e 5 mostram, respectivamente, os tempos de execução e consumo de energia de três aplicações estêncil, variando-se o tamanho do `Array2D` de entrada (de 2048^2 até 12288^2) e os tamanhos do *tile* (de 32^2 até 128^2). *Arrays* de entrada maiores que 12288^2 e *tiles* maiores que 128^2 extrapolam às memórias LPDDR3 e dos *clusters*, respectivamente.

[1] http://ccl.northwestern.edu/netlogo/models/Fur

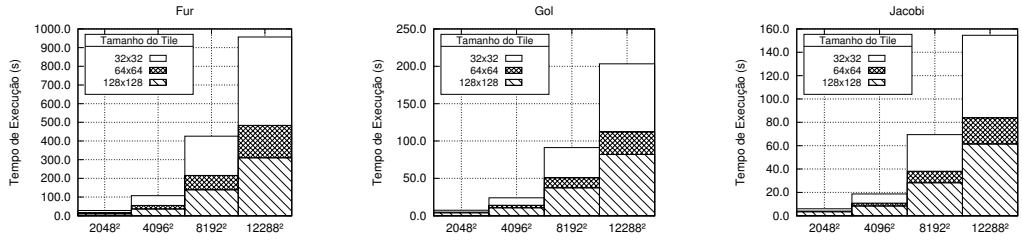


Figura 4. Tempos de execução das aplicações para diferentes tamanhos de *tile* e `Array2D` no MPPA-256.

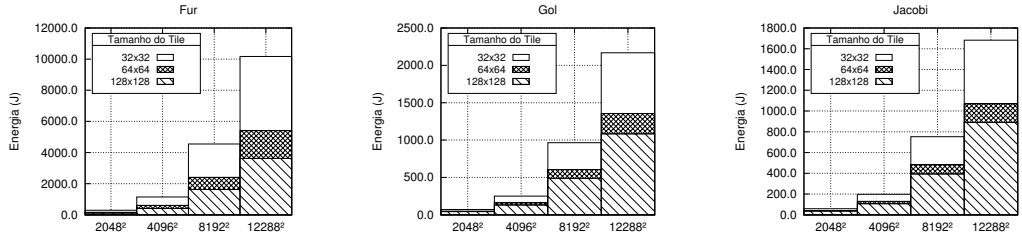


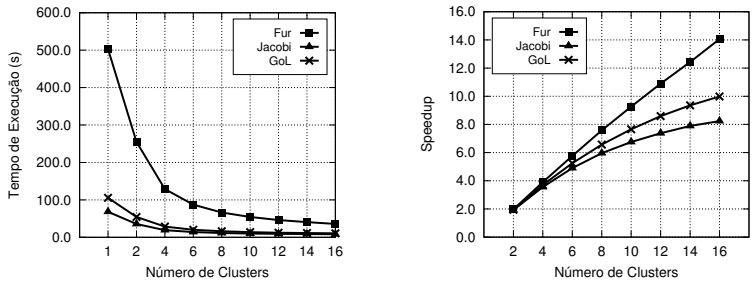
Figura 5. Consumo de energia das aplicações para diferentes tamanhos de *tile* e `Array2D` no MPPA-256.

Pode-se perceber uma redução no tempo de execução à medida em que se aumenta o tamanho do *tile* (Figura 4), pois há menos sincronizações e comunicações de *tiles* entre os processos mestre e escravos. O comportamento das aplicações é similar, sendo diferenciado apenas pela grandeza dos tempos de execução. A Figura 5 apresenta um comportamento similar para o consumo de energia, pois o tempo de execução reduz com o aumento do tamanho do *tile*, trazendo uma redução no consumo de energia.

4.3. Análise de Escalabilidade no MPPA-256

Em um segundo experimento, buscou-se verificar a escalabilidade das aplicações no MPPA-256. Para isso, variou-se o número de *clusters* em cada aplicação, com `Array2D` de entrada fixo de tamanho 4096^2 e *tiles* de tamanho 128^2. A Figura 6(a) apresenta os tempos de execução obtidos ao variar-se o número de *clusters* utilizados na computação. A Figura 6(b), por outro lado, apresenta o fator de aceleração (*speedup*) com relação ao tempo de execução com 1 *cluster*. Em outras palavras, o *speedup* com c *clusters* é computado dividindo-se o tempo de execução obtido com apenas 1 *cluster* pelo tempo de execução obtido com c *clusters*.

No geral, os resultados mostraram que a solução proposta para o MPPA-256 é escalável. Porém, pode-se notar que a aplicação *Fur* apresentou uma escalabilidade superior às demais aplicações. Esse comportamento está diretamente relacionado com a quantidade de operações realizadas pelo *kernel* da aplicação (complexidade do *kernel*). Tendo em vista a necessidade de comunicações no MPPA-256, o tempo total de execução de uma aplicação passa a ser composto pela soma do tempo de comunicação com o tempo de computação. Para um dado *tile* t de tamanho fixo, o tempo necessário para realizar comunicações de t entre mestre e escravo será constante. Por outro lado, quanto maior o número de operações (computações) feitas em t pelo *kernel* da aplicação, maior será o paralelismo a ser explorado. Nesse caso, o tempo de computação será proporcionalmente maior que o tempo de comunicação, melhorando assim a escalabilidade obtida. Este é o caso da aplicação *Fur* cujo *speedup* se aproxima do caso ideal. Em contrapartida,



(a) Tempo de execução. (b) Speedup.

Figura 6. Resultados de tempo e *speedup* das aplicações *Fur*, *GoL* e *Jacobi*.

aplicação *Jacobi* apresentou uma escalabilidade mais baixa que as demais, pois seu *kernel* apresenta baixa complexidade.

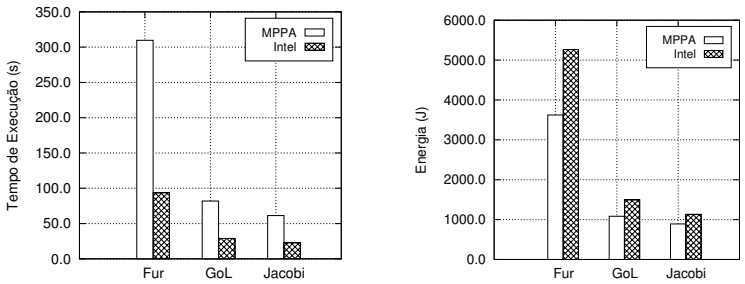
4.4. Kalray MPPA-256 vs. Intel Broadwell

Por fim, foram efetuados experimentos comparativos entre o processador Intel Xeon e o MPPA-256, como pode-se ver na Figura 7. Nesses experimentos, utilizou-se um `Array2D` de entrada de tamanho 12288^2 e *tiles* de tamanho 128^2. Ao ser comparado o tempo das aplicações em cada arquitetura, nota-se que o MPPA-256 tem um desempenho pior, contudo ao ser comparado o consumo de energia percebe-se um comportamento diferente: a energia consumida pelo MPPA-256 é menor, principalmente na aplicação *Fur*. No geral, o consumo de energia das aplicações *Fur*, *GoL* e *Jacobi* no MPPA-256 foi aproximadamente 1.45x, 1.38x e 1.27x menor que no Intel Xeon, respectivamente. Por outro lado, o tempo de execução dessas aplicações obtido no MPPA-256 foi 3.30x, 2.83x e 2.69x maior que no Intel Xeon, respectivamente.

5. Trabalhos Relacionados

Alguns trabalhos surgiram recentemente com intuito de avaliar o uso de processadores *manycore* em CAD, além de discutir os desafios do desenvolvimento de aplicações para esses processadores. Em [Totoni et al. 2012], os autores compararam o desempenho e o consumo energético de um processador *manycore* experimental da Intel denominado *Single-Chip Cloud Computer* (SCC) com outros tipos de processadores e GPUs. Para realizar essa análise, os autores utilizaram um conjunto de aplicações paralelas implementadas em Charm++ [Kale and Bhatele 2012]. Os resultados obtidos com o Intel SCC mostraram que *manycores* são uma alternativa viável, apresentando bom desempenho e baixo consumo energético. Em [Sirdey et al. 2013], os autores avaliaram o desempenho do processador *manycore* MPPA-256 no contexto de aplicações de decodificação de vídeo. Os resultados mostraram que o desempenho do MPPA-256 é comparável ao desempenho de processadores Intel atuais em uma decodificação de vídeo no padrão H.264, consumindo 6 vezes menos energia.

Trabalhos recentes revelaram o desempenho e consumo energético do processador MPPA-256, comparando-o a outros processadores *multicore* de propósito geral e embarcados, no contexto de diferentes aplicações científicas [Castro et al. 2014, Castro et al. 2013, Francesquini et al. 2014]. Os resultados mostraram que o processador *manycore* MPPA-256 apresenta em alguns casos desempenho superior a processadores *multicore* Intel Xeon 2.4 GHz com 8 *cores*, além de um consumo de energia de até 13 vezes menor em relação ao mesmo processador. Um outro trabalho recentemente publicado



(a) Tempo de execução. (b) Consumo de Energia.

Figura 7. Comparação do tempo de execução e consumo de energia das aplicações *Fur*, *GoL* e *Jacobi* em relação a arquitetura.

realizou uma análise comparativa de desempenho e consumo de energia entre processadores *multicore* Intel de alto desempenho e ARM [Padoin et al. 2015]. Os resultados mostraram que, apesar da potência dos processadores ARM ser pelo menos 10 vezes menor que a dos processadores Intel de alto desempenho, o consumo de energia nem sempre será melhor, sendo dependente das características da carga de trabalho a ser executada.

Trabalhos recentes que propuseram APIs e ambientes de execução para *manycores*. Em [Lam et al. 2013], os autores propuseram uma adaptação Espaço de Endereçamento Global Particionado (*Partitioned Global Address Space* – PGAS) para simplificar o desenvolvimento de aplicações paralelas para os processadores *manycore* TILE-Gx e TILEPro. Mais precisamente, os autores utilizaram a biblioteca OpenSHMEM como base para a proposta, utilizando-a como uma camada de abstração para as bibliotecas fornecidas pelo fabricante dos processadores. Com isso, aplicações atualmente implementadas utilizando a API OpenSHMEM podem ser executadas nos processadores *manycore* da linha TILE sem que haja a necessidade de modificações no código.

6. Conclusão

O desenvolvimento de aplicações que exploram o paralelismo em processadores *manycore* de baixa potência, tais como o MPPA-256, tornou-se muito importante, tendo em vista o aumento do consumo de energia de processadores de alto desempenho. Porém, o desenvolvimento de aplicações otimizadas nesses processadores é bastante desafiador devido a fatores importantes tais como a existência de um modelo de programação híbrido, capacidade limitada de memória no *chip*, ausência de coerência de *cache*, entre outros.

Neste artigo foi proposta uma adaptação de um *framework* para desenvolvimento de aplicações estêncil iterativas, denominado PSkel, para processador MPPA-256. A solução proposta permite esconder detalhes de baixo nível do MPPA-256, simplificando significativamente o desenvolvimento de aplicações estêncil nesse processador. Os resultados mostraram que a solução proposta apresenta boa escalabilidade. Além disso, foi observado uma redução significativa no tempo de execução e no consumo de energia das aplicações no MPPA-256 ao se utilizar a técnica de *tiling* trapezoidal. Isso se deve, principalmente, à redução do sobrecusto de comunicações e sincronizações de *tiles*.

A aplicação *Fur* apresentou os melhores resultados de escalabilidade dentre as 3 aplicações estudas, obtendo um *speedup* de 14x em relação à apenas um *cluster*. Analisando experimentos executados sobre a adaptação pôde-se perceber uma relação entre a quantidade de computação realizada pelo *kernel* da aplicação e o *speedup* obtido.

Por fim, experimentos comparativos entre o MPPA-256 e o processador Intel Broadwell mostraram que a solução proposta para o MPPA-256 apresenta uma eficiência energética superior apesar de um tempo de execução superior.

Como trabalhos futuros, pretende-se estudar formas de reduzir ainda mais os sobrecustos de comunicação através do uso de técnicas de *sofware prefetching*. Além disso, pretende-se realizar experimentos com outros *benchmarks* e aplicações que utilizam estruturas tridimensionais. Por fim, pretende-se realizar comparações de desempenho e consumo de energia com outros processadores embarcados.

Referências

Castro, M., Dupros, F., Francesquini, E., Méhaut, J.-F., and Navaux, P. O. A. (2014). Energy efficient seismic wave propagation simulation on a low-power manycore processor. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 57–64, Paris, France. IEEE Computer Society.

Castro, M., Francesquini, E., Dupros, F., Aochi, H., Navaux, P. O., and Méhaut, J.-F. (2016). Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Computing*, 54:108–120.

Castro, M., Francesquini, E., Nguélé, T. M., and Méhaut, J.-F. (2013). Analysis of computing and energy performance of multicore, NUMA, and manycore platforms for an irregular application. In *Workshop on Irregular Applications: Architectures & Algorithms (IAˆ3)*, pages 5:1–5:8, Denver, EUA. ACM.

Cole, M. (2004). Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406.

Demmel, J. W. (1997). *Applied numerical linear algebra*. SIAM.

Enmyren, J. and Kessler, C. W. (2010). SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA. ACM.

Francesquini, E., Castro, M., Penna, P. H., Dupros, F., de Freitas, H. C., Navaux, P. O. A., and Méhaut, J.-F. (2014). On the energy efficiency and performance of irregular applications on multicore, NUMA and manycore platforms. *J. Parallel Distrib. Comput.*, 76:32–48.

Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., Zhao, W., Yin, X., Hou, C., Zhang, C., Ge, W., Zhang, J., Wang, Y., Zhou, C., and Yang, G. (2016). The sunway taihulight supercomputer: system and applications. *SCIENCE CHINA Information Sciences*, 59(7):072001:1–072001:16.

Gardner, M. (1970). Mathematical Games - The Fantastic Combinations of John Conway's New Solitaire Game 'Life'. *Scientific American*, 223(3).

Gonzalez, R. C. and Woods, R. E. (2006). *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc.

Holewinski, J., Pouchet, L.-N., and Sadayappan, P. (2012). High-Performance Code Generation for Stencil Computations on GPU Architectures. In *ACM ICS*, pages 311–320.

Kale, L. V. and Bhatele, A., editors (2012). *Parallel Science and Engineering Applications: The Charm++ Approach*. CRC Press, 1st edition.

Lam, B. C., George, A. D., and Lam, H. (2013). TSHMEM: Shared-Memory Parallel Computing on Tilera Many-Core Processors. In *IEEE International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW)*, pages 325–334, Cambridge, USA. IEEE Computer Society.

Lutz, T., Fensch, C., and Cole, M. (2013). PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems. *ACM Trans. Archit. Code Optim.*, 9(4):59:1–59:24.

Meng, J. and Skadron, K. (2011). A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations. *International Journal of Parallel Programming*, 39(1):115–142.

Padoin, E. L., Pilla, L. L., Castro, M., Boito, F. Z., Navaux, P. O. A., and Méhaut, J.-F. (2015). Performance/Energy Trade-off in Scientific Computing: The Case of ARM big.LITTLE and Intel Sandy Bridge. *IET Computers & Digital Techniques*.

Pereira, A. D., Ramos, L., and Góes, L. F. W. (2015). PSkel: A stencil programming framework for cpu-gpu systems. *Concurrency and Computation: Practice and Experience*, 27(17):4938–4953.

Rocha, R. C. O., Pereira, A. D., Ramos, L., and Góes, L. F. W. (2017). TOAST: Automatic tiling for iterative stencil computations on GPUs. *Concurrency and Computation: Practice and Experience*, 29(8):e4053.

Sirdey, P. A., Beaucamps, P.-E., Blanc, F., Bobin, B., Carpov, S., Cudennec, L., David, V., Dore, P., Dubrulle, P., de Dinechin, B. D., François Galea, Goubier, T., and Harrand, M. (2013). Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor. In *International Conference on Computational Science (ICCS)*, volume 18, pages 1624–1633, Barcelona, Spain. Elsevier Science.

Steuwer, M., Kegel, P., and Gorlatch, S. (2011). SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1176–1182, Washington, DC, USA. IEEE Computer Society.

Totoni, E., Behzad, B., Ghike, S., and Torrellas, J. (2012). Comparing the Power and Performance of Intel's SCC to State-of-the-Art CPUs and GPUs. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 78–87, New Brunswick, Canada. IEEE Computer Society.

Varghese, A., Edwards, B., Mitra, G., and Rendell, A. P. (2014). Programming the Adapteva Epiphany 64-core network-on-chip coprocessor. In *International Parallel Distributed Processing Symposium Workshops (IPDPSW)*, pages 984–992, Phoenix, USA. IEEE Computer Society.

Weaver, V. M., Johnson, M., Kasichayanula, K., Ralph, J., Luszczek, P., Terpstra, D., and Moore, S. (2012). Measuring energy and power with PAPI. In *2012 41st International Conference on Parallel Processing Workshops*, pages 262–268.

Geração Automática de Estênceis Otimizados para GPUs

Alyson D. Pereira[1], **Rodrigo C. O. Rocha**[3], **Márcio Castro**[1], **Luís F. W. Góes**[2]

[1] Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD)
Universidade Federal de Santa Catarina (UFSC) – SC, Brasil

[2] Grupo de Computação Criativa e Paralela (CreaPar)
Pontifícia Universidade Católica de Minas Gerais (PUC Minas) – MG, Brasil

[3] Institute for Computing Systems Architecture (ICSA)
University of Edinburgh – Escócia, Reino Unido

alyson.pereira@posgrad.ufsc.br, r.rocha@ed.ac.uk,
marcio.castro@ufsc.br, lfwgoes@pucminas.br

Resumo. *Neste artigo propomos uma ferramenta que utiliza uma análise estática para detectar computações estêncil em laços aninhados em um códigos C/C++ e um gerador de código que, baseado nas informações do padrão de vizinhança da computação estêncil, gera um código CUDA otimizado. Para validar a nossa ferramenta, analisamos um conjunto de códigos presentes no benchmark Polybench, o qual contem códigos dos domínios de estatística, algebra linear e estêncil. Os resultados mostraram que a análise estática foi capaz de detectar corretamente o padrão estêncil. Além disso, o código gerado pela ferramenta proposta apresentou desempenho de até* $2.25\times$ *ao código gerado automaticamente por um compilador referência no estado da arte.*

1. Introdução

Um importante padrão paralelo utilizado em aplicações científicas é o estêncil, obtido através da discretização de equações diferenciais parciais. Este padrão opera em estruturas de dados multidimensionais, nas quais uma mesma computação é realizada sobre cada elemento utilizando os valores de seus vizinhos. Pesquisas recentes tentam explorar o uso eficiente de Unidades de Processamento Gráfico (*Graphics Processing Units* - GPUs) para a execução de aplicações estêncil, tendo em vista o seu alto grau de paralelismo de dados [Meng and Skadron 2011, Maruyama and Aoki 2014, Holewinski et al. 2012].

Diversas são as abordagens para a programação de GPUs: linguagens de baixo nível, bibliotecas de esqueletos paralelos, linguagens de domínio específico e diretivas de compilação. A linguagem de baixo-nível CUDA permite um controle maior sobre otimizações de código, porém requer um conhecimento da arquitetura da GPUs e pode ser desafiadora para os menos experientes. As bibliotecas de esqueletos paralelos e as linguagens de domínio específico facilitam a programação ao mesmo tempo que permitem explorar determinadas otimizações, no entanto requerem uma reescrita de códigos estêncil já existentes e geralmente apresentam certas restrições ao programador. Com o uso de diretivas, como OpenACC e OpenMP 4.5, o programador pode utilizar um código-fonte sequencial já existente e apenas anotá-lo com as devidas diretivas, indicando os trechos de códigos que serão executados na GPU. Apesar do uso das diretivas ser mais simples do ponto de vista de programação, elas não exploram otimizações de baixo nível que podem ser obtidas ao se utilizar as abordagens anteriores.

Neste artigo propomos uma abordagem que utiliza uma análise estática que detecta computações estêncil em laços aninhados em um código-fonte C/C++ e um gerador de código que, baseado nas informações do padrão de vizinhança da computação estêncil obtido através da analise anterior, gera um código CUDA otimizado. Ambas as contribuições foram implementadas utilizando a infra-estrutura de compilação LLVM e integradas em uma ferramenta denominada `PSkelCC`. Para validar a nossa ferramenta, analisamos um conjunto de códigos presentes no *benchmark* Polybench[1], que contem códigos dos domínios de estatística, algebra linear e estêncil. A análise estática foi capaz de detectar corretamente os programas selecionados deste *benchmark* que são estêncil e rejeitar os que não são. Além disso, analisamos um conjunto de códigos que implementam um estêncil laplaciano 3D e comparamos o desempenho do código gerado pela nossa ferramenta frente a implementações otimizadas manualmente e a de códigos gerados automaticamente por um compilador *source-to-source* referência no estado da arte. Obtemos um desempenho de até $2.25\times$ sobre o compilador de referência e até $1.26\times$ sobre códigos otimizados manualmente.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta os principais conceitos do padrão estêncil, modelo de programação CUDA e do compilador LLVM. As Seções 4 e 5 discutem a detecção do padrão estêncil em códigos sequenciais e a geração de código CUDA para os estênceis detectados, respectivamente. Os resultados obtidos são apresentados na Seção 6. Por fim, a Seção 3 apresenta os trabalhos relacionados ao tema abordado e a Seção 7 apresenta as conclusões deste trabalho.

2. Fundamentação Teórica

2.1. Computação Estêncil

O padrão estêncil opera em dados multidimensionais e aplica uma computação em cada um de seus elementos, utilizando os valores dos seus vizinhos. Em estênceis iterativos, este procedimento se repete, utilizando os dados de saída como entrada para a próxima iteração. A Figura 1 ilustra alguns padrões de vizinhanças que ocorrem em estênceis 2D e 3D.

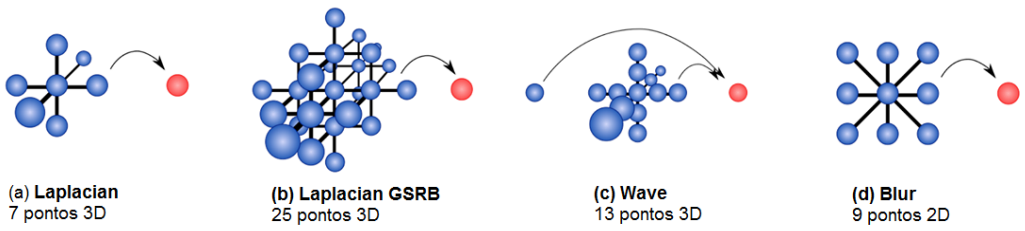


Figura 1. Diferentes vizinhanças do padrão estêncil [Christen et al. 2012].

O Código 1 exemplifica uma computação estêncil iterativa de uma dimensão. Neste exemplo, o número de iterações é determinado pelo parâmetro `tsteps` (linha 1). A computação estêncil é realizada sobre cada elemento do vetor `A`, de tamanho `N`, utilizando o coeficiente `c1` e uma vizinhança de 2 elementos (linha 7), exceto para os

[1]http://web.cse.ohio-state.edu/ pouchet.2/software/polybench/

elementos em sua borda. O resultado da computação é armazenado em um elemento correspondente do vetor `B`. Após efetuar esta computação em todos os elementos do vetor, existe um laço para a troca de dados entre os vetores `A` e `B` (linhas 9 a 10), pois o vetor de resultado de uma iteração t deverá ser utilizada como vetor de entrada da iteração $t + 1$.

```
1  void jacobi(int tsteps, int N, float *A, float *B){
2    int t, i, j;
3    float c1 = 0.33333;
4
5    for (t = 0; t < tsteps; t++) {
6      for (i = 1; i < N - 1; i++)
7        B[i] = c1 * (A[i-1] + A[i] + A[i + 1]);
8
9      for (j = 1; j < N - 1; j++)
10       A[j] = B[j];
11   }
12 }
```

Código 1. Exemplo de um código estêncil 1D iterativo em C.

2.2. Modelo de Programação CUDA

No modelo de programação CUDA, a GPU é vista como um co-processador com massivo paralelismo de dados, capaz de executar centenas de *threads* em paralelo no modelo de execução *Single Instruction, Multiple Threads* (SIMT). Para tal, uma função denominada *kernel* opera, geralmente, em cada elemento dos dados de entrada e é compilada para o conjunto de instruções (*Instruction Set Architecture* - ISA) da GPU. Para manter uma portabilidade entre diferentes microarquiteturas, é utilizado um conjunto de instruções virtual, denominado *Parallel Thread Execution* (PTX). Em tempo de execução, o programa resultante é mapeado em *threads*. Para organizar as diferentes *threads* que executam um mesmo *kernel* no dispositivo, conjuntos de *threads* são combinados em um bloco de *threads*, permitindo que elas compartilhem dados e sincronizem entre si.

Uma GPU moderna é composta por um conjunto de *Streaming Multiprocessors* (SM), onde cada SM é composto por núcleos de processamento, unidades aritméticas, unidades de controle, e sua hierarquia de memória. Podemos considerar que existem três conexões na hierarquia de memória: memória global $\leftrightarrow$ *cache* L2 $\leftrightarrow$ memórias locais $\leftrightarrow$ registradores. A memória *cache* L2 é compartilhada entre todos os SMs da GPU. As memórias locais (internas ao SM) incluem *cache* L1, *cache* somente leitura e memória compartilhada. A largura de banda entre cada uma dessas conexões são consideravelmente diferentes, sendo a largura de banda da memória global a menor de todas. O volume de dados também se difere, sendo o maior de todos entre a memória local e os registradores. Por esse motivo, apesar das GPUs possuírem um alto poder de computação teórico (GPUs atuais alcançam um desempenho na ordem de TeraFLOPS[2]), o desempenho de computações estêncil se torna limitado à largura de banda das GPUs, sendo necessário manter a maior quantidade de dados disponíveis em *cache* para reuso pelos blocos de *threads*, explorando a localidade de dados tanto de maneira espacial (compartilhando dados entre as *threads* de um mesmo bloco através das memórias locais) quanto de maneira temporal (realizando mais de uma iteração do estêncil internamente, reutilizando os dados já armazenados nas memórias locais e reduzindo a comunicação com as memórias externas, ao custo de computações redundantes).

[2]http://www.nvidia.com/object/tesla-p100.html

2.3. Análise Estática com LLVM

O *Low Level Virtual Machine* (LLVM) [Lattner and Adve 2004] é uma infraestrutura de compilação amplamente utilizada na indústria. Ele é composto principalmente por uma representação intermediária (*LLVM Intermediate Representation* - IR[3]) e bibliotecas para análise e otimização de códigos. O processo de compilação é composto por três etapas: um *frontend*, um otimizador e um *backend*, como exibido na Figura 2. O *frontend* é responsável por realizar um *parsing* no código fonte, verificando por erros sintáticos e semânticos, e transformá-lo na representação intermediária do LLVM. O otimizador é responsável por realizar uma vasta quantidade de análises e otimizações sobre a IR de maneira independente da arquitetura alvo. Por fim, o *backend* traduz o código da representação intermediária para o conjunto de instruções da máquina alvo.

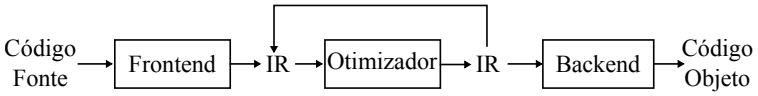


Figura 2. Visão geral da compilação em três passos do LLVM.

A IR do LLVM é um conjunto de instruções de baixo nível do tipo RISC, com tipagem forte, que opera em uma arquitetura *load/store* considerando um número infinito de registradores virtuais. Além disso, a IR segue a forma *Static Single Assignment* (SSA), onde cada registrador virtual é escrito uma única vez e seu uso ocorre após a sua definição. A principal vantagem de se utilizar esta forma é que ela simplifica e melhora diversas otimizações e análises [Cytron et al. 1991]. Uma outra característica da forma SSA é o uso da função ϕ. Esta função é aplicada na transformação para a forma SSA canônica, quando são atribuídos para uma mesma variável valores distintos de acordo com o fluxo de controle do programa.

Informações sobre a IR de um programa podem ser obtida através das diversas análises do LLVM, implementadas como passes[4]. Neste trabalho, fazemos uso de duas análises para a detecção do padrão estêncil: `Scalar Evolution` e `LoopInfo`. O passe `Scalar Evolution` calcula para cada registrador inteiro uma expressão fechada, denominada evolução escalar, que descreve o seu valor durante a execução do programa [Bachmann et al. 1994]. A vantagem de seu uso é que ela abstrai as instruções que compõem o valor de um registrador e foca no cálculo de uma maneira geral. A expressão de uma evolução escalar é construída recursivamente através de diversos elementos. Existem dois elementos base principais e um conjunto de elementos definidos indutivamente. Os elementos base são *integer constant*, que correspondem as constantes inteiras conhecidas em tempo de compilação, e o *unknown value*, que corresponde a valores dinâmicos, conhecidos apenas em tempo de execução, por exemplo, como parâmetros de função ou uma instrução de `load`. Alguns dos elementos indutivos são as operações n-árias como adição (+), multiplicação (`*`), máximos com e sem sinal (`smax` e `umax`) e a adição recursiva. A adição recursiva representa expressões que se alteram durante a avaliação de um laço. Elas contêm o formato {`<base>, +, <step>`}`<loop>`. A *base* de uma adição recursiva corresponde ao seu valor na primeira iteração do laço *loop* e o *step* define o valor adicionado em cada iteração subsequente. Caso o registrador esteja

[3]http://llvm.org/docs/LangRef.html
[4]https://llvm.org/docs/Passes.html

contido em laços aninhados e seu valor dependa dos laços externos (como o indice do laço mais externo, por exemplo), a base da adição recursiva será uma outra adição recursiva.

O passe `LoopInfo` obtém informações sobre laços que utilizam, por exemplo, a construção `for`. Com o uso do passe `LoopInfo`, é possível obter a quantidade de vezes que o laço é executado, na forma de uma expressão escalar, e qual a sua variável de indução canônica, na forma de uma instrução ϕ. Uma variável de indução canônica representa uma variável com valor inicial zero e que é incrementada em um à cada iteração do laço. Uma variável de indução que é incrementada em um laço mas não se inicia em zero não é canônica.

3. Trabalhos Relacionados

Alguns trabalhos recentes propuseram abordagens para a geração automática de um código sequencial em um código correspondente para GPUs. O compilador PPCG é o atual estado da arte em compilação automática de códigos sequenciais para CUDA [Verdoolaege et al. 2013]. O compilador é baseado no modelo *poliedral* e é capaz de realizar análise de dependências e transformações de laços sem intervenção do usuário. No entanto, existe a restrição de que apenas laços estáticos sejam transformados.

Em um trabalho prévio [Pereira et al. 2017] foi proposto uma diretiva de compilação específica para o padrão estêncil e um compilador *source-to-source* que transforma códigos sequenciais anotados com a diretiva estêncil em códigos equivalentes de um *framework* de esqueletos paralelos [Pereira et al. 2015]. Neste presente trabalho, eliminamos a necessidade do uso de diretivas e propomos uma análise estática para detecção de computações estêncil em um código sequencial. Além disso, neste trabalho realizamos uma geração de código diretamente em CUDA, o que permite uma maior flexibilidade na escolha de otimizações e transformações de código.

O compilador *BONES* [Nugteren and Corporaal 2014] é um compilador *source-to-source* que utiliza um conceito denominado "espécies algorítmicas", que classifica trechos de código de acordo com o padrão de acesso à memória: *element*, *neighbourhood*, *chunk*, *full* e *shared*. Uma espécie correspondente ao padrão estêncil é formada pela combinação dos padrões *neighbourhood* e *element*. Bones recebe um código C anotado com as espécies algorítmicas e gera um código CUDA adequado.

O compilador CUDA-Chill transforma códigos sequenciais em códigos CUDA baseado em um conjunto de regras informadas pelo usuário [Khan et al. 2013]. Estas regras incluem o mapeamento de dados na hierarquia de memória, o tamanho dos *tiles* e níveis para desenrolamento de laços.

Por fim, o compilador DawnCC, diferentemente das abordagens anteriores, realiza a anotação automática das diretivas de compilação dos padrões OpenACC e OpenMP 4.5 em um código-fonte sequencial, permitindo sua execução em GPU. A ferramenta é implementada como uma conjunto de análises estáticas em LLVM, capaz de detectar se um laço é paralelizável [Mendonca et al. 2017].

4. Detecção do Padrão Estêncil

Nesta seção, ilustramos as etapas da detecção de um padrão estêncil implementado na ferramenta `PSkelCC`. A detecção foi implementada como um passe de otimização que é

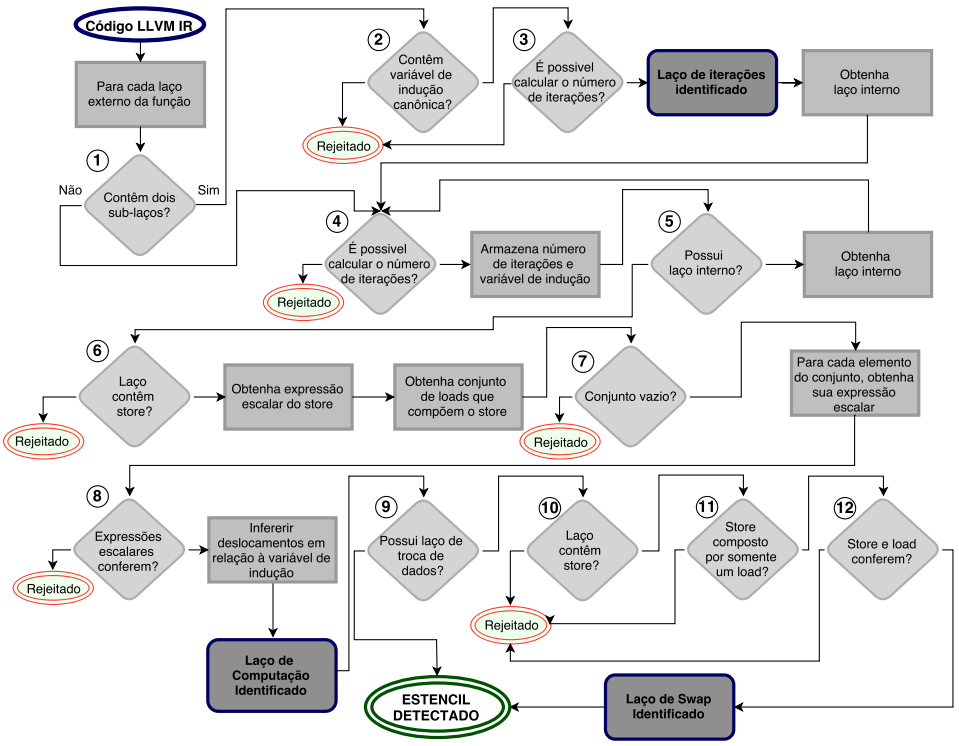


Figura 3. Visão geral da detecção do padrão estêncil.

executado para cada função existente no código IR. Na implementação atual, consideramos como pré-condições a verificação entre as dependências de dados e que os ponteiros analisados não possuem *aliasing* entre si. Uma análise de dependências de dados poderia, por exemplo, verificar as condições básicas Bernstein [Bernstein 1966]. Tais condições garantem que um código pode ser executado em paralelo, mantendo a semântica original. Para que as condições de Bernstein possam ser verificadas, é necessário que se tenha uma análise de *aliasing* de ponteiros, para que as condições de leitura e escrita em memória possam ser garantidas mesmo com o uso de ponteiros.

A Figura 3 exibe o fluxograma do processo de detecção do padrão estêncil proposto neste artigo. Primeiramente, obtemos os resultados da análise `LoopInfo` para que se possa analisar a estrutura dos laços. Para cada laço mais externo da função analisada, realiza-se as verificações enumeradas de 1 a 12 no fluxograma. As etapas de 1 a 3 se referem a detecção do laço de iterações. O laço de iterações deve conter dois sublaços (etapa 1), um correspondente ao laço mais externo da computação estêncil (linhas 6 e 7 do Código 1) e outro correspondente ao laço da troca de ponteiros (linhas 9 e 10 do Código 1). Caso ele contenha apenas um sublaço, ele é considerado um laço de computação e executa-se a etapa 4. Caso ele contenha dois sublaços, obtém-se uma expressão que corresponde ao número de vezes que o laço iterações é executado. A análise de evolução escalar é utilizada para inferir o número de iterações executadas pelo laço. Verifica-se também se a variável de indução do laço de iterações é utilizada apenas para o controle do laço. Para isso, verificamos se a varíavel de indução é utilizada apenas nos blocos básicos de cabeçalho e retorno do laço, sem nenhuma utilização no corpo do laço.

As etapas 4 a 8 correspondem a detecção dos laços de computação. A quantidade de laços aninhados corresponde à quantidade de dimensões do estêncil. De maneira similar ao laço de iterações, para cada laço aninhado de computação, obtém-se a expressão correspondente à quantidade de execuções do laço bem como sua variável de indução. As expressões, obtidas na forma de evolução escalar, correspondem ao tamanho de cada dimensão do estêncil. Essas informações serão utilizadas para analisar as expressões de indexação dos acessos à memória.

As principais instruções utilizadas para realizar manipulação de memória são as instruções de `load` e `store`. Essas instruções recebem como um de seus operandos um endereço de memória. Estes endereços, por sua vez, são computados pela instrução `getelementptr`, dado um endereço base e o índice desse elemento. Assim, identificar uma vizinhança estêncil corresponde a identificar todas as instruções de `load`, utilizadas para acessar os dados de entrada, e a instrução de `store`, responsável por armazenar o resultado de saída. As instruções de `load` não podem utilizar um mesmo ponteiro base que é utilizado pela instrução de `store`.

A etapa 6 verifica se o laço mais interno contêm uma instrução de `store` e obtém a expressão de evolução escalar correspondente à esta instrução. Por exemplo, no Código 2, referente à atribuição de um valor ao endereço de memória `B[i]`, a expressão de evolução escalar referente ao registrador `%storeidx` é definida por {`(4 + %B),+,4`}`<%for.cond>`}. Como o ponteiro base %B é do tipo `float`, cada posição de memória corresponde à 4 bytes. A evolução escalar considera a primeira iteração do laço e, como neste laço o valor inicial de sua variável de indução é 1, a base da adição recursiva corresponde ao endereço do ponteiro %B incrementado pelo valor de uma posição de memória. Outras informações obtidas desta expressão é que a instrução está contido no laço que tem início no bloco básico `for.cond`, a cada iteração o endereço base é incrementado em uma posição de memória (*step* da adição recursiva).

```
1  for.cond:
2    %i = phi i64 [ 1, %for.body ], [ %inc, %for.inc ]
3    ...
4  for.body:
5    ...
6    %storeidx = getelementptr float, float* %B, i64 %i
7    store float %val, float* %storeidx
8    ...
9  for.inc:
10   %inc = add i32 %i, 1
11   br label %for.cond
```

Código 2. Armazenamento em um endereço de memória em LLVM-IR.

Posteriormente, na etapa 7, as instruções que compõem o valor a ser armazenado pela instrução `store` são percorridas e cada instrução de `load` é armazenada em uma lista. Para cada elemento desta lista, é gerado a sua expressão de evolução escalar. De maneira que esses acessos componham uma vizinhança estêncil, os valores de *step* e dos laços da adição recursiva devem ser iguais, além de possuirem um mesmo ponteiro base, diferente do ponteiro base da operação de `store` (etapa 8). O ponteiro base de `load` é definido como entrada do estêncil. O valor do deslocamento em relação ao elemento central da computação estêncil é inferido analisando a expressão de indexação de acesso à memória. No Código 3, referente à leitura de um valor no endereço de memória `A[i+1]`, a expressão de acesso é composta pelo valor da variável de indução (registrador `%i`)

incrementado pelo deslocamento em relação ao elemento central da computação estêncil, acessando a partir do ponteiro base %A.

```
1  %add = add i64 %i, 1
2  %loadidx = getelementptr float, float* %A, i64 %add
3  %data = load float, float* %loadidx
```

Código 3. Leitura de um endereço de memória em LLVM-IR.

As etapas 9 a 12 são executadas caso tenha sido detectado previamente um laço de iteração. Caso contrário, o conjunto de laços analisados compõem uma computação estêncil de uma única iteração. Nestas etapas, é verificado se a instrução `store` é composta por apenas uma instrução de `load`, e se o ponteiro base das instruções de `load` correspondem ao ponteiro de saída, e o ponteiro base da instrução de `store` corresponde ao ponteiro definido como entrada. Caso tais condições sejam satisfeitas, o conjunto de laços analisados compõem uma computação estêncil iterativa. Ao final da detecção, é gerado uma estrutura de dados contendo todas as informações obtidas que caracterizam a computação estêncil. Esta estrutura será acessada pelo passe de otimização responsável pela geração de código CUDA.

5. Geração de Código CUDA

Uma vez que as informações sobre o padrão de vizinhança da computação estêncil já foram obtidas pelo passe descrito na seção anterior, é executado um segundo passe LLVM realizando a geração de código na linguagem CUDA. Na implementação atual, limitamos o laço de computação mais interno para conter apenas um bloco básico sem chamada de função. Essa limitação simplifica ambas a detecção do padrão estêncil como a geração de código CUDA, mas representa uma grande classe das aplicações no padrão estêncil. Além disso, limitamos a geração de código somente para estênceis 3D.

O passe de geração de código em CUDA realiza a extração do código de computação do laço mais interno, criando uma nova função *kernel* para execução em GPU. Neste *kernel*, as variáveis de indução dos laços de computação são transformados em variáveis que indexam o acesso dos dados na memória da GPU. Além disso, é criada uma função que gerencia a transferência de dados para a GPU e é gerado um laço, correspondente ao laço de iterações da computação estêncil, que realiza chamadas para a execução do *kernel* em GPU.

Realizamos duas gerações de códigos distintas, uma simples e uma com otimizações. A geração de código simples realiza uma tradução direta do bloco básico em LLVM para o código CUDA. Para a geração de código mais otimizada, foi escolhida uma otimização que apresentou um melhor resultado geral em trabalhos anteriores da literatura [Maruyama and Aoki 2014, Nasciutti and Panetta 2016]. Nesta otimização, é realizado uma iteração sobre o eixo Z do estêncil 3D, armazenando os valores deste eixo na memória *cache* somente leitura, e é explorado a localidade de dados do eixo XY, armazenando estes valores na memória compartilhada. A seleção dos valores que serão armazenadas na memória compartilhada (eixo XY) ou na *cache* somente leitura (eixo Z) é feita a partir dos deslocamentos em relação ao elemento central do estêncil, previamente obtidos no passe de detecção. Adicionalmente, é realizado um bloqueio temporal de duas iterações, realizadas internamente na GPU, a fim de diminuir a comunicação entre a *cache* L2 e a memória global, porém ao custo da realização de computações redundantes.

6. Experimentos

Nesta seção validamos o PSkelCC em relação à sua detecção do padrão estêncil e o desempenho obtidos pelos códigos gerados automaticamente em relação aos códigos otimizados escritos manualmente e aos códigos gerados automaticamente pelo compilador do estado da arte PPCG, na versão 0.07. O PSkelCC foi desenvolvido utilizando LLVM 3.7. Todos os códigos gerados foram compilados utilizando-se CUDA 8.0. Os experimentos foram realizadas em uma GPU Tesla K40c, composta por 2.880 núcleos CUDA, 12GB de memória memória global (DRAM), 1.5MB de memória *cache* L2, 48KB de *cache* somente leitura, 48KB de memória compartilhada e 16KB de *cache* L1.

6.1. Detecção de Códigos Estêncil

Utilizamos um conjunto de códigos do *benchmark* Polybench para avaliar a detecção do padrão estêncil e selecionamos os seguintes códigos: `conv2d`, `conv3d`, `jacobi1d` e `jacobi2d`. Além disso, foram implementadas versões sequenciais do estêncil laplaciano 3D com diferentes vizinhanças. Todos estes códigos estêncil foram detectados corretamente pelo PSkelCC. Todos os demais código do *benchmark* pertencentes a outro domínio também foram analisados corretamente, como não sendo pertencentes ao padrão estêncil. Estes códigos foram os seguintes: `2mm`, `3mm`, `atax`, `bicg`, `correlation`, `covariance`, `gemm`, `gemsumv`, `mvt`, `syrk` e `syr2k`. Os códigos estêncil `adi`, `fdtd-2d`, `fdtd-ampl e seidel` não foram analisados pois, apesar de possuirem o padrão de vizinhança estêncil, possuem um padrão de computação que, no momento, não é detectado pelo PSkelCC.

6.2. Desempenho

Nesta seção apresentamos o desempenho das implementações de um estêncil laplaciano 3D. Foram realizadas diferentes implementações variando a dimensão do raio da vizinhança do estêncil de 1 a 5, sobre cada eixo do espaço 3D, totalizando uma vizinhança de 7, 13, 19, 25 e 31 elementos, respectivamente. Cada um desses estênceis foram executados com tamanhos de 64^3, 128^3 e 256^3, com 50 iterações cada, em um total de 10 execuções. Foi comparado o desempenho do código gerado pelo PSkelCC sem otimização (pskelcc-base) e com uma otimização utilizando memória cache somente leitura, internalização do eixo Z e bloqueio temporal utilizando a memória compartilhada (pskelcc-opt-temp) em relação aos códigos gerados pelo compilador PPCG de maneira completamente automática (ppcg-base) e configurado manualmente para um melhor desempenho (ppcg-opt), além de códigos escritos manualmente sem otimização (manual-base), somente com otimização do uso da memória cache somente leitura e internalização do eixo Z (manual-opt) e com uso de memória cache somente leitura, internalização do eixo Z e bloqueio temporal utilizando a memória compartilhada (manual-opt-temp).

A Figura 4 apresenta o desempenho em GFlop/s de cada uma das implementações apresentadas anteriormente. Além disso, é apresentado uma média geométrica das execuções das diferentes vizinhanças. A Tabela 1 apresenta a ocupação medida através de *profiling* para cada umas das implementações. A ocupação representa a relação entre a média de *warps* ativos por ciclo e o número máximo de *warps* suportados pela GPU. Ela está diretamente ligada a eficiência obtida na GPU, isto é, ela indica se os recursos da GPU estão sendo utilizados de maneira eficiente.

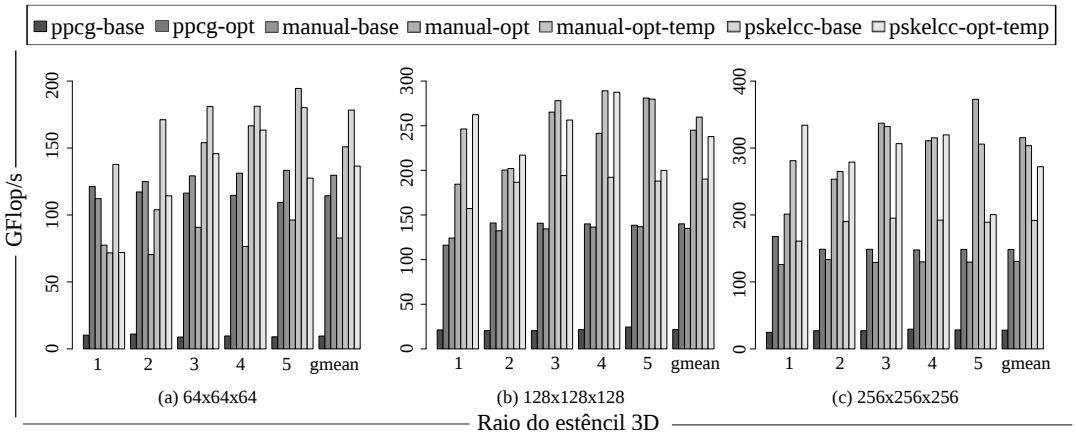


Figura 4. Desempenho de estênceis 3D.

A implementação base do PPCG foi a que obteve o pior desempenho para todos os casos. O baixo desempenho se deve ao uso de condicionais no código-fonte gerado, ocasionando divergências e diminuindo a eficiência da execução das *warps* (eficiência média de 92%) e a ocupação da GPU. Por outro lado, as implementações manuais e as geradas pelo PSkelCC possuem 100% de eficiência, já que não possuem condicionais. No geral, o pskelcc-base obteve um *speedup* médio de 10.98x sobre o ppcg-base.

A implementação ppcg-opt obteve um desempenho consideravelmente melhor que o ppcg-base, com um *speedup* médio de 11.92x. No entanto, o pskelcc-base apresenta uma maior ocupação em todos os casos, resultando em um *speedup* médio de 1.47x sobre o ppcg-opt (máximo de 1.64x). Por outro lado, a implementação otimizada do PSkelCC apresenta uma baixa ocupação, e obtem um menor desempenho em relação ao ppcg-opt para estênceis de raio 1 e 2 com tamanho de 64^3. No geral, a implementação pskelcc-opt-temp apresenta um *speedup* médio de 1.59x (máximo de 2.25x) em relação ao ppgc-temp.

Em relação às implementações manuais, o pskelcc-base obteve um desempenho melhor que o manual-base para todos os casos, com um *speedup* médio de 1.38x (máximo de 1.48x), apesar de ambos apresentarem ocupações similares. Como ambas as implementações se diferem apenas na ordem das instruções e na forma de indexação dos dados em memória, esta melhoria de desempenho se dá, principalmente, pela alocação de registradores realizada pelo compilador do CUDA. Ao analisar a quantidade de registradores alocados para cada *kernel*, observamos que a quantidade alocada para os códigos gerados pelo PSkelCC é menor, justificando o aumento de desempenho. A implementação pskel-opt-temp, que utilizada bloqueio temporal, somente não alcançou um desempenho melhor que o manual-base em três casos, para os estênceis de raios 1, 2 e 5, com tamanho 64^3. Um dos motivos para isto é que, por este tamanho ser relativamente pequeno, o uso das computações redundantes nestes estênceis não apresentou benefício em relação à diminuição dos acessos à memória global. Para o estêncil de raio 5, ao analisar a quantidade de registradores alocados ao kernel, observamos que foram alocados um número alto de registradores, que degrada o desempenho, além de diminuir a ocupação. Para os demais casos, é alcançado um *speedup* médio de 1.83x (máximo de 2.65x).

Ao compararmos o desempenho dos códigos manuais otimizados, observa-se que

Implementação	Ocupação Medida da GPU														
	1			2			3			4			5		
	64^3	128^3	256^3	64^3	128^3	256^3	64^3	128^3	256^3	64^3	128^3	256^3	64^3	128^3	256^3
ppcg-base	0.23	0.27	0.48	0.22	0.26	0.47	0.21	0.25	0.47	0.19	0.24	0.46	0.18	0.24	0.45
ppcg-opt	0.62	0.68	0.70	0.43	0.45	0.48	0.43	0.45	0.48	0.41	0.45	0.48	0.40	0.45	0.47
manual-base	0.84	0.84	0.83	0.87	0.85	0.85	0.87	0.87	0.87	0.89	0.89	0.89	0.68	0.68	0.68
manual-opt	0.49	0.53	0.72	0.24	0.53	0.72	0.24	0.53	0.72	0.24	0.52	0.72	0.24	0.53	0.73
manual-opt-temp	0.24	0.60	0.86	0.24	0.41	0.48	0.24	0.41	0.48	0.26	0.46	0.49	0.34	0.47	0.48
pskelcc-base	0.83	0.84	0.84	0.84	0.85	0.85	0.85	0.86	0.86	0.86	0.87	0.86	0.87	0.86	0.86
pskelcc-opt-temp	0.24	0.59	0.86	0.24	0.41	0.48	0.24	0.42	0.49	0.26	0.45	0.49	0.24	0.24	0.24

Tabela 1. Ocupação da GPU para diferentes raios de vizinhanças e tamanhos.

o uso do bloqueio temporal não apresentou melhoria em 4 dos 15 casos, para os estênceis de raio 1 com tamanho 64^3, raio 3 com tamanho 256^3 e raio 5 com tamanhos 128^3 e 256^3. No entanto, para os demais casos, esta otimização apresenta um *speedup* médio de 1.35x (máximo de 2.17x) sobre as implementações otimizadas sem bloqueio temporal. Ao compararmos com as implementações não otimizadas, o uso do bloqueio temporal, similar ao código gerado pelo PSkelCC, somente não apresenta melhoria para os casos com raios 1 e 2, com tamanho 64^3. Para os demais casos, um speedup médio de 1.9x (máximo de 2.57x) é alcançado.

Ao comparar o desempenho da implementação otimizada gerada pelo PSkelCC em relação à implementação manual otimizada com bloqueio temporal, obtêm-se um melhor desempenho em 7 dos 15 casos, com um *speedup* médio de 1.06x (máximo de 1.19x). Apesar de ambos os códigos implementerem o mesmo algoritmo de bloqueio temporal, certas diferenças na ordem de instruções e na maneira de acesso a memória fazem com que estes códigos tenham uma alocação de registradores diferentes entre si, similar ao que ocorre nas implementações base. Para os casos em que há melhoria de desempenho, foram alocados menos registradores e, nos casos em que há queda de desempenho, foram alocados mais registradores. Este aspecto teve um maior impacto no estêncil de raio 5, em que os códigos gerados alcançaram entre 65% e 71% do desempenho da implementação otimizada manualmente.

7. Conclusões e Trabalhos Futuros

Neste trabalho apresentamos uma ferramenta para detecção e geração automática de códigos estênceis para GPUs a partir de análises estáticas no código-fonte sequencial. A ferramenta proposta foi capaz de identificar corretamente código do padrão estêncil no conjunto de aplicações do benchmark Polybench, além de gerar códigos estênceis com desempenho até $2.25\times$ sobre o compilador do estado da arte PPGC e com desempenho similar ou superior à grande parte dos códigos gerados manualmente.

Trabalhos futuros incluem investigar os aspectos da geração de código que impactam na alocação de registradores e no desempenho, além de incluir e avaliar a geração de código para estênceis 1D, 2D e miniaplicações. Pode-se citar também a inclusão de análises de *aliasing* de ponteiros, dependência de dados e verificações para casos particulares de determinados estênceis. Por fim, pode-se incluir um modelo de custo para o uso da hierarquia de memória da GPU.

Referências

Bachmann, O., Wang, P. S., and Zima, E. V. (1994). Chains of recurrences: a method to expedite the evaluation of closed-form functions. In *Int. Symp. on Symbolic and Algebraic Computation*, pages 242–249.

Bernstein, A. J. (1966). Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763.

Christen, M., Schenk, O., and Cui, Y. (2012). PATUS for convenient high-performance stencils: Evaluation in earthquake simulations. In *Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 11:1–11:10.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490.

Holewinski, J., Pouchet, L.-N., and Sadayappan, P. (2012). High-performance code generation for stencil computations on GPU architectures. In *ACM Int. Conf. on Supercomputing (ICS)*, pages 311–320.

Khan, M., Basu, P., Rudy, G., Hall, M., Chen, C., and Chame, J. (2013). A script-based autotuning compiler system to generate high-performance CUDA code. *ACM Trans. Archit. Code Optim.*, 9(4):31:1–31:25.

Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *Int. Symp. on Code Generation and Optimization*.

Maruyama, N. and Aoki, T. (2014). Optimizing stencil computations for NVIDIA Kepler GPUs. In *Int. Workshop on High-Performance Stencil Computations*, pages 89–95.

Mendonca, G. S. D., Guimaraes, B. C. F., Alves, P. R. O., Pereira, M. M., Araujo, G., and Pereira, F. M. Q. (2017). Dawncc: Automatic annotation for data parallelism and offloading. *ACM Trans. Archit. Code Optim.*, 14:13:1–13:25.

Meng, J. and Skadron, K. (2011). A performance study for iterative stencil loops on GPUs with ghost zone optimizations. *Int. Journal of Parallel Programming*, 39(1):115–142.

Nasciutti, T. C. and Panetta, J. (2016). Impacto da arquitetura de memória de gpgpus na velocidade da computação de estênceis. In *XVII Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD-SSC)*, pages 1–12.

Nugteren, C. and Corporaal, H. (2014). Bones: An automatic skeleton-based c-to-cuda compiler for gpus. *ACM Trans. Archit. Code Optim.*, 11(4):35:1–35:25.

Pereira, A. D., Castro, M., Dantas, M. A. R., Rocha, R. C. O., and Góes, L. F. W. (2017). Extending OpenACC for efficient stencil code generation and execution by skeleton frameworks. In *Int. Conf. on High Perf. Comp. Simulation (HPCS)*, pages 719–726.

Pereira, A. D., Ramos, L., and Góes, L. F. W. (2015). PSkel: A stencil programming framework for CPU-GPU systems. *Concur. and Comp.: Practice and Experience*, 27(17):4938–4953.

Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., and Catthoor, F. (2013). Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23.

Execução Eficiente do Algoritmo de Leilão nas Novas Arquiteturas Multicore

Alexandre C. Sena[1], Aline Nascimento[2], Cristina Vasconcelos[2] e Leandro A. J. Marzulo[1]

[1]Instituto de Matemática e Estatística
Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, Brasil

[2]Instituto de Computação
Universidade Federal Fluminense (UFF), Rio de Janeiro, Brasil

{asena,leandro}@ime.uerj.br,{aline,crisnv}@ic.uff.br

Resumo. *O algoritmo de leilão tem sido amplamente utilizado para resolver o problema de emparelhamento de grafos bipartidos e sua implementação paralela é empregada para encontrar soluções ótimas em um tempo computacional aceitável. Além disso, as novas arquiteturas* multicore*, além de seus vários núcleos de processamento, possuem um conjunto de instruções SIMD que pode aumentar o desempenho da aplicação quando exatamente as mesmas operações necessitam ser realizadas em múltiplos dados. Nesse contexto, o objetivo deste trabalho é explorar todo o potencial dessas arquiteturas na execução do algoritmo de leilão. Para alcançar este objetivo, versões vetorizadas foram implementadas e avaliadas. Em seguida, essas versões foram executadas em paralelo utilizando a biblioteca OpenMP. Os resultados mostram que a versão vetorizada consegue, em média, um desempenho dez vezes melhor que a versão sequencial, enquanto a versão vetorizada paralela é capaz de aproveitar todo o potencial das novas arquiteturas* multicore*, atingindo um desempenho até* 200 *vezes melhor do que a versão sequencial.*

1. Introdução

Existem diversos problemas onde são recebidos como entrada dois conjuntos distintos, a partir dos quais um emparelhamento ótimo deve ser encontrado. Esses problemas podem ser modelados através de grafos bipartidos, cujos nós representam os elementos a serem emparelhados e as arestas podem ser associadas com pesos que representam os custos para emparelhar os nós correspondentes. A combinação (emparelhamento) ótima é o subconjunto final de pares, em que cada nó está presente em no máximo um único par e a soma das arestas escolhidas é mínima/máxima.

O problema de emparelhamento de grafos bipartidos é explorado em diversas áreas tais como controle distribuído e alocação de instalações (*distributed control and facility allocation*), na bioinformática para verificar as interações e semelhanças entre proteínas [Kollias et al. 2013] e na Visão Computacional, onde o objetivo pode ser, por exemplo, o emparelhamento de duas imagens ou reconhecimento de objetos, avaliando as semelhanças entre seus pontos [Shokoufandeh and Dickinson 1999, Vasconcelos and Rosenhahn 2009].

O algoritmo de leilão [Bertsekas 1979] é amplamente utilizado para resolver o problema de emparelhamento de grafos bipartidos. Originalmente, ele foi proposto para

atribuir m pessoas a n objetos distintos, onde cada par (pessoa, objeto) possui um custo associado que representa sua afinidade. O objetivo principal é atribuir uma pessoa ao objeto com maior afinidade, maximizando a soma total [Bertsekas 1992, Carpaneto et al. 1988].

A análise de imagens pode requerer uma grande quantidade de processamento, uma vez que imagens densas podem ter milhares de pontos a serem considerados ou, mesmo para imagens menores, pode ser necessário comparar uma sequência grande de imagens. Portanto, para resolver o problema de emparelhamento de duas imagens em um tempo computacional razoável, a computação paralela tem sido amplamente utilizada [Bertsekas and Castañon 1991, Kollias et al. 2012, Sathe et al. 2012]. Uma opção para aumentar o desempenho do algoritmo de leilão são as novas arquiteturas *multicore* disponíveis, que além de múltiplas unidades de processamento, possuem também instruções SIMD que, quando utilizadas eficientemente, são capazes de aumentar consideravelmente o desempenho das aplicações sequenciais.

Nesse contexto, o principal objetivo deste trabalho é executar o algoritmo de leilão eficientemente nas máquinas *multicore* modernas. Para isso, foram implementadas e analisadas versões sequenciais e paralelas (em OpenMP) do algoritmo de leilão, utilizando vetorização com instruções *intrinsics*. Resultados mostram que foi possível atingir um ganho de mais 10 vezes com a versão sequencial vetorizada e de até 200 vezes com a versão paralela vetorizada.

Este trabalho está dividido da seguinte maneira: a Seção 2 apresenta o algoritmo de leilão. Na Seção 3, a versão vetorizada proposta é descrita e avaliada. A implementação paralela utilizando a biblioteca OpenMP é apresentada e avaliada na Seção 4. Alguns trabalhos relacionados são descritos na Seção 5. Por fim, a Seção 6 apresenta as conclusões e possíveis trabalhos futuros.

2. Algoritmo de Leilão

Esta seção apresenta uma breve descrição sobre o Algoritmo de Leilão e sua implementação. O algoritmo de leilão clássico é definido como o problema de emparelhar m pessoas com n objetos supondo que há um benefício a_{ij} associado ao emparelhamento da pessoa i com o objeto j. Além disso, cada objeto tem um preço p_j associado e a pessoa que receber o objeto deve pagar seu preço p_j [Bertsekas 1979]. O valor associado ao objeto j pela pessoa i é $a_{ij} - p_j$ e cada pessoa i deseja ser associada ao objeto j_i de maior valor l_i (maior lance), conforme a Equação 1:

$$l_i = \max_{j=1,\ldots,n} \{a_{ij} - p_j\} \tag{1}$$

O Algoritmo de Leilão possui duas etapas principais: (1) OFERTA (Figura 1(a)), na qual as pessoas dão lances dinamicamente pelos objetos que desejam se associar, e (2) ASSOCIAÇÃO (Figura 1(b)), na qual o melhor lance recebido para cada objeto é selecionado individualmente, determinando suas associações e novos preços. O preço de cada objeto é aumentado de acordo com o valor do melhor lance por ele recebido.

O fato de os preços não serem reduzidos em nenhum momento do algoritmo garante que mesmo em casos de disputas de um conjunto de pessoas por um objeto específico, em algum momento o encarecimento de seu preço com o passar das interações

torna outros objetos mais e mais atrativos, de maneira que em algum momento alguma pessoa fará o lance final ao objeto disputado e a ele será associado enquanto que os demais terão perdido o interesse e passarão a investir em outros objetos [Bertsekas 1979].

O algoritmo itera em rodadas repetindo as duas etapas principais. Ao fim de cada rodada é produzido um conjunto de preços e associações (Figura 1(b), *linhas 5 e 6*). Se todas as pessoas estão satisfeitas com isso, ou seja, cada pessoa está associada a um objeto, o algoritmo termina. Caso contrário, cada pessoa livre i ofertará um novo lance para um objeto j, onde os lances são calculados como a diferença entre o objeto de maior e segundo maior interesse para uma determinada pessoa (Figura 1(a), *linhas 8 a 18*). É importante ressaltar que uma constante infinitesimal ϵ é adicionada ao valor do maior lance para tratar a convergência em casos de empate. Após a oferta de lances, um objeto j será então associado a pessoa i que ofereceu o maior valor associado (Figura 1(b), *linhas 9 a 16*).

Desta forma, enquanto existirem pessoas sem objetos associados, o leilão continua trocando as pessoas associadas aos objetos, e ajustando o preço de cada objeto j com os valores dos maiores lances atribuídos a eles (Figura 1(b), *linha 5*).

3. Vetorização do Algoritmo de Leilão

As máquinas *multicore*, além de vários núcleos de processamento, possuem instruções SIMD especializadas que podem aumentar consideravelmente o desempenho da aplicação quando uma mesma operação pode ser realizada em um conjunto de dados. O processo de conversão de uma implementação escalar de um programa (que realiza uma operação em um par de operandos por vez) para um processo vetorial (em que uma única instrução pode se referir a um vetor) é chamado de vetorização [Mark-Sabahi 2012]. Embora para alguns casos o próprio compilador seja capaz de vetorizar o programa do usuário, em geral, ele só é capaz de vetorizar códigos simples ou que estejam bem otimizados. Essa dificuldade acontece pois, para vetorizar um programa, o compilador tem que ser capaz de identificar a ausência de dependência de dados nas operações e que o acesso a memória seja contíguo [Mark-Sabahi 2012].

Assim, em muitos casos, o próprio programador tem que ser capaz de vetorizar o seu programa e, com isso, extrair todo o potencial de desempenho das instruções SIMD.

```
01 | ofertaLance(A, p, l, m, n) {
02 |   for (i = 0 ; i < m ; i++)
03 |     if pessoa i não possui objeto associado
04 |       (l[i].obj,l[i].val) = maiorSegMaior(A[i],p,n);
05 |     else
06 |       (l[i].obj,l[i].val) = (-1,-1);
07 | }
08 | maiorSegMaior(Ai, p, n) {
09 |   (mInd,mValor,smVal) = (0,(Ai[0]-p[0]),-1);
10 |   for (j = 1 ; j < n ; j++) {
11 |     vaux = Ai[j]-p[j];
12 |     if ( mVal < vaux )
13 |       (smVal,mVal,mInd) = (mVal,vaux,j);
14 |     else if ( vaux > smVal )
15 |       smVal = vaux;
16 |   }
17 |   return ( mInd, (mVal - smVal + ε) );
18 | }
```

(a) Oferta

```
01 | associacao(l, p, m, n) {
02 |   for ( j = 0 ; j < n ; j++ ){
03 |     (mVal,mInd) = maiorValParaObj(l,j,m);
04 |     if Existe um novo vencedor para o objeto j
05 |       p[j] = p[j] + mVal;
06 |       atualizaAssociacao(mVal,mInd);
07 |   }
08 | }
09 | maiorValParaObj(l, j, m) {
10 |   (mVal,mObj) = (-1,-1);
11 |   for ( i = 0; i < m ; i++ )
12 |     if ( l[i].obj == j )
13 |       if ( l[i].val > mVal )
14 |         (mVal,mInd) = (l[i].val,i);
15 |   return (mVal, mInd);
16 | }
```

(b) Associação

Figura 1. Algoritmo de leilão.

Esta seção descreve três versões vetorizadas para o algoritmo de leilão. Elas foram desenvolvidas utilizando funções *Intrinsics*, que permitem o acesso a diversas instruções SIMD específicas dos processadores Intel, sem a necessidade de escrever código *assembly* [Intel 2007, Intel 2017].

A Figura 2 provê uma visão simplificada da estratégia de vetorização utilizada para o algoritmo de leilão. Neste exemplo, o algoritmo original para obter o maior valor de um vetor e seu índice é apresentado na Figura 2(a). Uma vetorização inicial para esse algoritmo é detalhada na Figura 2(b), onde as operações de desvios são trocadas por operações lógicas e todas as operações são realizadas em um bloco de elementos simultaneamente (4, neste exemplo). Por fim, o algoritmo de uma versão vetorizada otimizada, que evita executar um conjunto de instruções quando todos os elementos do bloco são menores ou iguais aos maiores valores vistos até o momento pode ser visto na Figura 2(c).

Uma explicação detalhada de como funciona a vetorização é apresentada na Figura 2(d). Do lado esquerdo da figura se encontra uma simplificação do algoritmo vetorizado detalhado na Figura 2(b), para achar o maior valor e seu índice, considerando um vetor de 8 elementos inteiros de 4 bits. O lado direito da figura mostra o valor (em hexadecimal) das variáveis ao longo da execução do algoritmo. Para facilitar foi destacado em azul o que está sendo lido, em vermelho o que está sendo atribuído e em cinza escuro o que foi lido e atribuído para a mesma variável. A linha 1 do algoritmo carrega as 4 primeiras posições do vetor v na variável vetorizada `mVal`, enquanto que a linha 2 carrega os índices dessas posições na variável `mInd`. A linha 3 carrega as próximas 4 posições do vetor na variável `auxV` e a linha 5 os índices dessas posições na variável `auxI`. Por sua vez, a linha 4 compara `auxV` com `mVal` colocando na variável vetorizada `mask` o resultado dessa comparação que é F (15) quando os valores de `auxV` são maiores que os correspondentes em `maiorV` e 0 caso contrário. Na linha 6, `mVal` recebe os maiores valores, comparando cada posição de `auxV` com `mVal` através da função `MAX`. As linhas de 7 a 9 descrevem os passos para pegar os índices das variáveis `mInd` e `auxI`, de acordo os maiores valores de cada uma das variáveis `auxV` e `maiorV`. Por fim, a redução encontra o maior valor e seu índice percorrendo todos os elementos da variável vetorizada. É importante observar que a redução só é executada uma vez ao final do algoritmo, enquanto as etapas anteriores acontecem $m/chunk$ vezes onde m é o numero de objetos (colunas da matriz) e $chunk$ o número de elementos na variável vetorizada (4, neste exemplo).

Neste trabalho foram elaboradas 3 versões vetorizadas para o algoritmo de leilão:

- **simples**: Nesta versão foram vetorizadas as funções `maiorSegMaior` e `maiorValorParaObj` (Figura 1), eliminando todos os comandos `if` com operações lógicas e comparação com máscara, conforme o exemplo da Figura 2(b). Desta forma, é necessário executar sempre todas as instruções para os elementos do vetor.
- **cAssociação**: Esta versão é baseada na versão **simples**, onde é feita uma otimização utilizando instruções condicionais na função `maiorValorParaObj` (Figura 1(b)), conforme o exemplo da Figura 2(c). Desta forma, evita-se a execução de instruções quando o objeto j não for encontrado em um bloco de l (linha 12 da Figura 1(b)).
- **cCompleto**: Esta versão é baseada na versão **cAssociação**, onde é feita uma otimização utilizando instruções condicionais na função `maiorSegMaior` (Fi-

```
01 | mVal = Ai[0];
02 | mInd = 0;
03 | for (j=1 ; j<m ; j++) {
04 |   if (Ai[j] > mVal) {
05 |     mVal = Ai[j];
06 |     mInd = j;
07 |   }
08 | }
```

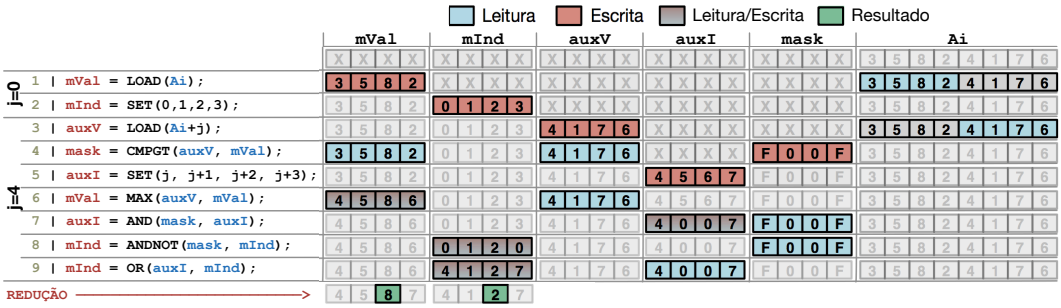
(a) Código original

```
01 | mVal = LOAD(Ai);
02 | mInd = SET(0,1,2,3);
03 | for (j=4 ; j<m ; j+=4) {
04 |   auxV = LOAD(Ai+j);
05 |   mask = CMPGT(auxV, mVal);
06 |   auxI = SET(i, j+1, j+2, j+3);
07 |   mVal = MAX(auxV, mVal);
08 |   auxI = AND(mask, auxI);
09 |   mInd = ANDNOT(mask, mIind);
10 |   mInd = OR(auxI, mInd);
11 | }
12 | REDUÇÃO
```

(b) Código vetorizado

```
01 | mVal = LOAD(Ai);
02 | mInd = SET(0,1,2,3);
03 | auxif = SET1(-1);
04 | for (j=4 ; j<m ; j+=4) {
05 |   auxV = LOAD(Ai+j);
06 |   mask = CMPGT(auxV, mVal);
07 |   cond = TESTZ(mask, auxif);
08 |   if (!cond) {
09 |     auxI = SET(i, j+1, j+2, j+3);
10 |     mVal = MAX(auxV, mVal);
11 |     auxI = AND(mask, auxI);
12 |     mInd = ANDNOT(mask, mIind);
13 |     mInd = OR(auxI, mInd);
14 |   }
15 | }
16 | REDUÇÃO
```

(c) Código vetorizado otimizado



(d) Execução vetorizada - baseada no código de (b)

Figura 2. Exemplo de vetorização para achar o maior elemento de um vetor e o seu índice.

gura 1(a)), conforme o exemplo da Figura 2(c). Neste caso são usadas duas instruções condicionais para evitar a execução de instruções quando nenhum valor de $vaux$ no bloco é maior do que os correspondentes em $mVal$, ou quando todos os valores de $vaux$ no bloco são maiores do que os correspondentes em $mVal$ (linha 12 da Figura 1(a)).

3.1. Análise Experimental

Todos os experimentos deste trabalho foram realizados em uma máquina *multicore* NUMA (*Non-Uniform Memory Access*) com 36 núcleos de processamento (2 *chips* Intel® Xeon® CPU E5-2699 v3 @ 2.30GHz) com 128 GB de RAM. Esse processador conta com instruções SIMD do tipo AVX2 de 256 bits. Todos os programas foram compilados utilizando o compilador icc da Intel com otimização -O3. No primeiro experimento foram utilizadas 35 matrizes reais distintas para o problema de emparelhamento de duas imagens. Para este problema os pontos da primeira imagem são representados nas linhas, enquanto que os pontos da segunda imagem nas colunas. Cada elemento da matriz representa a afinidade de um ponto da primeira imagem com um ponto da segunda imagem.

É importante ressaltar que, apesar do tempo para calcular o emparelhamento de algumas das matrizes da Tabela 1 ser muito pequeno, ele corresponde apenas ao tempo necessário para se emparelhar duas imagens. Para emparelhar uma sequência de imagens de um filme, por exemplo, o tempo pode aumentar consideravelmente. Neste caso, mesmo para matrizes pequenas, o uso do algoritmo de leilão vetorizado proposto é fundamental para se conseguir emparelhar sequências de imagens com um tempo de execução substancialmente menor. Como, para a maioria das instâncias, o tempo de execução é muito

pequeno, cada versão do algoritmo de leilão implementada para cada uma das matrizes foi executada 100 vezes e o coeficiente de variação foi calculado.

Tabela 1. Tempo de execução, coeficiente de variação e *speedup* das versões vetorizadas

Imagem	Características			Tempos (segundos)				Coef. de Variação (%)				*Speedup*		
	#Lin	#Col	#Iter	Orig	simp	cA	cC	Orig	simp	cA	cC	simp	cA	cC
alamo13	795	1241	404	0.6383	0.2362	0.0822	0.0851	8	16	16	13	2.70	**7.77**	7.50
alamo14	1241	1544	998	2.6583	0.8690	0.2807	0.2828	1	2	3	5	3.06	**9.47**	9.40
alamo16	1020	1241	964	1.7339	0.5902	0.2190	0.2145	3	5	13	14	2.94	7.92	**8.08**
alamo17	966	1241	925	1.5775	0.5303	0.1889	0.1903	5	7	16	15	2.97	**8.35**	8.29
alamo18	1069	3306	3097	15.3270	4.9661	1.5891	1.5219	2	10	1	0	3.09	9.65	**10.07**
bank	654	1418	134	0.2511	0.0737	0.0255	0.0057	14	13	12	11	3.41	**9.85**	9.77
bdom	1237	1470	841	2.1164	0.6814	0.2113	0.2109	0	3	3	3	3.11	10.01	**10.03**
cars	187	730	110	0.0368	0.0118	0.0059	0.0059	7	10	11	11	3.13	**6.19**	**6.19**
chinesebuilding	861	1683	224	0.4964	0.1806	0.0652	0.0647	7	14	14	14	2.75	7.62	**7.67**
eifell	387	1917	121	0.1882	0.0561	0.0239	0.0231	13	14	10	11	3.35	7.87	**8.14**
essighaus	951	898	3698	4.8937	1.6273	0.5866	0.5734	5	3	1	3	3.01	8.34	**8.53**
grafitti	1559	1641	1775	6.1955	1.9848	0.5845	0.5812	0	0	0	0	3.12	10.60	**10.66**
londonbridge	1219	1332	2472	5.4977	1.7815	0.5552	0.5432	0	1	1	2	3.09	9.90	**10.12**
madrid	1159	1360	671	1.4950	0.4964	0.1679	0.1696	2	3	9	9	3.01	**8.91**	8.82
metz	1286	1773	460	1.4388	0.4594	0.1378	0.1367	3	0	0	0	3.13	10.44	**10.52**
miduomo	719	1143	1374	1.6831	0.5829	0.2168	0.2145	3	9	16	16	2.89	7.76	**7.85**
miduomo02	2932	4369	1081	18.8940	6.0137	1.6779	1.6414	0	0	0	0	3.14	11.26	**11.51**
montreal	991	1193	1173	1.9659	0.6661	0.2406	0.2437	3	6	14	13	2.95	**8.17**	8.07
neubrandeburg	2142	3526	2380	24.4941	7.8014	2.2193	2.1722	0	0	0	0	3.14	11.04	**11.28**
notredame	3026	3297	2087	28.2164	8.9569	2.4323	2.4014	0	0	0	0	3.15	11.60	**11.75**
notredame12	806	1246	176	0.3255	0.1020	0.0327	0.0330	15	15	14	14	3.19	**9.94**	9.85
notredame16	845	1246	235	0.4233	0.1433	0.0469	0.0469	10	15	11	12	2.95	**9.02**	**9.02**
pantheon	1339	3306	288	1.7659	0.5711	0.1802	0.1736	2	0	5	2	3.09	9.80	**10.17**
pantheon2	2083	4195	1093	12.9919	4.1978	1.1913	1.1644	0	3	0	0	3.09	10.90	**11.16**
portcullis	907	1411	211	0.4371	0.1563	0.0512	0.0513	9	15	13	13	2.80	**8.54**	8.52
postoffice	837	1189	468	0.7249	0.2729	0.0952	0.0908	7	15	13	15	2.66	7.62	**7.98**
riga	963	1683	642	1.4520	0.4891	0.1732	0.1721	1	3	9	8	2.97	8.38	**8.44**
sanmarco	851	934	2014	2.3241	0.7918	0.3084	0.3060	3	7	13	13	2.94	7.54	**7.60**
sanmarco2	2454	3213	501	5.3515	1.7000	0.4690	0.4654	1	1	0	0	3.15	11.41	**11.50**
startgarder	3044	4339	786	14.0710	4.4684	1.2225	1.2126	0	0	0	2	3.15	11.51	**11.60**
startgarder3	4028	4484	3004	73.6040	23.4472	6.3750	6.2465	0	1	0	0	3.14	11.54	**11.78**
taj	528	1122	546	0.6012	0.2000	0.0731	0.0723	8	11	9	11	3.01	8.22	**8.32**
tavern	1286	2719	134	0.6443	0.2080	0.0637	0.0633	0	3	0	1	3.10	10.11	**10.19**
townsquare	1004	1112	1479	2.3336	0.7959	0.2914	0.2871	2	4	11	12	2.93	8.01	**8.13**
worldbuilding	1268	2067	438	1.5835	0.5017	0.1501	0.1484	2	0	0	1	3.16	10.55	**10.67**

Os resultados para as três versões vetorizadas, descritas na Seção 3, assim como o algoritmo de leilão sequencial original (Seção 2) podem ser vistos na Tabela 1. Para cada uma das matrizes (cada linha da tabela) são apresentados o nome da matriz e a sua quantidade de linhas, colunas e iterações, assim como, a média do tempo de execução, o coeficiente de variação e o *speedup* das versões vetorizadas em relação a versão sequencial original. Os maiores *speedups* para cada uma das linhas da tabela estão destacados em negrito.

O tempo para executar cada imagem é diretamente proporcional ao tamanho da matriz de pontos e a quantidade de iterações para achar o emparelhamento ótimo, uma vez que o algoritmo de leilão, a cada iteração, percorre toda a matriz. As versões vetorizadas tiveram um desempenho muito superior a versão original, aproveitando o grande potencial das instruções SIMD. A média dos tempos de execução da versão **simples** (**simp**), considerando todas as matrizes, foi aproximadamente três vezes menor do que da versão original (**Orig**), enquanto que as médias das versões **cAssociação** (**cA**) e **cCompleto** (**cC**) foram mais de 9 vezes menor. Para aproximadamente metade das matrizes, os coeficientes

de variação foram bem pequenos (abaixo de 10%), para outra metade, onde os tempos de execução são significativamente menores, o coeficiente de variação foi um pouco maior (abaixo de 16%). É importante ressaltar que para as maiores instâncias (onde o tempo de execução do algoritmo original foi maior que 5 segundos) o maior coeficiente de variação encontrado foi 3%, sendo que a maior parte ficou menor ou igual a 1%.

Analisando os melhores *speedups*, que estão destacados em negrito, fica claro que, para a maioria das matrizes, a versão **cCompleto** produziu os melhores desempenhos. Praticamente em todos os casos que **cAssociação** apresentou o maior *speedup*, a diferença foi muito pequena em relação a **cCompleto** e o coeficiente de variação muito alto, o que mostra uma grande variação nos tempos de execução para essas matrizes. Desse modo, a próxima seção irá analisar o desempenho apenas das implementações paralelas baseadas no algoritmo original e na versão vetorizada **cCompleto**.

4. Paralelização do Algoritmo de Leilão para Arquiteturas Multicore

O presente trabalho tem o objetivo de aproveitar todo potencial das arquiteturas *multicore*. Assim, além da vetorização que maximiza o uso das instruções SIMD, é necessário utilizar os múltiplos núcleos de processamento de maneira eficiente, através de técnicas de paralelização baseadas em memória compartilhada. Para isso, foi utilizado o modelo de programação paralela com memória compartilhada *OpenMP* [Chandra et al. 2000].

Como descrito na Seção 2, o algoritmo de leilão é basicamente uma sucessão de iterações, onde em cada uma dessas iterações são executadas as fases de OFERTA e ASSOCIAÇÃO. Na fase de OFERTA, o cálculo do lance dado por cada pessoa para um objeto não depende do lance de outra pessoa. Assim, cada lance pode ser realizado em paralelo, bastando para isso distribuir as m iterações do laço da Linha 02 da função `ofertaLance` (Figura 1(a)) entre as p *threads* disponíveis, através de uma diretiva `parallel for` do openMP.

Por sua vez, na fase de ASSOCIAÇÃO cada objeto seleciona o melhor lance recebido e determina sua associação e novo preço, sem depender das seleções dos outros objetos. Assim, o código OpenMP da fase de ASSOCIAÇÃO é simplesmente distribuir as n iterações do laço da Linha 02 da função `associacao` (Figura 1(b)) entre as p *threads* disponíveis, através de uma diretiva `parallel for` do openMP.

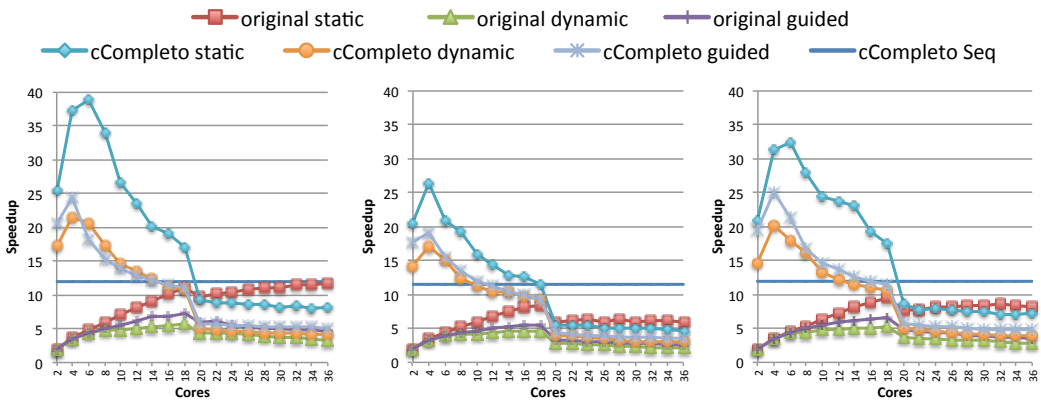
4.1. Análise Experimental

Os experimentos para avaliar as versão paralelas utilizaram o mesmo ambiente descrito na Subseção 3.1. Inicialmente, foram executadas as três matrizes com os maiores tempo de execução da Tabela 1 (*startgarder3*, *neubrandeburg* e *notredame*), variando a quantidade de *threads* OpenMP de 2 até 36. Além disso, dois outros parâmetros de execução foram avaliados: **escalonamento de laços** e **afinidade com o processador**. O **escalonamento de laços** do OpenMP permite uma série de opções de como dividir as tarefas do laço paralelizado entre as *threads*. Foram avaliadas as opções: *static*, *dynamic* e *guided*. A opção *static* divide todas as iterações do laço em frações de mesmo tamanho entre as *threads*. Já, as opções *dynamic* e *guided* dinamicamente vão atribuindo um conjunto de iterações para as *threads*. Enquanto a opção *dynamic* distribui uma iteração por vez, a opção *guided* inicia distribuindo frações grandes para cada *thread* que vão diminuindo durante a execução. Por sua vez, a **afinidade com o processador** pode ser habilitada para

determinar o conjunto de *cores* onde as *threads* do programa poderão ser escalonadas. O uso da afinidade pode ser interessante para evitar o uso de *Hyper-threading* e para mitigar custos com acesso à memória (NUMA).

O *speedup* para cada um dos cenários avaliados, calculado a partir da média de 5 execuções (o coeficiente de variação foi na média abaixo de 2% e não ultrapassou 5%), pode ser visto na Figura 3. Foram exibidos apenas os resultados com afinidade, visto que o desempenho das versões sem afinidade foi de aproximadamente metade do que o da versão com afinidade. A razão para tal comportamento se deve principalmente ao fato de que entradas pequenas tiveram uma grande redução do tempo de execução para as versões vetorizadas, resultando em baixa granularidade. Sendo assim, os custos de acesso à memória da arquitetura NUMA se tornaram mais evidentes, ao usar desnecessariamente *cores* dos dois *chips*. Para os demais experimentos, optamos por não executar os experimentos sem afinidade.

Com relação as políticas de escalonamento de laços, a opção *static* foi a que produziu os melhores resultados tanto para versão paralela original como também para a versão paralela vetorizada (*cCompleto*). Apesar da execução não ser uniforme, ou seja, a oferta de lances das pessoas serem espalhadas ao longo das linhas da matriz e a associação dos objetos as pessoas também serem espalhadas, isso não causa um desbalanceamento suficiente para que as políticas de escalonamento dinâmicas tirem proveito. Assim, a baixa sobrecarga da política estática produziu os melhores resultados.



(a) startgarder3 com afinidade (b) neubrandenburg com afinidade (c) notredame com afinidade

Figura 3. *Speedup* das versões paralelas vetorizada e original

Ao se analisar a escalabilidade é possível verificar que a versão paralela original apresentou ganho de desempenho a medida que a quantidade de *cores* aumenta até o limite de 18 cores. Especificamente para a matriz *startgarder3*, a maior das três, também houve melhora de desempenho com mais de 30 cores. Porém, para todos os cenários, o desempenho da versão paralela original foi pior do que a versão vetorizada sequencial (*cCompleto seq*). Por sua vez, a versão paralela vetorizada só escalou até 6 cores, sendo que para a menor matriz (*neubrandeburg*) somente até 4 cores. A razão para baixa escalabilidade é a granularidade fina das tarefas em função do excelente desempenho da versão vetorizada sequencial. Por exemplo, o tempo total de execução da versão original sequencial para a matriz *startgarder3* é de 73,6 segundos, enquanto que o tempo da versão

vetorizada sequencial é de apenas $6,2$. Por sua vez, tempo da execução paralela com 6 cores foi de apenas $1,92$ segundos para executar as 3004 iterações, ou seja, um tempo de aproximadamente $0,00063$ segundos por iteração. Assim, como o paralelismo ocorre a cada iteração do algoritmo, fica evidente a granularidade fina das tarefas nesse ponto, limitando sua escalabilidade. Mesmo assim, é importante destacar que para essa matriz a versão paralela vetorizada foi quase 40 vezes mais rápida que a versão original sequencial, utilizando apenas 6 *cores*. Para a menor matriz (*neubrandeburg*) o desempenho foi 26 vezes melhor utilizando apenas 4 *cores*.

É possível observar uma queda brusca de desempenho de 18 para 20 cores em todos os cenários apresentados na Figura 3. A explicação para tal comportamento é a sobrecarga no acesso a memória em função da arquitetura NUMA. Para até 18 cores, apenas um *chip* (processador) da máquina é utilizado, o que garante o acesso a memória próxima a ele. Por sua vez, com 20 ou mais cores pelo menos uma das *threads* vai ter que acessar a memória mais distante (acesso mais custoso), aumentando a sobrecarga.

O próximo experimento avalia as versões paralelas para matrizes maiores. Para isso foi utilizada uma aplicação que simula o movimento de partículas, ao longo do tempo, dentro de um ambiente 3D com diferentes velocidades, perturbadas por ruídos aleatórios. Essa aplicação permite criar casos de estudo com maior concorrência ou maior esparsidade pela escolha do número de partículas e definição da caixa envolvente. Dessa forma, permite criar matrizes de custo/benefício com diferentes características de alocação. Foram geradas matrizes de custo/benefício com diferentes tamanhos e quantidades de iterações e o emparelhamento dessas partículas, entre dois instantes de tempo, foi realizado utilizando as versões paralela original e vetorizada.

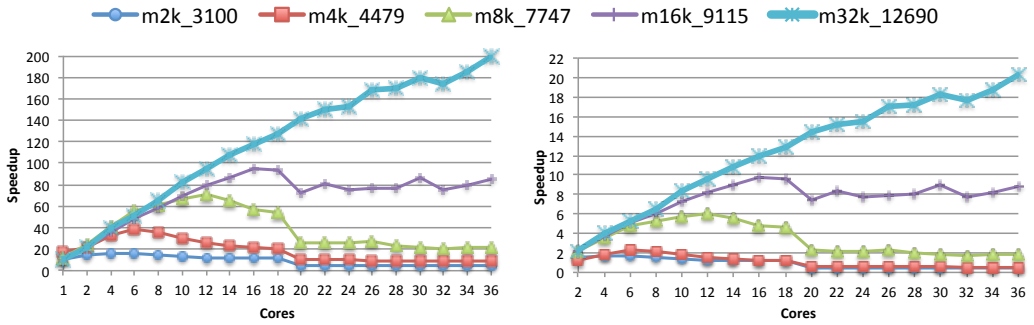
A Figura 4 apresenta os gráficos das execuções da versão paralela **cCompleto** com políticas de escalonamento de laços *static*, para as seguintes matrizes de entrada com diferentes tamanhos e iterações: 2000×2000 elementos e 3100 iterações (m2k_3100); 4000×4000 elementos e 3100 iterações (m4k_4479); 8000×8000 elementos e 7747 iterações (m8k_7747); 16000×16000 elementos e 9115 iterações (m16k_9115); 32000×32000 elementos e 12690 iterações (m32k_12690). As Figuras 4(a) e 4(b) mostram os *speedups* relativos a versão sequencial original e sequencial vetorizada, respectivamente.

As duas figuras mostram claramente o aumento da escalabilidade com o aumento do tamanho das matrizes de entrada. Por exemplo, como mostra a Figura 4(b), a entrada m32k_12690 atinge *speedup* de $20,2$ para 32 *cores*, enquanto que a entrada m2k_3100 apresenta *speedup* de apenas $1,2$ para o mesmo cenário. Como já destacado anteriormente, o motivo da baixa escalabilidade das matrizes pequenas é a granularidade fina das tarefas, principalmente em função do excelente desempenho da versão vetorizada sequencial. Assim, é esperado que para matrizes grandes todo o potencial das arquiteturas *multicore* possa ser aproveitado, mesmo considerando a sobrecarga de acesso a memória das arquiteturas NUMA.

Já, a Figura 4(a) destaca o excelente desempenho alcançado, em função da combinação da vetorização, que permite o uso eficiente das instruções SIMD, com o paralelismo, que permite uso dos múltiplos núcleos de processamento disponíveis nas arquiteturas *multicore*, atingindo o patamar de 200 de *speedup* para a entrada m32k_12690, executando em 36 *cores*, o que significou uma redução de mais 5 horas de execução com

a versão sequencial original para apenas $90,7$ segundos, com a versão vetorizada paralela.

Uma característica interessante é que, de maneira geral, a maioria das associações ocorrem nas primeiras iterações (por exemplo, para todas as matrizes apresentadas mais de 90% das associações ocorreram antes de atingir 500 iterações). Isso faz com que as iterações intermediárias e mais próximas do fim executem menos processamento, pois a maioria dos objetos já está associada, restando poucas pessoas disputando poucos objetos. Experimentos iniciais mostraram que é possível ajustar durante a execução a quantidade de *threads* para tirar proveito deste comportamento. Mais especificamente, foi verificado que, para a matriz m16k_9115, ao se iniciar a execução com 18 *threads* e, a partir do momento que mais de 90% dos objetos forem associados, reduzir a quantidade de *threads* para 16, diminui o tempo de execução.



(a) *speedups* em relação ao sequencial original (b) *speedups* em relação ao sequencial vetorizado

Figura 4. Execução Paralela para Matrizes de Emparelhamento de Partículas

5. Trabalhos Relacionados

O algoritmo de leilão para o problema de emparelhamento já foi cuidadosamente analisado em [Bertsekas 1979, Bertsekas 1992], onde foi demonstrado a importância do incremento ϵ-scaling que garante a otimalidade do algoritmo. Além disso, implementações paralelas síncronas e assíncronas do algoritmo de leilão foram propostas e executadas em uma máquina paralela real em [Bertsekas and Castañon 1991].

O trabalho em [Buš and Tvrdík 2009] introduziu a estratégia, chamada de *look-back*, que estende a implementação clássica com a habilidade de reutilizar informação de lances anteriores. O artigo mostra que é possível utlizar lances anteriores eficientemente, embora o algoritmo falhe em quase 30% para um dos tipos das entradas testadas.

Um algoritmo de leilão paralelo capaz de emparelhar grandes grafos bipartidos densos e esparsos foi apresentado em [Sathe et al. 2012], propondo uma nova estratégia ϵ-scaling. Resultados experimentais em um supercomputador Cray XE6 mostram que a implementação híbrida MPI−OpenMP reduziu drasticamente o tempo de execução, embora nenhuma análise do desempenho tenha sido apresentada.

O problema de alinhamento de grafo global em *clusters* de alto desempenho foi apresentado em [Kollias et al. 2014]. Este trabalho não só acha os vértices similares através do uso do algoritmo de leilão para emparelhamento de grafos bipartidos, mas, diferentemente dos outros trabalhos apresentados nesta seção, previamente computa a

matriz de similaridades. Matrizes contendo de 5.000 até $1.500.000$ vértices foram executadas em um supercomputador Cray XE6 eficientemente, especialmente para as matrizes grandes.

O trabalho apresentado em [Nascimento et al. 2016] analisa como o tamanho da matriz e quantidade de iterações influenciam no tempo de execução de uma versão híbrida (OpenMP/MPI) do algoritmo de leilão. Uma importante contribuição foi mostrar o custo de comunicação entre os nós do *cluster*, a cada iteração, e como isso afeta o desempenho do algoritmo de leilão.

Diferentemente de todos os trabalhos descritos nesta seção, este trabalho implementa e avalia versões sequenciais vetorizadas que tiram proveito das instruções SIMD disponíveis nas novas arquiteturas. Além disso, também implementa e avalia versões paralelas com a biblioteca OpenMP que mostram que o uso eficiente das instruções SIMD em conjunto com os processadores (*cores*) potencializam o ganho de desempenho nas novas arquiteturas *multicore*.

6. Conclusões e Trabalhos Futuros

A motivação prática para a adoção do algoritmo de leilão na resolução do problema de emparelhamento de grafos bipartidos é a existência de cenários onde o tamanho do problema é altamente proibitivo, tornando sua natureza distributiva uma formulação valiosa para o uso do paralelismo.

Este trabalho implementou e avaliou versões vetorizadas do algoritmo de leilão, assim como, implementações paralelas utilizando memória compartilhada. Os resultados mostram que é possível aproveitar todo potencial das novas arquiteturas *multicore*. Enquanto que a versão vetorizada foi em média 10 vezes mais rápida que a versão sequencial, sendo suficiente para resolver problemas pequenos muito rapidamente, a versão paralela vetorizada atingiu 200 de *speedup*, se mostrando como uma excelente alternativa para execução de matrizes médias e grandes em arquiteturas *multicore*.

Trabalhos futuros irão avaliar técnicas para variar a quantidade de unidades de processamento a serem utilizadas ao longo da execução, uma vez que a quantidade de trabalho a ser realizada diminui com o número de iterações. Além disso, otimizações para melhorar a sobrecarga de acesso a memória NUMA serão investigadas com o objetivo de aumentar ainda mais a escalabilidade do algoritmo de leilão nas máquinas *multicore*.

Agradecimentos

À FAPERJ, ao CNPq e à CAPES pelo apoio dado aos autores deste trabalho. Os autores também agradecem o uso dos recursos computacionais *manycore* mantidos e operados pelo Núcleo de Computação Científica da Universidade Estadual Paulista (NCC/UNESP), financiado parcialmente pela Intel, no contexto do projeto Intel/UNESP Modern Code.

Referências

Bertsekas, D. P. (1979). A distributed algorithm for the assignment problem. Technical report, Lab. for Information and Decision Systems, M.I.T., Cambridge, MA.

Bertsekas, D. P. (1992). Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications*, 1(1):7–66.

Bertsekas, D. P. and Castañon, D. A. (1991). Parallel synchronous and asynchronous implementations of the auction algorithm. *Parallel Comput.*, 17(6-7):707–732.

Buš, L. and Tvrdík, P. (2009). Towards auction algorithms for large dense assignment problems. *Computational Optimization and Applications*, 43(3):411–436.

Carpaneto, G., Martello, S., and Toth, P. (1988). Algorithms and codes for the assignment problem. *Annals of Operations Research*, 13(1):191–223.

Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., and McDonald, J. (2000). *Parallel Programming in OpenMP*. Morgan Kaufmann, 1st edition.

Intel (2007). Intel c++ intrinsic reference. Technical report, Intel.

Intel (2017). Intel intrinsics guide. Technical report, Intel.

Kollias, G., Sathe, M., Mohammadi, S., and Grama, A. (2013). A fast approach to global alignment of protein-protein interaction networks. *BMC Research Notes*, 6(1):1–11.

Kollias, G., Sathe, M., Schenk, O., and Grama, A. (2012). Fast parallel algorithms for graph similarity and matching. Technical report RR12-010, Department of Computer Science, Purdue University.

Kollias, G., Sathe, M., Schenk, O., and Grama, A. (2014). Fast parallel algorithms for graph similarity and matching. *Journal of Parallel and Distributed Computing*, 74(5):2400 – 2410.

Mark-Sabahi (2012). A guide to auto-vectorization with intel c++ compilers. Technical report, Intel.

Nascimento, A. P., Vasconcelos, C. N., Jamel, F. S., and Sena, A. C. (2016). A hybrid parallel algorithm for the auction algorithm in multicore systems. In *Inter. Symp. on Computer Architecture and High Perf. Comp. Workshops (SBAC-PADW)*, pages 73–78.

Sathe, M., Schenk, O., and Burkhart, H. (2012). An auction-based weighted matching implementation on massively parallel architectures. *Parallel Computing*, 38(12):595 – 614.

Shokoufandeh, A. and Dickinson, S. (1999). Applications of bipartite matching to problems in object recognition. In *In Proceedings, ICCV Workshop on Graph Algorithms and Computer Vision*, page http://www.cs.cornel.

Vasconcelos, C. N. and Rosenhahn, B. (2009). *Bipartite Graph Matching Computation on GPU*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Implementação e Avaliação de Técnicas de Paralelização no Algoritmo de Hirschberg para Sistemas Multicore

Mario João Jr.[1], **Alexandre C. Sena**[2] **e Vinod E. F. Rebello**[3]

[1] Laboratório Médico de Pesquisas Avançadas, UERJ – Rio de Janeiro – RJ – Brasil

[2]Instituto de Matemática e Estatística, UERJ – Rio de Janeiro – RJ – Brasil

[3]Instituto de Computação, UFF – Niterói – RJ – Brasil

junior@lampada.uerj.br, asena@ime.uerj.br, vinod@ic.uff.br

Resumo. *Descobrir a maior subsequência comum entre duas sequências em um tempo razoável é fundamental para solucionar diversos problemas. Para garantir que a solução ótima seja encontrada, algoritmos baseados em programação dinâmica são necessários. O algoritmo de Hirschberg possui complexidade linear de espaço, podendo ser usado para comparar sequências longas. Porém, devido à sua complexidade quadrática de tempo, o uso do paralelismo é fundamental. Assim, o objetivo deste trabalho é implementar e avaliar técnicas de paralelismo para o algoritmo de Hirschberg que permitam a comparação de sequências de caracteres longas. Para alcançar este objetivo, três estratégias de paralelismos são implementadas e investigadas em cima de melhorias na versão sequencial do algoritmo. Os resultados mostram que é possível executar mais eficientemente o algoritmo de Hirschberg em máquinas* multicore*, especialmente para grandes cadeias de caracteres, sendo possível alcançar um desempenho até* 33 *vezes melhor do que a versão sequencial original.*

1. Introdução

Encontrar a maior subsequência comum (*Longest Common Subsequence - LCS*) entre duas cadeias de caracteres em um tempo aceitável é fundamental para várias aplicações, como por exemplo, comparação de sequências de nucleotídeos ou proteínas para investigar a similaridade entre espécies, comparação de arquivos para identificar as diferenças entre os mesmos, reconhecimento de padrões, entre outras [Gusfield 1997]. Embora existam ferramentas heurísticas para achar a maior subsequência comum entre duas cadeias, as mesmas não garantem o resultado ótimo, podendo conter erros não desprezíveis [Martins et al. 2001]. Assim, nos casos onde seja necessário achar o valor exato, algoritmos que garantem a solução ótima propostos por Needleman e Wunsch [Needleman and Wunsch 1970] ou Hirschberg [Hirschberg 1975] são necessários. Apesar desses dois algoritmos serem baseados em programação dinâmica, o algoritmo de Needleman e Wunsch possui complexidade $O(m \times n)$ em tempo e espaço, assumindo duas cadeias de tamanhos m e n com $n \geq m$, enquanto o algoritmo de Hirschberg possui também complexidade de tempo $O(m \times n)$, mas complexidade de espaço linear $O(n)$, sendo necessário manter apenas uma linha da matriz de similaridade em memória.

Um algoritmo de ordem quadrática no espaço limita bastante o tamanho das cadeias a serem comparadas por sistemas computacionais. Por exemplo, para analisar

duas cadeias de 100.000 caracteres (onde são necessários 4 *bytes* para armazenar os dados relativos ao cálculo para cada caracter) são utilizados aproximadamente 40 GB de espaço de armazenamento apenas para guardar a matriz de similaridade. Por ser linear no espaço, o algoritmo de Hirschberg não possui essa limitação, podendo ser utilizado para cadeias com milhões de caracteres. Porém, por possuir complexidade de tempo quadrática, o tempo para comparar sequências grandes é bastante elevado, sendo necessário computação de alto desempenho para se conseguir comparar grandes cadeias em tempos aceitáveis para os usuários.

Com o fim da era dos processadores baseados em uma única unidade de processamento, em função do problema da dissipação de calor e consumo de energia, todo computador passou a ser uma máquina paralela [Diaz et al. 2012]. Assim, para aproveitar todo potencial dessas novas arquiteturas paralelas com memória compartilhada é necessário desenvolver aplicações paralelas eficientes. Desse modo, o objetivo deste trabalho é implementar e avaliar técnicas de paralelismo para o algoritmo de Hirschberg, de maneira a permitir a comparação de grandes cadeias de caracteres em tempos aceitáveis. Mais especificamente, três abordagens para arquiteturas *multicore* com memória compartilhada são investigadas e implementadas, assim como melhorias para aumento de desempenho da versão sequencial. Os resultados mostram que é possível executar eficientemente o algoritmo de Hirschberg nessas arquiteturas e, com isso, diminuir bastante o tempo necessário para se conseguir achar a maior subsequência comum exata entre duas cadeias. Os resultados mostram que para cadeias muito longas foi possível atingir um desempenho 33 vezes melhor ao se comparar a versão paralela otimizada com a versão sequencial original.

O restante deste artigo está dividido da seguinte maneira: na Seção 2 são apresentados alguns trabalhos relacionados enquanto a Seção 3 apresenta o algoritmo de Hirschberg. As estratégias para aumentar o desempenho da versão sequencial são apresentadas na Seção 4. Por sua vez, a Seção 5 descreve as três abordagens paralelas investigadas e seus desempenhos. Por fim, as conclusões e trabalhos futuros são apresentados na Seção 6.

2. Trabalhos Relacionados

Encontrar a maior subsequência comum é uma etapa fundamental para o alinhamento de cadeias de nucleotídeos e proteínas e tem sido objeto de estudo desde a década de 70, quando foram propostos os algoritmos exatos de Needleman e Wunsch [Needleman and Wunsch 1970] e Hirschberg [Hirschberg 1975]. Uma revisão sobre os principais algoritmos exatos para o alinhamento de cadeias par a par pode ser visto em [Sandes et al. 2016]. Além de descrever e classificar os algoritmos, o artigo descreve os principais avanços, as arquiteturas em que os mesmos foram executados e o tamanho das cadeias alinhadas.

Uma implementação paralela em dois níveis do algoritmo de Hirschberg para busca por sequência de proteínas homólogas (sequências que compartilham um ancestral comum) foi proposta em [Rashid et al. 2007]. No primeiro nível de paralelismo (granularidade grossa), composto de um banco de dados de sequências a serem comparadas, foi utilizado MPI (*Message Passing Interface*) para distribuir essas sequências entre os processadores. Por sua vez, no segundo nível de paralelismo (granularidade fina), cada

instância do algoritmo é paralelizada utilizando Pthreads. Em razão das dependências da matriz de similaridades, a paralelização é bastante trivial e limitada. Os autores simplesmente dividiram a matriz em dois blocos que são computados distintamente. O primeiro bloco, composto pela metade superior da matriz, é computado a partir da primeira linha e coluna da matriz. Por sua vez, no segundo bloco, composto pela metade inferior da matriz, é computado a partir da última linha e coluna da matriz. É importante notar que o segundo nível de paralelismo fica limitado a um *speedup* de no máximo 2.

Apesar do trabalho proposto em [Driga et al. 2003] também apresentar um algoritmo exato com complexidade de espaço linear, na prática, ao invés de manter uma linha da matriz por vez em memória, o algoritmo utiliza um parâmetro k que deve ser informado pelo usuário e, de acordo com os experimentos apresentados, varia em função do tamanho da cadeia e da quantidade de processadores. Assim, diferentemente do algoritmo de Hirschberg que mantêm apenas duas linhas da matriz durante toda a execução, este algoritmo necessita de uma quantidade de espaço muito maior, sendo necessário manter k linhas e colunas, assim como, quadrantes de tamanho n/k, onde n é o tamanho da cadeia. Para melhorar o desempenho do algoritmo os autores utilizam o tradicional paralelismo *wavefront* [Anvik et al. 2002] [Mohanty and Cole 2014], que permite executar os elementos das antidiagonais de cada bloco da matriz em paralelo.

Diferentemente dos trabalhos apresentados anteriormente nesta seção, o trabalho apresentado neste artigo implementa e avalia técnicas para melhorar o desempenho do algoritmo de Hirschberg que permitam sua execução para cadeias contendo milhões de caracteres em um tempo aceitável. Enquanto que a paralelização dos algoritmos exatos que usam a matriz de similaridades é tradicionalmente feita através da técnica de *wavefront* [Mohanty and Cole 2014], no algoritmo de Hirschberg a mesma não pode ser usada, uma vez que apenas uma linha da matriz é computada por vez para se conseguir uma complexidade de espaço linear. Os resultados obtidos mostram que a execução com as melhorias propostas reduziram o tempo significantemente, principalmente para grandes sequências.

3. O Algoritmo de Hirschberg

Em [Hirschberg 1975], o autor desenvolveu um algoritmo, que recebeu seu nome, para a descoberta da maior subsequência comum (*LCS - Longest Common Sequence*). Tal algoritmo, baseado em programação dinâmica, se preocupa em resolver o problema em $O(m \times n)$ em tempo e $O(n)$ em memória, onde m e n são os tamanhos das respectivas sequências com $n \geq m$.

3.1. O Algoritmo

Uma descrição do algoritmo de Hirschberg pode ser vista no Algoritmo 1. O *Caso Trivial* (linhas 2 a 14) se dá quando uma das sequências é vazia ou quando a primeira sequência possui apenas um elemento.

Se não for um *Caso Trivial*, são gerados dois vetores L_1 e L_2 com os pesos das similaridades (linhas 15 a 17), uma vez que a comparação das sequências A e B é realizada em duas etapas. O primeiro vetor é gerado a partir da comparação da sequência B com a metade superior da sequência A, enquanto que o segundo vetor a partir da comparação da sequência B com a metade inferior da sequência A. O algoritmo utilizado para a

Algoritmo 1: Algoritmo de Hirschberg

```
1  Função Hirschberg (m, n, A, B, C)
       /* Caso Trivial */
2      se n = 0 então
3          C ← ∅;
4          retorna;
5      senão
6          se m = 1 então
7              se ∃j ⩽ n tal que A[1] = B[j] então
8                  C ← A[1];
9              senão
10                 C ← ∅;
11             fim
12             retorna;
13         fim
14     fim
       /* Geração dos vetores */
15     i ← ⌊m/2⌋;
16     Geravet (i, n, A[0..i − 1], B, L1);
17     Geravetinv (m − i, n, A[i..m], B, L2);
       /* Ponto de Divisão Vertical */
18     M ← max(L1[j], L2[n − j]), para 0 ⩽ j ⩽ n;
19     k ← min(j), tal que L1[j] + L2[n − j] = M;
       /* Divisão e Conquista */
20     Hirschberg (i, k, A[0..i], B[0..k], C1);
21     Hirschberg (m − i, n − k, A[i + 1..m], B[k + 1..n], C2);
       /* Retorno */
22     C ← concatena(C1, C2);
```

comparação é o mesmo, mas para a metade inferior da sequência A, as sequências são percorridas do fim para o início.

Tendo os vetores L_1 e L_2, é encontrado k (linhas 18 e 19) para que seja feita a divisão e conquista (linhas 20 e 21) e ao final retornada a LCS (linha 22) como a concatenação de C_1 e C_2.

3.2. Geração dos Vetores de Similaridade

O Algoritmo de Hirschberg gera os vetores de similaridade conforme o Algoritmo 2. Dessa forma, não há necessidade de gerar uma matriz para poder armazenar as similaridades como em [Needleman and Wunsch 1970] [Smith and Waterman 1981], o que faz com que sua complexidade de memória seja linear. Cada elemento j de K_1 pode ser calculado da seguinte forma:

Algoritmo 2: Geração dos Vetores de Similaridade

```
1  Função Geravet (m, n, A, B, LL)
2      K1[j] ← 0, ∀j 0 ⩽ j ⩽ n;
3      para 1 ⩽ i ⩽ m faça
4          K0[j] ← K1[j], ∀j 0 ⩽ j ⩽ n;
5          para 1 ⩽ j ⩽ n faça
6              se A[i] = B[j] então
7                  K1[j] ← K0[j − 1] + 1;
8              senão
9                  K1[j] ← max(K1[j − 1], K0[j]);
10             fim
11         fim
12         LL ← K1;
13     fim
```

$$K_1[j] = \begin{cases} K_0[j-1] + 1, & \text{se A[i] = B[j]} \\ & \text{ou} \\ max(K_1[j-1], K_0[j]), & \text{caso contrário} \end{cases}$$

4. Aprimoramento do Desempenho da Versão Sequencial

Antes da utilização de técnicas de paralelização para o algoritmo de Hirschberg, ainda cabem melhorias para o aprimoramento do desempenho na sua versão sequencial. Tais melhorias são abordadas nas subseções a seguir. A saber: duas alterações no algoritmo original e a utilização das instruções SIMD (*Single Instruction, Multiple Data*) disponíveis em processadores modernos. A seção termina com a análise experimental das melhorias supramencionadas.

4.1. Remoção da Cópia dos Vetores de Similaridade

Na linha 4 do Algoritmo 2 observa-se a cópia de K_1 para K_0, uma vez que para gerar o novo K_1 o algoritmo necessita do vetor anterior. Tal cópia foi removida e o algoritmo foi alterado de forma que as iterações ímpares do loop iniciado na linha 3 utilizam K_0 para gerar K_1 e as pares utilizam K_1 para gerar K_0. Devido a remoção da cópia, ainda foi necessária a alteração na inicialização (linha 2), onde K_0 passa a ser inicializado e a alteração no retorno da função (linha 12), onde, após as alterações, faz-se necessário verificar qual é o último vetor gerado e atribuí-lo a LL.

4.2. Remoção da Dependência na Geração do Vetor de Similaridades

Em [Yang et al. 2010], para o algoritmo de Smith-Waterman [Smith and Waterman 1981], os autores mostram que a dependência na mesma linha para gerar K_1 pode ser removida se for utilizada uma matriz auxiliar P baseada no dicionário de itens utilizados. Com isso, a geração de K_1 depende apenas de K_0.

A matriz P para a sequência B é definida como:

$$P[i,j] = \begin{cases} 0, & \text{se i = 0 ou j = 0} \\ j, & \text{se B[j-1] = C[i]} \\ P[i,j-1], & \text{caso contrário} \end{cases}$$

Onde C é o dicionário de itens, ou seja, um vetor com todos os possíveis símbolos. Por exemplo, para análises de sequências homólogas de DNA, o dicionário C seria $\{A, C, G, T\}$.

Assim, $K_1[j]$ pode ser determinado da seguinte forma:

$$K_1[j] = \begin{cases} 0, & \text{se j = 0} \\ K_0[j], & \text{se P[c,j] = 0} \\ max(K_0[j], K_0[P[c,j]-1]+1), & \text{caso contrário} \end{cases}$$

Onde c é o índice de $A[i]$ em C. Dessa forma, cada iteração do loop (linhas 5 a 11) do Algoritmo 2 pode ser executada independentemente, propiciando uma grande oportunidade de ganho de desempenho.

4.3. Análise Experimental

A Tabela 1 mostra o tempo médio de execução em segundos e o *speedup* obtido, em relação a versão original, ao se executar as versões sequenciais do algoritmo de Hirschberg, compiladas com Intel ICC versão 16.0.1 20151021 e opção O3, em uma máquina multicore com um total de 36 núcleos (dois processadores Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz) com 128 GB de memória compartilhada. Esse ambiente foi utilizado para todos os experimentos deste trabalho. Foram utilizadas sequências de tamanhos que variam entre 50K e 1600K caracteres. Para cada sequência, cada versão foi executada 3 vezes para se minimizar a influência do ambiente computacional de uso não exclusivo. Para se ter a precisão da média, a tabela apresenta também o coeficiente de variação (Co-Var) para cada média calculada.

Os tempos obtidos ao se executar o algoritmo descrito na Seção 3 são exibidos na primeira linha, linha **Original**, da Tabela 1. Com a melhoria descrita na subseção 4.1 (linha **Sem Cópia** da Tabela 1) observa-se um *speedup* de 1,2 em relação a versão original. Ao se agregar a essa versão a melhoria descrita na subseção 4.2 (linha **Sem Dependência** da Tabela 1), já é possível observar um *speedup* de até 1,4. Por fim, o código fonte foi recompilado para utilizar as instruções de extensão vetorial disponíveis para os processadores utilizados (*Advanced Vector Extensions 2* - AVX2) [Lento 2014] e, combinadas com a versão anterior, obteve-se um *speedup* de até 1,9 em comparação com a versão original (linha **Sem Dependência com SIMD** da Tabela 1). Tal ganho só foi possível em virtude da remoção da dependência descrita na subseção 4.2. É importante ressaltar o baixo valor do coeficiente de variação, o que mostra que os tempos de execução foram muito próximos uns dos outros.

5. Paralelização do Algoritmo de Hirschberg

Enquanto que os algoritmos baseados em programação dinâmica para o cálculo da maior subsequência comum utilizam, de maneira geral, a técnica de *wavefront* [Mohanty and Cole 2014], por utilizarem uma matriz de similaridade, o algoritmo de Hirschberg utiliza apenas um vetor para diminuir a complexidade de espaço, não sendo possível a aplicação dessa técnica. Assim, analisando as dependências do algoritmo de Hirschberg, três abor-

Tabela 1. Tempos de execução em segundos para as versões sequenciais

		50K	100K	200K	400K	800K	1600K
Original	Tempo	13,4	55,8	227,6	922,2	3680,3	14844,5
	CoVar	0,9%	0,2%	0,0%	0,0%	1,2%	0,0%
Sem Cópia	Tempo	11,4	46,1	185,5	742,4	2966,1	11857,3
	Speedup	1,2	1,2	1,2	1,2	1,2	1,3
	CoVar	1,4%	0,4%	0,1%	0,0%	0,0%	0,0%
Sem Dependência	Tempo	10,0	39,6	158,4	631,7	2554,4	10427,1
	Speedup	1,3	1,4	1,4	1,4	1,4	1,4
	CoVar	1,5%	0,3%	0,0%	0,1%	0,6%	0,0%
Sem Dependência com SIMD	Tempo	7,5	29,8	118,7	469,8	1877,4	7926,1
	Speedup	1,8	1,9	1,9	1,9	1,9	1,9
	CoVar	1,8%	0,5%	0,3%	0,5%	0,3%	0,1%

dagens paralelas podem ser adotadas. As subseções a seguir descrevem, analisam e avaliam a implementação de cada uma dessas abordagens.

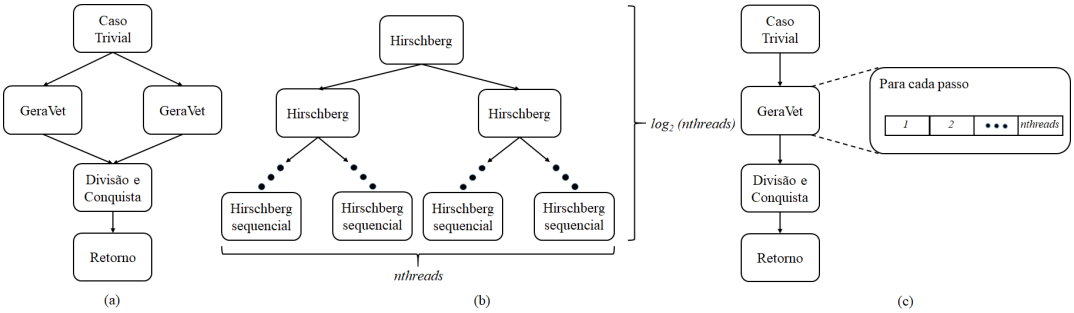


Figura 1. Abordagens para a Paralelização

5.1. Paralelização da Geração dos Vetores de Similaridade

Entre as linhas 15 e 17 do Algoritmo 1, os vetores de similaridade são gerados de forma independente, suscitando a primeira abordagem de paralelização. Como pode ser visto na Figura 1(a), a cada iteração do algoritmo a função *Geravet*, descrita no Algoritmo 2, é chamada duas vezes, podendo portanto ser realizada, em paralelo, a geração dos dois vetores. Para tal, são criadas duas tarefas, uma para cada uma das chamadas. Somente após término dessas duas tarefas a execução da divisão e conquista pode ocorrer, sendo o único ponto de sincronismo desta abordagem, ficando assim restrita a dois processadores. Esta técnica já foi implementada e avaliada em [Rashid et al. 2007].

5.2. Paralelização da Divisão e Conquista

Entre as linhas 20 e 21 do Algoritmo 1 é utilizada divisão e conquista. Cada ramo da recursão é independente, o que sugere a segunda abordagem de paralelização. Conforme pode ser visto na Figura 1(b), a cada divisão o número de tarefas dobra e podem ser processadas em paralelo. Assim, a paralelização adotada foi criar uma *thread* para cada uma dessas tarefas, que são sincronizadas apenas no final da fase de conquista. Neste

caso, seria necessário uma quantidade exponencial de processadores, uma vez que cada ramo se dividirá em dois até que seja atingido um dos casos triviais.

Para limitar a quantidade de *threads* e, com isso, não diminuir significativamente a granularidade das tarefas, a abordagem adotada foi a de utilizar um parâmetro adicional para indicar o nível da chamada recursiva, sendo este iniciado com 0 e incrementado a cada nível da recursividade. Caso $2^{nível}$ seja maior que o número de processadores disponíveis, a recursão prosegue de forma sequencial. Por exemplo, para 16 processadores, a partir do 5^o nível, a computação passa a ser sequencial.

5.3. Paralelização Fina da Geração dos Vetores de Similaridade

Com a independência obtida pela melhoria descrita na subseção 4.2, é possível executar cada passo da geração dos vetores de similaridade de forma paralela, ou seja, cada elemento do vetor pode ser calculado de maneira independente. Nesta abordagem, a cada passo, os elementos do vetor são calculados em paralelo pelos processadores disponíveis, como mostra a Figura 1(c), havendo um sincronismo a cada passo da geração do vetor. Em cada um desses passos é gerado um novo vetor até que todas as linhas, do que seria a matriz de similaridades, sejam percorridas. Por exemplo, para comparar duas sequências com 10K caracteres em 16 processadores, são necessários 10K passos, onde, a cada passo, cada processador calcula em paralelo 625 $(10.000/16)$ caracteres.

A medida que o algoritmo de Hirschberg promove a divisão das sequências a serem comparadas, o tamanho dos vetores de similaridade diminui. Por exemplo, para uma sequência *B* inicial de 50K caracteres processada por 4 *threads*, no primeiro nível da recursão, cada *thread* seria responsável por calcular 12.500 elementos. Para o segundo nível, assumindo uma divisão equânime, cada *thread* seria responsável por 6.250 elementos. O algoritmo segue assim até atingir o caso trivial. Dessa forma, a partir de determinado tamanho da sequência *B*, pode não haver mais ganho em se executar de forma paralela cada elemento do vetor, uma vez que há custos envolvidos na criação e sincronização das *threads* necessárias para tal. Assim, nessa abordagem, foi criado um limite mínimo de corte para o tamanho da sequência *B* a partir do qual a execução é realizada sequencialmente. O valor desse limite é avaliado na subseção 5.4.

5.4. Análise Experimental

O presente trabalho se propõe a utilizar técnicas de paralelização baseadas em memória compartilhada, com o intuito de aproveitar da melhor maneira a existência de diversos núcleos (CPUs) nos processadores. Assim, foi utilizado o modelo de programação paralela com memória compartilhada *OpenMP* [Chandra et al. 2000]. Para a implementação das técnicas descritas nas subseções 5.1 e 5.2 foram utilizadas as primitivas *OpenMP* para manipulação de *Tasks*. Já a abordagem da subseção 5.3 foi implementada utilizando a primitiva *Parallel for*.

Na subseção 5.3 foi descrita a utilização de um limite mínimo a partir do qual a geração dos elementos do vetor de similaridade passa a ser sequencial. A Tabela 2 mostra os tempos de execução dessa abordagem para sequências de tamanhos que variam entre 50K e 1600K caracteres, executando com 2, 4, 8, 16 e 32 *threads* e limites mínimos 1024, 2048 e 4096. Os melhores resultados foram obtidos, em 75% das vezes, para o limite mínimo de 2048 caracteres. Dessa forma, estes tempos serão utilizados para a comparação entre as melhorias.

Tabela 2. Tempos de execução em segundos para a abordagem da subseção 5.3

		50K			100K			200K			400K			800K			1600K		
Thr	Limite	1K	2K	4K	1K	2K	4K	1K	2K	4K	1K	2K	4K	1K	2K	4K	1K	2K	4K
2	Tempo	4,5	4,5	4,9	15,6	15,4	15,7	60,2	60,2	61,1	242,5	242,8	243,8	921,2	915,4	914,7	3863,2	3917,3	3944,5
	CoVar	3,4%	4,0%	6,7%	1,4%	1,2%	1,6%	0,3%	0,3%	1,4%	0,5%	0,8%	1,3%	0,3%	0,3%	0,1%	1,4%	0,4%	1,0%
4	Tempo	3.3	3,3	3,1	10,1	9,8	10,1	35,9	35,0	35,9	139,2	130,3	133,2	501,1	502,5	501,6	2078,7	2081,2	2083,3
	CoVar	4,5%	8,2%	3,2%	2,8%	0,5%	1,9%	0,8%	1,9%	4,3%	4,5%	2,2%	2,8%	1,1%	0,7%	1,5%	2,0%	0,7%	0,4%
8	Tempo	2,7	2,8	2,9	7,4	7,4	7,5	23,3	23,0	23,9	78	77,4	78,5	288,7	293	314,2	1123,9	1141,2	1138,5
	CoVar	2,7%	2,9%	4,5%	1,3%	4,7%	3,8%	2,3%	1,7%	1,6%	1,3%	0,3%	0,9%	0,2%	1,4%	0,6%	1,3%	1,0%	0,3%
16	Tempo	2,7	2,8	2,7	6,8	6,6	6,7	18,7	18,4	18,5	56,1	55,3	55,5	180,6	179,4	186,5	665,6	660,3	668,4
	CoVar	1,0%	2,4%	1,3%	3,5%	0,2%	3,0%	3,0%	0,4%	1,3%	1,1%	1,4%	1,2%	1,2%	1,5%	2,4%	0,7%	0,8%	0,4%
32	Tempo	3,3	3,2	3,3	7,4	7,2	7,5	17,9	17,6	17,6	43,6	42,6	43	129,1	129,9	128,8	444,2	440,3	440,9
	CoVar	2,3%	0,7%	1,4%	5,2%	3,2%	1,3%	2,5%	0,6%	0,4%	2,8%	0,9%	3,9%	1,4%	3,4%	0,9%	1,3%	1,4%	0,7%

A Tabela 3 mostra os tempos de execução das três abordagens: paralelização da geração dos vetores de similaridade (subseção 5.1 - *VetSim*), paralelização da divisão e conquista (subseção 5.2 - *DivCon*) e paralelização fina da geração dos vetores de similaridade (subseção 5.3 - *ParFin*). São utilizadas sequências de tamanhos que variam entre 50K e 1600K caracteres. As abordagens foram executadas com 2, 4, 8, 16 e 32 *threads*, com exceção da abordagem *VetSim* que está naturalmente limitada a 2 *threads*. Nesta abordagem, o *speedup* (em relação à melhor versão sequencial Seção 4.3) ficou entre 1,7 e 1,9, mostrando-se quase linear.

A abordagem *DivCon* apresenta um *speedup* não maior que 1,3, independente do número de *threads* utilizadas. Para se entender tal desempenho, faz-se necessário analisar as etapas responsáveis pelo tempo de execução do Algoritmo 1. Cerca de 70% do tempo total de execução está na geração dos vetores de similaridade (linhas 15 a 17) da primeira chamada recursiva da função. Com isso, só é possível melhorar o desempenho paralelizando os 30% restantes. Assumindo o maior número de *threads* utilizadas neste trabalho (32), o tempo de execução da parte restante seria, no melhor caso, dividido por 32. Utilizando a Lei de Amdahl [Amdahl 1967], o *speedup* máximo teórico que se poderia obter com a paralelização é dado por:

$$Speedup_{Max} = \frac{1}{(1-p) + \frac{p}{t}} \approx 1,4, \text{ para } p = 0,3 \text{ e } t = 32$$

Assim, fica claro que o desempenho obtido está muito próximo do limite teórico.

Tabela 3. Tempos em segundos e Speedups para as três abordagens propostas

		50K			100K			200K			400K			800K			1600K		
Thr		VetSim	DivCon	ParFin	VetSim	DivCon	ParFin	VetSim	DivCon	ParFin	VetSim	DivCon	ParFin	VetSim	DivCon	ParFin	VetSim	DivCon	ParFin
2	Tempo	4,4	7,0	4,5	15,3	26,3	15,4	61,4	102,3	60,2	247,0	430,2	242,8	1013,3	1631,5	915,4	4522,2	7103,5	3917,3
	CoVar	3,5%	0,8%	4,0%	1,2%	1,8%	1,2%	1,6%	0,2%	0,3%	0,0%	0,3%	0,8%	0,2%	0,1%	0,3%	0,4%	0,2%	0,4%
	SpeedUp	1,7	1,1	1,7	1,9	1,1	1,9	1,9	1,2	2,0	1,9	1,1	1,9	1,9	1,2	2,1	1,8	1,1	2,0
4	Tempo		6,6	3,3		24,1	9,8		93,9	35,0		396,7	130,3		1497,6	502,5		6735,4	2081,2
	CoVar		2,7%	8,2%		0,4%	0,5%		0,2%	1,9%		0,2%	2,2%		0,1%	0,7%		2,5%	0,7%
	SpeedUp		1,1	2,3		1,2	3,0		1,2	3,4		1,2	3,6		1,3	3,7		1,2	3,8
8	Tempo		6,3	2,8		23,7	7,4		93,7	23		390,5	77,4		1469,6	293		6425,3	1141,2
	CoVar		1,0%	2,9%		0,2%	4,7%		0,9%	1,7%		0,3%	0,3%		0,0%	1,4%		0,6%	1,0%
	SpeedUp		1,2	2,7		1,3	4		1,3	5,2		1,2	6,1		1,3	6,4		1,2	6,9
16	Tempo		6,5	2,7		24,0	6,6		92,0	18,4		388,0	55,3		1460,5	179,4		6405,5	660,3
	CoVar		0,9%	2,4%		1,6%	0,2%		0,1%	0,4%		0,1%	1,4%		0,2%	1,5%		0,8%	0,8%
	SpeedUp		1,1	2,8		1,2	4,5		1,3	6,5		1,2	8,5		1,3	10,5		1,2	12
32	Tempo		6,5	3,2		23,8	7,2		92,1	17,6		386,5	42,6		1459,5	129,9		6384,9	440,3
	CoVar		0,6%	0,7%		0,2%	3,2%		0,3%	0,6%		0,2%	0,9%		0,2%	3,4%		0,5%	1,4%
	SpeedUp		1,2	2,3		1,2	4,2		1,3	6,8		1,2	11,0		1,3	14,5		1,2	18,0

Diferente das duas anteriores, a abordagem *ParFin* obteve *speedups* expressivos, à medida que o tamanho das sequências aumenta, apesar de sua granularidade fina e que varia consideravelmente. Para os experimentos realizados, a granularidade varia entre o mínimo de 64 caracteres, para 32 *threads* e o limite mínimo de 2048, e $800K$, para 2 *threads* e o tamanho de sequência 1600K. O desempenho da abordagem *ParFin* pode

ser observado no gráfico da Figura 2, onde fica clara a influência da granularidade fina principalmente no desempenho para 32 *threads*, onde o *speedup* tem um crescimento mais acelerado a partir de sequências com tamanho 200K.

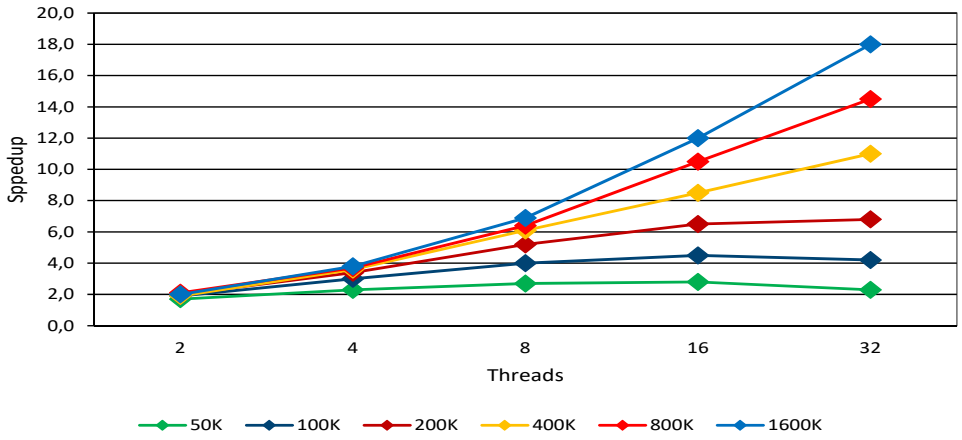


Figura 2. Speedup da Abordagem ParFin em relação à quantidade de threads

Para ilustrar o ganho no desempenho total obtido por essa abordagem, acrescido da melhoria na versão sequencial, o gráfico da Figura 3 mostra o *speedup* quando calculado em relação à versão sequencial original do algoritmo. Pode-se observar que este *speedup* chega a 33 para uma sequência de tamanho 1600K executando com 32 *threads*.

É importante destacar que a perda de desempenho, especialmente para as menores sequências, é fruto da sobrecarga da execução paralela, em função da granularidade fina das tarefas, como já explicado anteriormente. Além disso, como a máquina utilizada é do tipo NUMA (*Non-Uniform Memory Access*), ou seja, o acesso a memória não é uniforme, foi observado que parte da perda de desempenho surge em função desse acesso não uniforme. Experimentos preliminares restringindo o acesso a uma parte da memória NUMA com acesso uniforme, apresentou ganhos significativos. A melhora do desempenho nessas arquiteturas será estudada em trabalhos futuros.

Cabe ressaltar também que foram realizadas avaliações iniciais juntando as técnicas de paralelismo avaliadas neste trabalho. O desempenho da versão paralela juntando a técnica *DivCon* com a *ParFin* foi muito ruim, principalmente, em função do desbalanceamento dos ramos de cada divisão da técnica de divisão e conquista. Como cada ramo produz tarefas com tamanhos distintos, a execução de cada ramo é realizada em um tempo distinto, ao invés de forma balanceada, o que prejudica consideravelmente o desempenho.

Com maior potencial para aumentar o desempenho, a versão paralela juntando a técnica *VetSim* com a *ParFin*, apesar de um desempenho bem melhor que a anterior, também não foi melhor que somente a versão *ParFin*. Neste caso, apesar dos dois vetores de similaridades serem balanceados, a perda de desempenho em função do acesso à memória NUMA prejudicou o desempenho. Em trabalhos futuros serão feitas

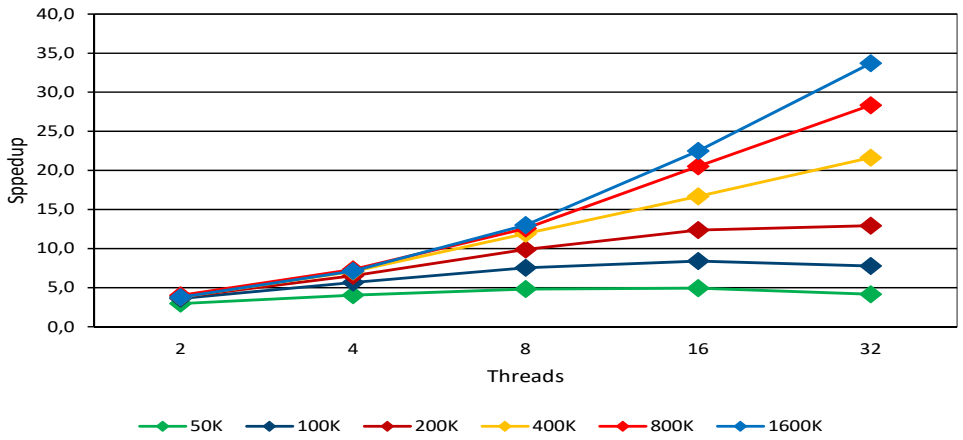


Figura 3. Speedup da Abordagem ParFin em relação à versão sequencial original

investigações para avaliar se o melhor uso da memória NUMA pode fazer com que essa combinação de abordagens execute eficientemente.

6. Conclusão

Achar a maior subsequência comum (*Longest Common Subsequence - LCS*) entre duas sequências em um tempo aceitável é fundamental para várias aplicações. O algoritmo de Hirschberg permite encontrar a solução ótima do problema, com um gasto linear de memória, o que permite a comparação de sequências longas. Este trabalho implementou e avaliou melhorias de desempenho para versão sequencial do algoritmo de Hirschberg, assim como, técnicas de paralelismo para uso eficiente de máquinas *multicore*. Os resultados mostraram que a versão sequencial ficou aproximadamente duas vezes mais rápida do que a versão original, o que proporcionou um *speedup* de 18 quando executando a versão paralela para sequências longas em 32 *cores*. Se for considerado o *speedup* em relação à versão sequencial original, o *speedup* da nova proposta atinge 33.

Como trabalhos futuros, pretende-se investigar a sobrecarga de acesso à memória em arquiteturas NUMA, assim como, a vetorização manual da abordagem *ParFin* e, com isso, avaliar e implementar técnicas para melhorar ainda mais o desempenho do algoritmo de Hirschberg nesses ambientes.

Agradecimentos

Os autores agradecem o uso dos recursos computacionais *manycore* mantidos e operados pelo Núcleo de Computação Científica da Universidade Estadual Paulista (NCC/UNESP), financiado parcialmente pela Intel, no contexto do projeto Intel/UNESP Modern Code.

Referências

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), páginas 483–485, New York, NY, USA. ACM.

Anvik, J., MacDonald, S., Szafron, D., Schaeffer, J., Bromling, S., and Tan, K. (2002). Generating parallel programs from the wavefront design pattern. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, páginas 165–172.

Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., and McDonald, J. (2000). *Parallel Programming in OpenMP*. Morgan Kaufmann, 1st edition.

Diaz, J., Muñoz-Caro, C., and Niño, A. (2012). A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386.

Driga, A., Lu, P., Schaeffer, J., Szafron, D., Charter, K., and Parsons, I. (2003). FastLSA: a fast, linear-space, parallel and sequential algorithm for sequence alignment. In *2003 International Conference on Parallel Processing, 2003. Proceedings.*, páginas 48–57.

Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

Hirschberg, D. S. (1975). A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343.

Lento, G. (2014). Optimizing performance with intel® advanced vector extensions. Technical report, Intel.

Martins, W. S., del Cuvillo, J., Useche, F. J., Theobald, K. B., and Gao, G. R. (2001). A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. In *Pacific Symposium on Biocomputing*.

Mohanty, S. and Cole, M. (2014). Autotuning wavefront applications for multicore multi-gpu hybrid architectures. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, PMAM'14, páginas 1:1–1:9, New York, NY, USA. ACM.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453.

Rashid, N. A., Abdullah, R., and Talib, A. Z. H. (2007). Parallel homologous search with hirschberg algorithm: A hybrid MPI-pthreads solution. In *Proceedings of the 11th WSEAS International Conference on Computers*, páginas 228–233.

Sandes, E. F. D. O., Boukerche, A., and Melo, A. C. M. A. D. (2016). Parallel optimal pairwise biological sequence comparison: Algorithms, platforms, and classification. *ACM Comput. Surv.*, 48(4):63:1–63:36.

Smith, T. and Waterman, M. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197.

Yang, J., Xu, Y., and Shang, Y. (2010). An efficient parallel algorithm for longest common subsequence problem on GPUs. In *Proceedings of the World Congress on Engineering*, páginas 499–504.

Paralelização Híbrida e em Múltiplos Níveis de um Algoritmo de Contabilização de Frequências de *K-mer*

Fabrício Vilasbôas[1], **Micaella Coelho**[1],
Carla Osthoff[1], **Kary Ocaña**[1], **Ana Tereza Vasconcelos**[1]

[1]Laboratório Nacional de Computação Científica

{fabricio, micaella, osthoff, karyann, atrv}@lncc.br

Resumo. *Este trabalho apresenta um estudo sobre o ganho de desempenho gerado com o aumento do nível de paralelismo de execução de um algoritmo de contabilização de k-mers de dados de amostras genômicas e metagenômicas. O algoritmo foi originalmente desenvolvido para permitir a execução em paralelo da contabilização de k-mers em uma arquitetura manycores de uma GPU. Neste trabalho apresentamos o ganho de desempenho ao implementarmos paralelismo para a execução paralela de módulos do algoritmo em múltiplos núcleos de uma CPU, em múltiplas GPU's e em múltiplos nós de processamento de uma cluster híbrida.*

1. Introdução

Com o surgimento da tecnologia *NGS (Next Generation Sequencing)* muitos foram os avanços na área da pesquisa genômica. As técnicas de sequenciamentos mais avançados geravam 84 MB de dados por rodada e hoje, com o NGS, são gerados terabytes de dados por rodada. Os avanços na tecnologia de sequenciamento aumentaram a necessidade do desenvolvimento de técnicas para permitir a análise dos dados de forma mais rápida e precisa. Neste trabalho apresentamos otimizações de desempenho de um algoritmo de contabilização de *k-mers* para dados de amostras genômicas e metagenômicas, chamado de "Contabilizador da Frequência de Repetição de *k-mers*"ou **CFRK**.

O algoritmo **CFRK** [Vilasboas et al. 2015] foi desenvolvido para efetuar contabilização da frequência de repetição de *k-mers* de bases de dados de amostras genômicas em ambientes de *GPU's*. O termo *k-mer* é utilizado como referência a todas as possíveis combinações de comprimento k que estão contidas em uma sequência de dados arbitrária. Essa técnica está contida na interseção entre a mineração de dados e a análise combinatória. As aplicações dos *k-mers* são inúmeras. *K-mers* podem ser utilizados para montagem [Garg et al. 2013] de amostras de DNA, RNA ou proteínas, para alinhamento de sequências [Church et al. 2011], ou para análise qualitativa de um sequenciamento [Plaza Onate et al. 2015]. Também podem ser utilizados como entrada para algum método de classificação ou agrupamento [Fiannaca et al. 2015]. Os valores de k utilizados variam de acordo com a análise feita.

A primeira versão do **CFRK** apresentou um ganho de aproximadamente 6 vezes em relação ao **Jellyfish**, considerado o estado da arte para contabilização de *k-mer*, para valores de k menores que 5, porém tinha a limitação de processar apenas arquivos de tamanho menor ou igual à memória principal da *GPU*. A segunda versão do **CFRK**

[Vilasboas et al. 2016] implementou uma rotina para dividir os dados de entrada e possibilitou a execução de arquivos de tamanho maior do que a memória principal da *GPU*. Neste trabalho apresentamos a terceira versão do **CFRK**, que introduziu mais três níveis de paralelismo: um primeiro nível para permitir o processamento de parte do algorítimo em múltiplos núcleos do *módulo CPU*, um segundo nível para permitir o processamento de parte do algoritmo em múltiplas *GPU's* e um terceiro nível para permitir a execução de múltiplas instâncias do algoritmo **CFRK** em nós de processamento de uma *cluster*.

Este trabalho analisa o ganho de desempenho do **CFRK** para cada otimização implementada e o ganho de desempenho total do **CFRK** com todas as otimizações. Demonstramos que a adição de mais 3 níveis de paralelismo aumenta significativamente o desempenho do algoritmo.

Este trabalho está dividido da seguinte forma: Seção 2 apresenta os trabalhos relacionados; Seção 3 apresenta a descrição do algoritmo **CFRK** e das otimizações desenvolvidas. Seção 4 apresenta a análise de desempenho. Seção 5 apresenta a descrição do **MCFRK**, resultados e análises. Seção 6 apresenta as conclusões e os trabalhos futuros.

2. Trabalhos Relacionados

O *Jellyfish* [Marçais and Kingsford 2011] é um algoritmo desenvolvido para processamento em memória compartilhada e é considerado o estado da arte entre os algoritmos de contabilização da frequência de repetição de *k-mers*. Ele utiliza várias estruturas *lock-free*, que são estruturas que permitem operações atômicas sem o bloqueio da memória e, por isso, não degradam o desempenho em ambientes multiprocessáveis. O *Jellyfish* apresenta bom desempenho para a contabilização de valores de $10 \leq k \leq 32$ enquanto que o **CFRK** possui melhor desempenho para valores de $k \leq 9$. Por outro lado, o *Jellyfish* foi projetado para processar amostras de genoma, pois ele considera todos os *reads*[1] como sendo pertencente a um mesmo organismo. O **CFRK** foi projetado para permitir a análise de amostras de genomas e de metagenomas, pois considera que cada *read* pode pertencer a um organismo distinto. Isto possibilita uma análise conclusiva para grande classe de pesquisas em bioinformática, conforme o trabalho [Edwards et al. 2012].

O **Gerbil** [Erbert et al. 2017] é um algoritmo para contabilização da frequência de repetição de *k-mers* com suporte a múltiplas *GPU's* e que faz uso do disco para melhorar a eficiência em relação à memória. Sua execução consiste de duas fases: distribuição e contabilização. Este algoritmo é projetado para o processamento de genomas. Isto permite que o *Gerbil* efetue menos transferência de dados entre a *GPU* e o *host* e apresente um bom desempenho para valores de $k \leq 32$. Porém, assim como o *Jellyfish*, o **Gerbil** não pode ser utilizado para pesquisas que necessitam gerar um vetor de frequência para cada *read*.

3. Descrição do algoritmo

CFRK é acrônimo para "Contabilizador da Frequência de Repetição de *k-mers*", foi desenvolvido para *GPU's* da *NVidia*, utilizando a linguagem *CUDA* e os *drivers* disponibilizados pelo fabricante.

Para o desenvolvimento do **CFRK** utilizamos o conceito de programação modular, onde cada módulo é responsável por uma atividade específica na execução do algoritmo.

[1]Fragmentos de material genético resultantes do sequenciamento

O **CFRK** é composto por dois módulos principais: o módulo que contém as funções que são executadas pela *CPU*, chamado *módulo CPU*, e o módulo que contém as funções que são executadas pela *GPU*, chamado *módulo GPU*. O *módulo CPU* é composto por dois submódulos: o *main* e o *kmer_main*. O submódulo *main* é responsável pela leitura do arquivo de entrada, pela preparação dos *chunks* para serem enviados as *GPU's* e pela gerência da execução dos outros submódulos. O módulo *kmer_main* é responsável pela preparação da *GPU* para a execução e possui as rotinas de gerenciamento e transferência de memória e chamadas aos *kernels*. O *módulo GPU* é composto pelo submódulo *kmer_kernel*, que contém os *kernels* executadas na *GPU*.

3.1. Descrição das funções do *módulo CPU*

Nesta seção descreveremos o funcionamento dos dois submódulos contidos no *módulo CPU*, o submódulo *main* e o submódulo *kmer_main*.

3.1.1. Submódulo *main*

O arquivo resultante do sequenciamento de uma amostra genômica ou metagenômica contém um grande número de fragmentos, denominadas *reads*. Cada *read* possui uma sequência de nucleotídeos que são representados pelos caracteres A, C, G, T ou N. O caractere N é observado no *read* quando o sequenciador não reconhece com precisão o nucleotídeo pertencente àquela posição. Quando o arquivo é lido, é feita uma codificação para transformar os caracteres A, C, G, T ou N para os valores numéricos 0, 1, 2, 3 ou -1, respectivamente. Os *reads* são dispostos em sequência formando uma estrutura de *array* unidimensional contendo todos os nucleotídeos de todos os *reads*, chamado *vetor de reads*. Para identificar o final de cada *read*, é utilizado o valor -1. O valor -1 é utilizado para representar tipos de dados inválidos, ou seja, o *k-mer* que contém o valor -1 não deverá ser processado. Os arquivos gerados pelos sequenciadores de tecnologia *NGS* podem conter uma grande quantidade de informação e a memória global da *GPU* pode não ser suficiente para armazenar toda essa informação. Por isto o *vetor de reads* é subdividido e armazenado em estruturas chamadas *chunks*. Cada *chunk* possui uma quantidade pré-definida de *reads*, bem como as informações da posição inicial e o tamanho de cada *read*. Posteriormente, os *chunks* são enfileirados e serão enviados um a um para o processamento na *GPU*. Com isso pode-se ajustar a quantidade de dados que será enviada para o processamento na *GPU* e, assim, tem-se o controle da quantidade total de memória utilizada para o processamento. O usuário tem liberdade para definir a quantidade de *reads* por *chunk*, sendo que a quantidade máxima varia de acordo com a disponibilidade da memória da placa que está em utilização. Esta função demanda um grande tempo de processamento devido a manipulação de memória. Esta função pertence ao módulo *main*, no qual estão todas as funções que são executadas exclusivamente pela *CPU*.

3.1.2. Submódulo *kmer_main*

Os *chunks* são enviados para o submódulo *kmer_main* através de uma estrutura que contém as sequências a serem processadas o número de sequências, o tamanho de cada sequência e a posição inicial de cada sequência. É neste módulo de que se encontram todas as funções de gerenciamento de memória, funções para alocação e desalocação

dos vetores, funções de transferência de memória entre *host* e *device* e as chamadas aos *kernels*. Este módulo foi alvo do estudo publicado em [Vilasboas et al. 2016].

3.1.3. Otimização no *módulo CPU*

Foi feito um estudo do comportamento das funções executadas pela *CPU* durante a execução do **CFRK**. Foi utilizada a ferramenta *VTune* disponibilizada pela *Intel* em seu pacote *Intel Parallel Studio*. Observamos nestes resultados que a função de leitura de arquivo e a função responsável pela seleção dos *chunks* compunham os *hotspots*. O VTune nos permitiu visualizar as instruções que mais demandaram processamento dentro destes *hotspots* e com base nestas informações foi selecionado um laço de repetição para a paralelização via diretivas do *OpenMP*.

O laço de repetição selecionado foi o que mais demandou processamento na função de seleção dos *chunks*. A operação realizada por este laço de repetição consiste em uma cópia de memória entre dados lidos do arquivo de entrada e a estrutura que armazenará os *chunks* para o processamento. A diretiva inserida foi a *#pragma omp parallel for*. Essa diretiva cria uma região paralela na qual as iterações do laço de repetição são divididas para serem executadas em paralelo pelos núcleos da *CPU*.

3.2. Descrição das funções do *módulo GPU*

O processo de contabilização da frequência de repetição de *k-mers* é dividido em duas fases. A primeira fase é a conversão dos valores da base 4 para a base 10 e a segunda fase é a contabilização da frequência de repetição de *k-mers*.

A primeira fase do processamento consiste em converter os valores numéricos dos *k-mers*, que estão na base 4, para a base 10. Essa conversão é feita para facilitar o mapeamento entre o valor da combinação do *k-mer* e a posição do contador no *vetor de frequência* referente àquela combinação, dado que o valor do *k-mer* na base 10 será exatamente o valor da posição a qual deverá ser feita a contabilização da repetição. O resultado desta conversão é armazenado em um vetor denominado *vetor de conversão*.

Na segunda fase é calculada da frequência de repetição de cada *k-mer*. É alocado um vetor chamado *vetor de frequência*. Este vetor irá armazenar o resultado do cálculo da frequência de repetição. Para cada *read* no *vetor de reads* será necessário alocar um *vetor de frequência* e cada posição do *vetor de frequência* será um contador correspondente ao valor do *k-mer* no *vetor de conversão*. Então o *vetor de conversão* é percorrido e é realizada a operação $+1$ no *vetor de frequência* na posição correspondente ao valor do *k-mer*.

3.2.1. Otimização no *módulo GPU*

A *GPU* foi utilizada para acelerar o processamento da conversão dos valores dos *k-mers* da base 4 para a base 10 e também a contabilização da frequência de repetição dos *k-mers*. Essas funções foram selecionadas para este tipo de abordagem pelo fato de se enquadrarem no paradigma *Single Instruction Multiple Data (SIMD)*, ou seja, vários dados sendo operados por uma instrução. Estas otimizações estão descritas no trabalho [Vilasboas et al. 2016].

Para este trabalho foi feita a alteração do tipo do dado utilizado no *vetor de reads*. Até a segunda versão do **CFRK** o *vetor de reads* era armazenado como *int* (inteiro), que ocupa 4 *bytes* na memória. A proposta foi alterar de *int* para o tipo *char* (caractere), que ocupa 1 *byte* na memória. Essa alteração foi possível porque a linguagem *CUDA* é uma extensão da linguagem *C* e esta usa o tipo do dado apenas como referência ao espaço de alocação em memória permitindo fazer cálculos com variáveis de qualquer tipo primitivo.

Essa alteração foi feita no *kernel ComputeIndex* do submódulo *kmer_kernel* que pertence ao *módulo GPU*.

3.3. Suporte a execução em múltiplas *GPU's*

De forma a possibilitar que a execução do **CFRK** em múltiplas *GPU's*, foi desenvolvida uma função no submódulo *main* que reconhece o número de *GPU's* disponíveis através da função *cudaGetDeviceCount* da biblioteca padrão do *CUDA* e utiliza a biblioteca *pthreds* para criar *threads* do módulo *kmer_main* para serem executadas em paralelo em cada *GPU*. A função desenvolvida, após gerar as *threads*, irá designar um grupo de *chunks* para serem processados em cada *thread* e a seguir enviar as *threads* para serem executadas em cada *GPU*. Após a execução os vetores de frequência gerados por cada *GPU* são coletados pelo módulo *main* e armazenados no arquivo de saída e as *threads* são destruídas.

4. Resultados e análises

Nesta seção apresentaremos os resultados referentes as otimizações implementadas nos dois módulos principais e seus respectivos resultados.

Para os experimentos foram utilizando os nós computacionais *GPU's* do supercomputador SDumont, onde cada nó possui a configuração: 2 x *CPU Intel Xeon E5-2695v2 Ivy Bridge, 2,4GHZ*, 24 núcleos (12 por *CPU*), totalizando de 1.296 núcleos, *64GB DDR3 RAM*, 2 x *Nvidia K40* (dispositivo *GPU*), sistema operacional *Red Hat Enterprise Linux Server release 6.4*, *kernel* versão *2.6.32-504.12.2.el6.x86_64*, *CUDA 8.0*, *Intel Parallel Studio XE 2016*, compilador *icpc* com flags as *flags -qopenmp e -Ofast*, *VTune* e *OpenMP 4.5*. Este supercomputador está alocado no Laboratório Nacional de Computação Científica (LNCC) e para este trabalho foram utilizados os nós *B715* com placas aceleradoras *GPU Nvidia K40*.

O arquivo de entrada utilizado foi uma amostra real de metagenoma com 8.7GB de dados cuja identificação no banco de dados *SRA* do *NCBI* (*National Center for Biotechnology Information*) é *SRX2021688*. Este arquivo foi escolhido por representar um típico arquivo de tamanho médio utilizado para pesquisas de metagenoma.

4.1. Resultado das otimizações no *módulo CPU*

Utilizamos o *VTune* para gerar o perfil de execução do **CFRK** paralelizado com a diretiva *OpenMP*. Dado que o valor de k não é utilizado pelas rotinas dos submódulos da *CPU*, os resultados apresentados são relativos a um valor fixo de k. Escolhemos o valor $k = 4$ que é utilizado para análise de genoma conforme o trabalho [Edwards et al. 2012]. Denominamos a versão do **CFRK** que possui os submódulos do *módulo CPU* paralelizados através do *OpenMP* de **CFRK_OMP**. A Tabela 1 apresenta na primeira coluna o nome das funções das duas versões do algoritmo **CFRK** sendo a primeira linha referente

a versão sem a diretiva *OpenMP*, **CFRK**, e a segunda linha referente a versão com a diretiva *OpenMP*, **CFRK_OMP**. A segunda coluna apresenta o tempo de execução em segundos de cada uma das funções. A terceira coluna apresenta o ganho em vezes da função com a diretiva *OpenMP* em relação a mesma função sem a diretiva. Nossos expe-

Função	Tempo (segundos)	Ganho (vezes)
SelectChunk (CFRK)	10. 427	—
SelectChunk (CFRK_OMP)	0.303	34.41

Tabela 1: Resultado obtido através do VTune para o CFRK e CFRK_OMP

rimentos demonstram que a paralelização ao nível dos múltiplos núcleos da *CPU* através da inserção de diretivas do *OpenMP* em um laço de repetição da função de cópia de dados do arquivo de entrada para uma estrutura de processamento gerou um ganho de 34 vezes para 12 núcleos em relação a execução com apenas um núcleo. Este ganho mostra que o ambiente *multicore*, além de aumentar a taxa de execução em paralela do algoritmo também ajuda a diminuir o tempo de acesso aos dados na memória *cache*, gerando um ganho "supralinear"para a função *SelectChunk*.

4.2. Resultado das otimizações no *módulo GPU*

O submódulo *kmer_kernel*, que executa na *GPU*, possui o *kernel ComputeIndex* que executa uma conversão dos valores dos *k-mers* da base 4 para a base 10 antes de realizar a contabilização da frequência de repetição dos *k-mers*. De forma a diminuir o espaço ocupado pelos dados na *cache* da *GPU*, realizamos a alteração do tipo do dado utilizado no *vetor de reads*. Originalmente este vetor era armazenado como *int* (inteiro), ocupando 4 *bytes* por elemento na memória. Este foi alterado para o tipo *char* (caractere), que ocupa 1 *byte* por elemento na memória. Essa alteração foi possível porque a linguagem *CUDA* é uma extensão da linguagem *C* e esta usa o tipo do dado apenas como referência ao espaço de alocação em memória permitindo fazer cálculos com variáveis de qualquer tipo primitivo.

Os resultados apresentados nesta seção são da execução do *kernel ComputeIndex* e foram obtidos através do *nvprof*. As Tabelas 2, 3, 4 e 5 apresentam os resultados da execução do *kernel ComputeIndex* gerados pelo perfilador *nvprof* para os valores de $k = \{2, 3, 4\}$ respectivamente. A primeira coluna apresenta o nome das versões do **CFRK** onde **CFRK_INT** se refere a versão original com o *vetor de reads* armazenado como inteiro e **CFRK_CHAR** se refere a nova versão com o *vetor de reads* armazenado como caractere. A segunda coluna apresenta a Ocupação que é a relação entre a média de *warps* ativos por ciclo e a quantidade máxima de *warps* suportada pela placa. A terceira coluna apresenta o IPC (*Instructions per cycle*) que é a quantidade de operações executadas por ciclo. A quarta coluna apresenta o valor médio de acessos repetidos à memória devido ao *cache miss*. A quinta coluna apresenta as Instruções por *warp* que é a média de instruções executadas por *warp*. A sexta coluna apresenta *Warps* por ciclo que é a quantidade média de *warps* ativos por ciclo. Ao observar as Tabelas 2, 3, 4 e 5 notamos uma grande diminuição da taxa de *Cache miss* e um consequente aumento do respectivo IPC da versão anterior, **CFRK_INT**, em relação a versão atual, **CFRK_CHAR**. Note que na versão **CFRK_INT** o valor de *cache miss* aumenta com proporção média de 3×10^{-3} e a versão **CFRK_CHAR** aumenta com proporção média de 3×10^{-4}. Por consequência deste fato

Versão	Ocupação	IPC	Cache miss	Inst. por Warp	Warps por ciclo
CFRK_INT	87.64%	2.80	0.00599	82.92	7.11
CFRK_CHAR	92.26%	4.07	0.00122	202.89	15.79

Tabela 2: Dados do perfilador para $k = 2$

Versão	Ocupação	IPC	Cache miss	Inst. por Warp	Warps por ciclo
CFRK_INT	89.30%	2.80	0.00956	103.90	6.68
CFRK_CHAR	93.96%	4.07	0.00175	282.84	17.72

Tabela 3: Dados do perfilador para $k = 3$

Versão	Ocupação	IPC	Cache miss	Inst. por Warp	Warps por ciclo
CFRK_INT	91.06%	2.77	0.01193	124.87	6.27
CFRK_CHAR	94.87%	3.94	0.00205	362.78	18.03

Tabela 4: Dados do perfilador para $k = 4$

Versão	Ocupação	IPC	Cache miss	Inst. por Warp	Warps por ciclo
CFRK_INT	91.58%	2.80	0.01362	145.85	7.55
CFRK_CHAR	94.55%	4.03	0.00224	442.73	17.93

Tabela 5: Dados do perfilador para $k = 5$

todas as outras taxas aumentam. A Ocupação aumentou 4% em média, o valor do IPC teve um aumento de aproximadamente 1.4 vezes, o número de Instruções por Warp teve um aumento de 3 vezes em média, o número de Warps por ciclo teve um aumento de 2.5 vezes em média.

Isto se deve pela diminuição da ocupação da memória. Ao trocar o tipo de dado inteiro, 4 *bytes*, para caractere, 1 *byte*, foi possível alocar 4 vezes mais dados nas memórias *cache* da *GPU*. Isso fez com que as requisições à memória global diminuíssem, aumentando a performance deste *kernel*.

4.3. Avaliação de desempenho do CFRK com 3 níveis de paralelismo

A Figura 1 apresenta o tempo de execução do **CFRK** da versão sem as otimizações apresentadas neste trabalho, identificada no gráfico como *orig*, em relação a versão com todas as otimizações apresentadas neste trabalho, identificada no gráfico como **CFRK_OMP_MG**. O eixo x apresenta o tempo de execução em segundos e o eixo y apresenta os valores de k. A Tabela 6 apresenta a redução do tempo obtido pelos aprimoramentos feitos na versão inicial do **CFRK**. A primeira linha apresenta os valores de k, a segunda linha apresenta o tempo em segundos e a terceira linha apresenta o ganho em porcentagem. Observando a Figura 1 e a Tabela 6 vemos que o tempo de processamento foi reduzido em mais de 80 segundos em todos os casos. A maior redução no tempo foi de 39.92% para k = 2, que executa menos operações de transferência de dados entre a *GPU* e o *host*. Conforme análise apresentada em trabalhos anteriores [Vilasboas et al. 2016], a medida que o valor de k aumenta, a complexidade do tempo de execução do *kernel* de contabilização de frequência de *k-mer* aumenta. Por outro lado, a medida que aumenta o valor de k aumenta o tamanho do vetor de contabilização de *k-mers* e consequentemente

Figura 1: Tempo de execução do CFRK e do CFRK_OMP_MG

k	2	3	4	5
Tempo (segundos)	90.19	81.16	84.06	88.53
Ganho (%)	39.92	35.40	34.99	29.58

Tabela 6: Tempo de processamento reduzido entre o CFRK_OMP_MG em relação ao CFRK

o tempo com a transferência de dados entre a *GPU* e o *host*, justificando a diminuição do ganho.

5. MCFRK

O **MCFRK** [Coelho et al. 2016] foi desenvolvido para possibilitar a execução de forma eficiente do **CFRK** com arquivos que possuem tamanho maior ou igual a memoria principal da máquina em execução. **MCFRK** é o acrônimo de **M**PI - **C**ontabilizador da **F**requência de **R**epetição de **K**-mers. Este algoritmo é uma extensão do **CFRK** para permitir a execução em ambientes de memoria distribuída, ou *cluster*, composta por diversos nós computacionais com placas aceleradoras *GPU's*. Para isso, foi utilizado o padrão de troca de mensagens *MPI*. A versão do **MCFRK** apresentada nesta seção integra todas as otimizações relatadas nas seções anteriores.

O **MCFRK** decompõe o arquivo original em arquivos menores para serem processados nos múltiplos nós de processamento da *cluster*. Foram adicionadas ao submódulo *main* do *módulo CPU* funções da biblioteca *MPI* para enviar, criar e gerenciar a execução de processos **CFRK** nos nós de processamento. A execução do **MCFRK** é dividida em duas fases: o pré-processamento e o processamento. Na fase de pré-processamento é realizada a divisão do arquivo. Esta divisão é realizada através de uma aplicação desenvolvida na linguagem C, chamada de split, que realiza o particionamento do arquivo em arquivos menores com o mesmo número de *reads*. Na fase de processamento e após a inicialização dos processos *MPI*, é fornecido a localização dos respectivos arquivos de entrada e de saída. Cada processo *MPI* realiza a execução de uma instância do **CFRK** e ao final do processamento os resultados do arquivo de saída são armazenados em disco.

5.1. Resultados e análise

Primeiramente vamos apresentar o resultado de cada uma das otimizações separadamente e posteriormente apresentar uma análise mais detalhada do **MCFRK**.

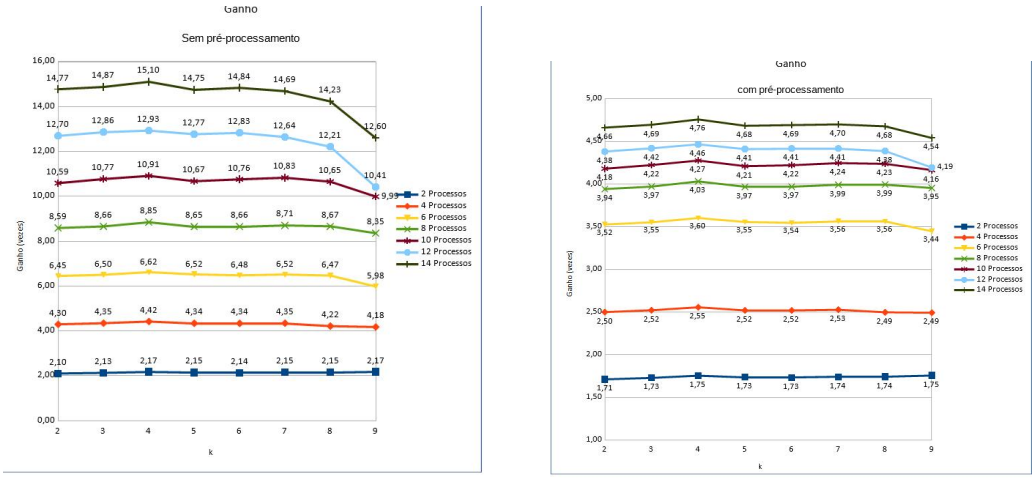
Para os experimentos apresentados na Tabela 7 foram utilizados 8 nós do supercomputador SDumont, onde cada nó possui as configurações apresentadas na Seção 4. O arquivo de entrada utilizado foi uma amostra real de metagenoma com 8.7GB de dados cuja identificação no banco de dados *SRA* do *NCBI* (*National Center for Biotechnology Information*) é *SRX2021688*. A Tabela 7 apresenta o tempo de execução e o ganho para cada otimização aplicada ao **MCFRK**. A primeira coluna apresenta a descrição da otimização; a segunda coluna apresenta os tempos de execução de cada versão com 1 processo *MPI*; a terceira coluna apresenta os tempos de execução de cada versão com 8 processos *MPI* sem a contabilização da operação de *split*; a quarta coluna apresenta os tempos de execução de cada versão com 8 processos *MPI* com a contabilização da operação de *split*; a quinta coluna apresenta o ganho de performance de cada versão sem a contabilização da operação de *split*; a sexta coluna apresenta o ganho de performance de cada versão sem a contabilização da operação de *split*.

Versão	1 processo	8 processos sem *split*	8 processos com *split*	Ganho sem *split*	Ganho com *split*
Sem otimização	160.39	20.4	70.545	7.86	2.27
Otimização módulo CPU	151.18	19.22	69.365	7.87	2.18
Otimização módulo GPU	155.97	19.99	70.135	7.80	2.22
Todas as otimizações	144.64	18.07	68.215	8,00	2.12

Tabela 7: Tempo de execução e ganho de cada otimização aplicada ao MCFRK

Para os experimentos apresentados nas Figuras 2a e 2b foram utilizados 14 nós do supercomputador SDumont, onde cada nó possui as configurações apresentadas na Seção 4. O arquivo de entrada utilizado foi uma amostra real de metagenoma com 15GB de dados cuja identificação no banco de dados *SRA* do *NCBI* (*National Center for Biotechnology Information*) é *SRX1142487*.

A Figura 2a apresenta o ganho obtido pela execução do **MCFRK** com múltiplos processos em relação a execução dele próprio com um processo sem incluir o tempo da fase de pré-processamento. A linha em azul escuro apresenta o ganho com 2 processos, a linha em laranja apresenta o ganho com 4 processos, a linha em amarelo apresenta o ganho com 6 processos, a linha em verde apresenta o ganho com 8 processos, a linha em vinho apresenta o ganho com 10 processos, a linha em azul claro apresenta o ganho com 12 processos, a linha em verde escuro apresenta o ganho com 14 processos. A Figura 2b apresenta o ganho obtido pela execução do **MCFRK** com múltiplos processos em relação a execução dele próprio com um processo incluindo o tempo de pré-processamento. A linha em azul escuro apresenta o ganho com 2 processos, a linha em laranja apresenta o ganho com 4 processos, a linha em amarelo apresenta o ganho com 6 processos, a linha em verde apresenta o ganho com 8 processos, a linha em vinho apresenta o ganho com 10 processos, a linha em azul claro apresenta o ganho com 12 processos, a linha em verde escuro apresenta o ganho com 14 processos. A diferença entre os ganhos de desempenho nas Figuras 2a e 2b, sem e com a operação split, mostram o impacto no desempenho do sistema causado pelas operações em disco. Conforme explicado anteriormente, a operação split é executada na fase de pré-processamento onde é feito o particionamento do arquivo



(a) Ganho do **MCFRK** sem o pré-processamento

(b) Ganho do **MCFRK** com o pré-processamento

Figura 2: Ganho do MCFRK

de entrada em arquivos menores para serem enviados para os nós de processamento da *cluster*. Isto constata o gargalo existente nos algoritmos que realizam muitas operações de leitura e escrita em disco.

Ao compararmos as Figuras 2a e 2b podemos observar que a medida em que aumentamos o número de processos, aumentamos o grau de execução em paralelo do algoritmo ao mesmo tempo em que diminuímos o tamanho do arquivo a ser processado em cada nó. Isto gera a diminuição no tempo de processamento por nó apresentada na figura 2a, porém o tempo de execução da operação split permanece o mesmo, aumentado a proporção do tempo de execução da operação split em relação as outras operações o que explica a diminuição do speed-up apresentada na figura 2b.

Demostramos que o **CFRK**, ao acrescentar o terceiro nível de paralelismo através da execução em uma *cluster* híbrida composta por nós de processamento com duas *GPU's* cada, é capaz de apresentar um ganho ”supralinear”para a maioria dos valores de k. O mesmo experimento mostra que ao contabilizar os gastos com o pré-processamento, a otimização em três níveis diminui e passa para um ganho de até 3.84 vezes com 14 nós de processamento, conforme a Figura 2b. Observamos que a medida em que aumentamos os nós de processamento, aumentamos o grau de execução em paralelo do algoritmo ao mesmo tempo em que diminuímos o tamanho do arquivo a ser processado em cada nó, ou seja, diminuímos a taxa de ”cache miss”gerando um ganho ”supralinear”no algoritmo **MCFRK**, conforme a Figura 2a. Por outro lado o tempo de execução da operação split se manteve constante, aumentado a proporção do tempo de execução da operação split em relação as outras operações e causando a diminuição no ganho total, conforme a Figura 2b.

Por final, podemos observar que o ganho de desempenho obtido com as otimizações implementadas neste trabalho possibilitou o aumento na eficiência do algoritmo **MCFRK** que passou a apresentar bom desempenho para valores de k até 9, conforme podemos observar nas Figuras 2a e 2b.

6. Conclusão e trabalhos futuros

Neste trabalho apresentamos otimizações desenvolvidas no algoritmo **CFRK**, originalmente desenvolvido para implementar paralelismo ao nível de *threads* em uma *GPU*, para permitir a execução com mais 3 níveis de paralelismo. Em um primeiro nível, utilizamos as diretivas do padrão *OpenMP* para permitir o processamento de módulos do algoritmo em múltiplos núcleos da *CPU*. Em um segundo nível, utilizamos as bibliotecas do modelo de programação *pthread* para permitir o processamento de mais de uma instância do *módulo GPU* em múltiplas *GPU's*. Por final, em um terceiro nível, utilizamos as bibliotecas do modelo de programação *MPI* para permitir a execução de múltiplos processos do algoritmo **CFRK** em nós de processamento de uma *cluster*.

Nossos experimentos demonstram que a paralelização ao nível dos múltiplos núcleos da *CPU*, para um nó de 12 núcleos, gerou um ganho de 34 vezes em relação a execução com apenas um núcleo. Este ganho mostra que a paralelização no ambiente *multicore*, além de aumentar a taxa de execução em paralela do algoritmo, diminui o tempo de acesso aos dados na memória, gerando um ganho "supralinear"para a função *SelectChunk*. Demonstramos também que ao acrescentarmos o segundo nível de paralelismo através da execução do algoritmo CFRK em duas *GPU's*, o tempo de execução diminuiu de 39% ao invés de 50% devido a latência de transferência de dados entre a *GPU* e a *CPU*. Demostramos que o **CFRK**, ao acrescentar o terceiro nível de paralelismo através da execução em uma *cluster* híbrida composta por nós de processamento com duas *GPU's* cada, também apresenta um ganho "supralinear"para a maioria dos valores de k devido ao aumento do grau de execução em paralelo do algoritmos e à diminuição do tamanho do arquivo a ser processado em cada nó e consequente diminuição da taxa de "cache miss". Por final, o experimento mostra que ao contabilizar os gastos com o pré-processamento, a otimização em três níveis tem um ganho muito menor, em até 4.78 vezes para 14 nós. Isto se dá porque apesar de aumentarmos o grau de execução em paralelo e diminuirmos a taxa de "cache miss", à medida em que aumentamos o número de nós de processamento, a latência da operação de transferência de dados se mantém constante, aumentando proporcionalmente o tempo de execução em transferência de dados em relação ao tempo de execução de processamento e causando uma diminuição no ganho total, conforme apresentado na Figura 2b.

Como trabalho futuro pretendemos desenvolver otimizações no **CFRK** de forma a diminuir o gargalo gerado pelas transferências de dados de entrada e saída. Pretendemos também avaliar o desempenho do **CFRK** em arquiteturas que apresentam um canal de alto desempenho para transferência de dados entre o host e a GPU tais como o nvlink da NVIDIA. Como trabalhos futuros também pretendemos acoplar o algoritmo **CFRK** a ferramentas para a análise de amostras de material genético para ser utilizado pelos usuários do portal de bioinformática do LNCC.

Referências

Church, P. C., Goscinski, A., Holt, K., Inouye, M., Ghoting, A., Makarychev, K., and Reumann, M. (2011). Design of multiple sequence alignment algorithms on parallel, distributed memory supercomputers. In *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 924–927. IEEE.

Coelho, M., Vilasboas, F., and Osthoff, C. (2016). Desenvolvimento de uma versão paralela híbrida para a contabilização da frequência de repetição de k-mers. *ERAD-RJ Escola Regional de Alto Desempenho do Rio de Janeiro*.

Edwards, R. A., Olson, R., Disz, T., Pusch, G. D., Vonstein, V., Stevens, R., and Overbeek, R. (2012). Real time metagenomics: using k-mers to annotate metagenomes. *Bioinformatics*, 28(24):3316–3317.

Erbert, M., Rechner, S., and Müller-Hannemann, M. (2017). Gerbil: a fast and memory-efficient k-mer counter with gpu-support. *Algorithms for Molecular Biology*, 12(1):9.

Fiannaca, A., La Rosa, M., Rizzo, R., and Urso, A. (2015). A k-mer-based barcode DNA classification methodology based on spectral representation and a neural gas network. *Artificial intelligence in medicine*, 64(3):173–84.

Garg, A., Jain, A., and Paul, K. (2013). GGAKE: GPU Based Genome Assembly Using K-Mer Extension. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 1105–1112. IEEE.

Marçais, G. and Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting ofoccurrences of k-mers. *Bioinformatics (Oxford, England)*, 27(6):764–70.

Plaza Onate, F., Batto, J.-M., Juste, C., Fadlallah, J., Fougeroux, C., Gouas, D., Pons, N., Kennedy, S., Levenez, F., Dore, J., Ehrlich, S. D., Gorochov, G., and Larsen, M. (2015). Quality control of microbiota metagenomics by k-mer analysis. *BMC genomics*, 16(1):183.

Vilasboas, F., Osthoff, C., Trelles, O., and Vasconcelos, A. T. (2015). Desenvolvimento de um algoritmo paralelo para contabilização da repetição de k-mers. *3ª Conferência Ibero Americana de Computação Aplicada 2015*.

Vilasboas, F., Osthoff, C., Trelles, O., and Vasconcelos, A. T. (2016). Otimização de um algoritmo paralelo para contabilização da repetição de k-mers. *II Escola Regional de Computação de Alto Desempenho do Rio de Janeiro*.

Using Petri-Net Modelling to Support the Case for HW-Assisted Task Scheduling

Lucas H. Morais[1], **Alfredo Goldman**[1], **Guido Araujo**[2]

[1]Instituto de Matemática - Universidade de São Paulo (USP)

[2]Instituto de Computação - Universidade Estadual de Campinas (Unicamp)

{moraislh,gold}@ime.usp.br, guido@ic.unicamp.br

Abstract. *Given the pervasiveness of multi-core processors in systems from various domains, the need for efficient parallelization tools has only increased during the last decade. Among the paradigms built to answer this demand, Task Parallelism stands out as a highly productive tool for leveraging data parallelism with minimum code altering. Nonetheless, its current supporting runtimes cannot efficiently execute workloads involving tasks in the fine 1-100us range, limiting its applicability. That said, by performing a thorough Petri-Net-based analysis of task parallel systems with several degrees of HW-assistance, we show that the development of Native CPU support for Task Parallelism is the key for efficiently serving these challenging workloads.*

1. Introduction

Since the dawn of the dark silicon era around 2005, designers have relied more and more on multi-core processor architectures to keep delivering aggregate performance improvements over time [Borkar and Chien 2011]. Even though the switch to multi-core architectures could not fully overcome the challenges posed by the disruption of Dennard Scaling, the benefits of such architectures on power efficiency and computational throughput have paved their way towards becoming the standard for modern processor design. As a matter of fact, multi-core processors can now be found everywhere from low-cost mobile devices to the HPC realm, with 16+ core processors being already available in the desktop market.

Nonetheless, if software engineers from the golden age of Dennard Scaling could always hope for their programs to get automatic performance improvements at the release of every new processor generation, single-threaded performance increase rate has gone down considerably since then [Borkar and Chien 2011]. In order to leverage the still increasing computing power of current processors, programmers are now demanded to exploit their multiple cores, vectorization features, superscalar pipelines, etc., in ways that may not be trivial [Asanovic et al. 2009]. In fact, the sole task of developing and maintaining programs capable of benefiting from all cores available on a system may be very daunting, given that:

1. Constructs provided by some parallel programming frameworks can obfuscate program logic and require extensive code reorganization.
2. The concurrency inherent to parallel software makes it harder do debug.
3. Some kinds of computations - like MD5 computation, LZMA compression, etc - are inherently serial, limiting parallelization or requiring algorithmic relaxation.

Among the various existing parallelization paradigms, *Task Parallelism* stands out as a generic and very productive tool for solving the first of these issues for programs from a great variety of domains [Gupta and Sohi 2011]. On the other hand, current software support for task parallelism is not able to efficiently handle task parallel workloads involving tasks in the small 1-100us size range [Kumar et al. 2007, Meenderinck and Juurlink 2010]. That being the case, by modelling systems with several degrees of HW-support for this paradigm, our work shows that adding native CPU support for Task Parallelism could enable even these fine-grained task workloads to be efficiently executed. This finding suggests that the Task Parallelism paradigm should boast even broader applicability if proper HW-support is provided.

This paper is organized as follows: in Background and Related Work (2), we give a brief overview of the Task Parallelism paradigm, define a three-class taxonomy of Task Scheduling Systems that should be useful for understanding prior work on this area and also define some relevant terminology; in Evaluating the Limitations of Current Task Scheduling Systems (3) we analyze the shortcomings of the current Task Scheduling Systems that do not rely on any HW-assistance for Task Scheduling; in The Case for Native Task Scheduling (4), by modelling Task Scheduling Systems with Petri Nets, we show how adding native support for Task Parallelism to CPUs could dramatically improve the performance of fine-grained task workloads with respect to all systems displaying a lower degree of HW-assistance; at last, in Conclusion (5), we draw our final remarks.

2. Background and Related Work

Task Parallelism frameworks allow programmers to require a runtime system to execute certain basic code blocks or function calls in an asynchronous manner. The code regions allowed to be run in such way are called *tasks*. By specifying how these tasks depend on variables available at the moment they were created, the programmer lets the runtime to automatically infer, at execution time, the dependence relationships between these tasks and run them in parallel. Support for this paradigm is currently offered by several frameworks, such as OpenMP 4.0, Intel TBB, Intel Cilk, BSC's OmpSS, etc.

The performance limitations of Software-based Task Scheduling Runtimes led to the development of several Task Scheduling Systems that, by incorporating some degree of hardware acceleration, allowed for improved scheduling performance. This section thus begins by presenting a three-class taxonomy that groups relevant Task Scheduling Systems from the literature according to their degree of HW-assistance for scheduling computation, while also highlighting the main shortcomings and advantages of each organization. Afterwards, we finish the section by displaying a glossary of relevant terms related to Task Scheduling.

2.1. Software-based Task Scheduling (SW-TS)

In a Software-based Task Scheduling system, task dependence inference is performed by a CPU-based software runtime, of which the Intel OpenMP Runtime [Intel 2013] and the GNU OpenMP Runtime [GNU 2013] are examples.

This is the simplest Task Scheduling configuration, since it only depends on the availability of a software runtime. Nonetheless, as the analysis of Section 3 shall demonstrate, the performance of these systems is severely degraded when they are used to serve

task applications generating fine-granularity tasks - that is, tasks with execution times in the range from 1 to 100us.

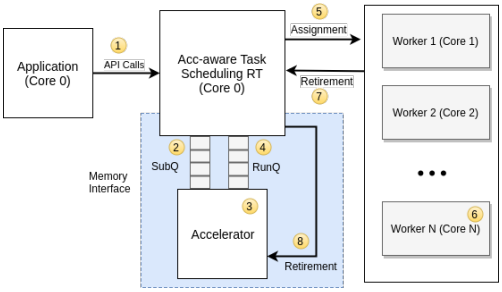


Fig. 1. ACC-TS organization

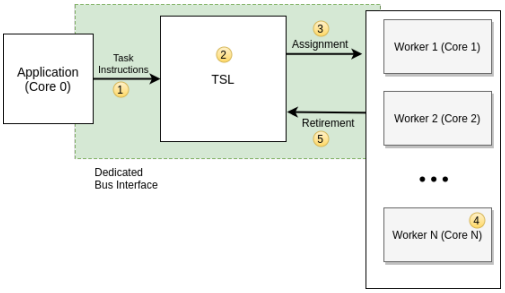


Fig. 2. Native-TS organization

2.2. Accelerator-based Task Scheduling (ACC-TS)

Accelerator-based Task Scheduling systems aim to improve Task Scheduling performance by implementing several scheduling actions in an FPGA-based accelerator, which interacts with task applications through the API provided by a lightweight SW Runtime [Yazdanpanah et al. 2015, Dallou et al. 2013, Wang et al. 2013, Bamnote and Nerkar 2015, Dallou et al. 2016]. Such organization is depicted in Fig. 1.

For these systems, maximum task throughput is expected to exceed that of SW-TS systems. Yet, as it is usual for computation-offloading systems [Vuduc et al. 2010], their performance is heavily dependent on the characteristics of the data paths connecting the CPU hosting the runtime and the FPGA holding the accelerator. Such systems excel at serving workloads related to denser task dependence graphs, which are more demanding of dependence resolution, but display poor performance for workloads related to sparse task dependence graphs, due to the significant cost of CPU-FPGA communication.

2.3. Native Task Scheduling (Native-TS)

In Native Task Scheduling systems, HW-support for Task Parallelism is provided by the processor itself, which makes it available as an ISA extension. Tasking constructs of high-level task parallel programs may thus be translated to tasking instructions during compilation time. In such systems, since Task Scheduling Logic dwells within the processor itself, task scheduling actions do not involve the traversal of data through data paths external to the CPU, effectively eliminating the main bottleneck of ACC-TS systems.

In the diagram of Fig. 2, we see how several elements involved in the execution of a task parallel program interact with each other in such a system. The application generating tasks engages the CPU's Task Scheduling Logic (TSL) through task-creation instructions. The TSL is then left responsible for (1) assigning tasks to available cores in agreement with the restrictions imposed by the inferred task dependence graph and (2) taking notice of cores signaling the conclusion of their tasks, triggering a task graph update and potentially making more tasks ready for assignment. The figure also depicts the fast dedicated bus interface that connects all those elements - the Application, the TSL and the cores running tasks (worker cores) - within the processor.

Given that the execution of task parallel programs by such systems does not require interaction with any software runtime, not only RT-ACC communication overheads

are eliminated, but also all overheads that arise from the communication between the application and such a runtime through its software API.

Although some works on ACC-TS systems have pointed before to the benefits that solutions like this could bring [Dallou and Engelhardt 2015], to the best of our knowledge, no Native-TS system has yet been implemented.

2.4. Relevant definitions

Task granularity refers to the mean execution time of a task. High granularity corresponds to long execution times, while low granularity relates to short execution times.

Task retirement is the action by which a core finishing the execution of some task informs the Task Scheduling Runtime about its conclusion. Once a task is retired, its corresponding node is removed from the task dependence graph and new tasks may become ready to be executed.

Task submission is the action by which the task parallel program creates tasks and sends their descriptors to a Task Scheduling Runtime.

In-flight task is a task that is currently referenced by the Task Dependence Graph - that is, a task that has already been submitted but that has not yet been retired.

Ready task is a task T such that there is no in-flight task T' such that T depends on T'.

Task assignment is the action by which the Task Scheduling Runtime assigns a ready task to an available processor, which immediately starts to execute it.

Task allocation is the action by which the Task Scheduling Runtime allocates enough space in memory for holding a single task metadata structure.

Run queue is a software queue for holding descriptors of ready tasks.

3. Evaluating the Limitations of Current Task Scheduling Systems

This section aims to assess the performance limitations of current Task Scheduling systems. In order to do so, we devise performance metrics (Sec. 3.1) that allow us to formulate performance upper bounds (Sec. 3.2) for systems of such kind. Once in place, we apply such tools to the analysis of a representative SW-TS Runtime (Sec. 3.3), leading us to conclusions about the intrinsic limitations of the SW-TS organization (Sec. 3.4).

3.1. Performance metrics

Mean Overhead Time (Eq. 1) is the sum of all average scheduling overheads related to the processing of a single task.

Maximum Dispatch Rate (Eq. 2) is the maximum task completion rate that the whole set of worker cores would be able to achieve if no scheduling overheads existed.

Mean In-flight Time (Eq. 3) is the average lifetime of the tasks generated by an application, considering both mean execution time and scheduling overheads.

Maximum Task Throughput (MTT) (Eq. 4) is the maximum number of tasks that a Task Scheduling system may be able to retire during an interval of time.

Overhead ratio (Eq. 5) is the mean fraction of in-flight time due to scheduling overhead.

$$T_{ovh} = T_{addition} + T_{RunQDeq} + T_{alloc} + T_{retirement} \quad (1) \qquad MDR = \frac{N_{wcores}}{T_{exec}} \quad (2)$$

$$T_{inflight} = T_{ovh} + T_{exec} \quad (3) \qquad MTT = \frac{1}{T_{ovh}} \quad (4) \qquad O_{ratio} = \frac{T_{ovh}}{T_{inflight}} \quad (5)$$

3.2. Devising Theoretical Limits to Application Speedup

3.2.1. From Overhead Ratio

Let us consider a task scheduling system with N cores for which an Overhead Ratio equal to O_{ratio} was measured for a certain workload W. In such situation, *assuming that all cores may perform scheduling actions* and given that W has enough intrinsic parallelism not to let cores starve for work, the maximum program speedup is attained when all cores are fully utilized. In such optmal case, any core has a chance of O_{ratio} of being, at a certain moment, performing scheduling actions. That being the case, the average number of cores that will actually be executing tasks will be:

$$N_{executing-tasks} = (1 - O_{ratio}) \times N, \tag{6}$$

which corresponds to the maximum application speedup MS_{ratio} for that scenario.

3.2.2. From MTT

Let us consider a task scheduling system with N cores for which an MTT of K was measured for a certain workload W. Tasks contained in W display mean task execution time equal to T_{exec}. Assuming that in this case *only one core is allowed to perform scheduling actions*, the Task Scheduling Runtime may not serve more than K tasks per unit of time. If we then assume that worker cores are able to keep running tasks without any scheduling overhead, each of these cores should be able to run as much as T_{exec}^{-1} tasks per second. From these assumptions, Ineq. 7 can be derived, leading to the application speedup upper bound of Eq. 8.

$$\frac{N_{executing-tasks}}{T_{exec}} \leq K, \tag{7}$$

$$MS_{MTT} = K \times T_{exec} \tag{8}$$

3.3. Evaluating SW-TS overheads

In this subsection, we instrument a software Task Scheduling Runtime running on a dual-core x86 machine to analyze its performance while serving several scheduling-related actions: (1) processing newly created tasks, which demands a node addition to the task graph; (2) task allocation; (3) task retirement, which demands a node to be removed from the task graph; (4) run-queue queuing.

For simplicity, the Task Scheduling Runtime of choice was MTSP [César 2016], a light-weight open-source implementation of the tasking provisions of OpenMP 4.0. Similar work could be done with the Intel OpenMP Runtime or with the GNU OpenMP Runtime. Experiments were performed on an Intel Core i5-3230M dual-core CPU @2.60GHz, with 16GB of DDR3 RAM running on Ubuntu 16.04.

In order to instrument MTSP's code, Intel's RDTSC and RDTSCP cycle-counting instructions in GCC Asm were used to enclose code fragments for each scheduling-related action of interest. The instrumented runtime was then evaluated while serving two task-parallel applications from the Kastors Benchmark Suite [Virouleau et al. 2014]. Each measurement value is the average resulting from ten experiment replications.

Task Granularity	Jacobi		SparseLU	
	E. Params	Cycles	E. Params	Cycles
small	-n 8192 -b 64	8.0e+03	-n 128 -m 64	9.4e+02
medium	-n 1024 -b 64	6.9e+04	-n 128 -m 16	2.3e+04
large	-n 128 -b 64	3.8e+05	-n 128 -m 4	1.5e+06

Table 1. Execution parameters for each benchmark

The overhead ratios measured for all six experiments according to Eq. 5 are summarized in Fig. 3. We can see that for fine-granularity workloads - for which, according to Table 1, mean task execution time is not higher than 8K cycles - overhead ratios are over 80%. On the other hand, as task granularity increases to about 380K cycles, overhead ratios become lower than 10% for both test programs.

These findings support our hypothesis that, although current task scheduling systems may be fast enough to serve coarse-granularity task workloads, they fail at serving applications generating smaller tasks, which is indicated by the high overhead ratios measured for both the *Small* and *Medium* workloads.

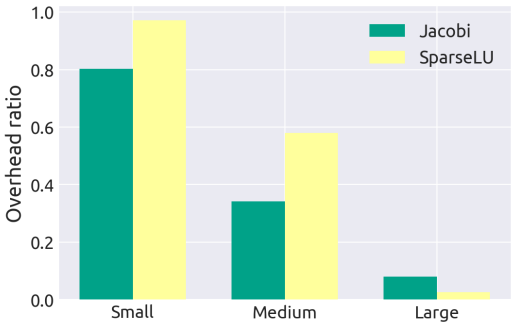


Fig. 3. Relationship between task granularity and overhead ratio

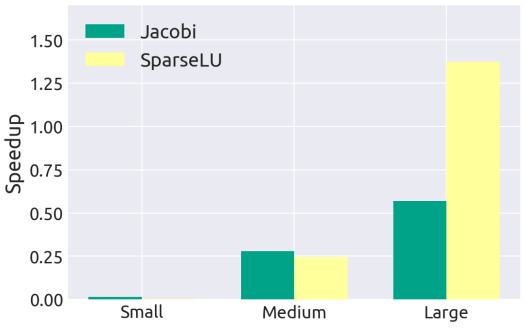


Fig. 4. Relationship between task granularity and program speedup

Fig. 5 showcases the components of scheduling overheads for all six experiments. The amount of overhead cycles reported for each (scheduling action, experiment) pair is an average value calculated from the set of all tasks produced by the experiment. This breakdown shows that actions related to task graph management (addition and retirement) are the most taxing, while allocation and run-queue dequeuing never take more than 10% and 5% of total per-task overhead, respectively. Such figure also suggests that both (1) relative weights of scheduling latency components and (2) mean total per-task overhead do not vary much depending on task granularity or task graph topology, since they remain very similar across all six experiments. One major implication of this is that one should not expect total per-task overhead to be lower than about 30K cycles for any task parallel application with data dependencies, regardless of its mean task granularity.

The speedup graph of Fig. 4 shows that application speedup will usually improve as workload granularity increases, going up to 1.37x for SparseLU-large (T_{exec} of about 1.5×10^6 cycles), the most coarse-grained workload. Nonetheless, slowdowns as high

Fig. 5. Overhead components for each experiment

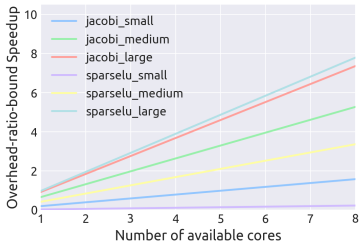


Fig. 6. Overhead-ratio-driven Speedup Bounds

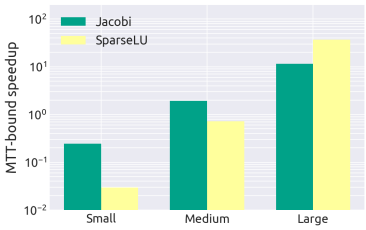


Fig. 7. MTT-driven Speedup Bounds

as 100x for Jacobi-small (T_{exec} of about 8.0×10^3 cycles) and 200x for SparseLU-small (T_{exec} of about 9.4×10^2 cycles) were also measured, illustrating the inability of SW-TS systems to handle fine-grained workloads.

Moreover, as shown by the optimistic theoretical speedup upper-bounds of Fig. 6 - which considers a perfectly parallelized TS Runtime - supposing a system with 8 cores, the optimal speedup figures for the fine-grained workloads would not be higher than 1.6 for Jacobi-small and 0.2 for SparseLU-small. By its turn, the more realistic evaluation of Fig. 7 - which supposes a serial TS Runtime - shows that even if an unbounded number of cores was available, one would not be able to achieve speedups higher that 0.25 and 0.029 for Jacobi-small and SparseLU-small, respectively.

3.4. Conclusions about SW-TS performance

From the data just showed and analyzed, one can come to the following conclusions regarding SW-TS performance:

1. SW-TS systems perform best for task applications generating tasks larger than about 1×10^6 cycles, which take around 500us in a 2GHz processor. For finer workloads, the performance of the parallel program may be considerably worse than that of its serial counterpart.
2. Mean total per-task overhead does not vary much depending on task granularity for task applications generating tasks with data dependencies.
3. The most computationally taxing actions performed by a SW-TS RT are those related to task graph management - namely, addition and retirement.

4. The Case for Native Task Scheduling

The theoretical analysis of Section 3 demonstrated the limitations of current Task Scheduling Systems relying on software runtimes. This section, in turn, aims to show how these shortcomings may be avoided by the introduction of enough hardware acceleration. Moreover, we show that while ACC-TS systems from the literature already displayed substantial gains with respect to organizations without any HW-assistance, Native Task Scheduling should represent a major step forward from this organization, since it circumvents its communication-related bottlenecks.

This section hence starts with the description of our proposed Petri Net Model for Task Scheduling Systems and finishes with the evaluation of the three organizations from

Section 2 according to this model, from which we can gather conclusions regarding their relative performance and their intrinsic limitations.

4.1. Modelling Task Scheduling Systems with Petri Nets

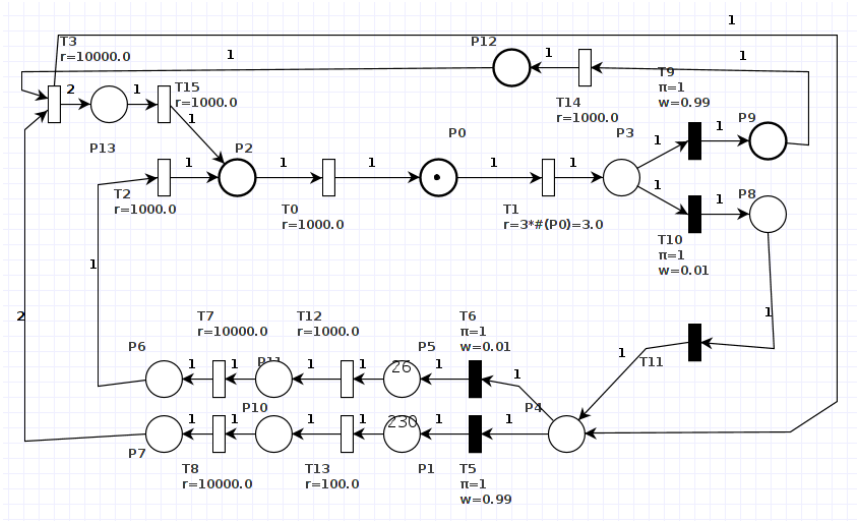


Fig. 8. ACC-TS Petri Net model

In this section, we present a Petri Net model for an ACC-TS system with realistic characteristics, which may be used to evaluate system bottlenecks and resource utilization as well as to predict application speedup. We start by discussing the reasons that lead us to organize the net topology and set the transitions parameters as they are. Then, we show how one can use the same net for simulating SW and Native TS systems by setting its transition rates accordingly. All experiments involving Petri Nets were performed using the PIPE4 simulation tool [Dingle et al. 2009], and the developed models can be found at a public Git [1]. Table 2 describes the parameters that can be varied across simulations.

High-level simulation feature	Dependent features
System characteristics (SW, ACC, Native)	System organization (SW, ACC, Native)
	number of CPU cores; timing characteristics
	presence/absence of communication queues
Workload characteristics	Task granularity; parallelism degree
	percentage of dependence-less tasks

Table 2. Tunable simulation features.

4.1.1. System topology analysis

Since their generation at the task parallel code, tasks go through the various sub-systems of a HW-accelerated Task Scheduling system according to the flow depicted in the system block diagram of Fig. 1. In the following lines, we show how each of the indicated steps were mapped to place-transition combinations in our model.

[1] https://bitbucket.org/lucashmorais/task-scheduling-petri-net-modeling

(1-2) Task-creating API calls and writes to the Submission Queue are modeled by places {P1, P5, P10, P11} and transitions {T5, T6, T12, T13}. The number of marks in P5 represent the number of dependence-less tasks that the system was required to schedule, while the number of marks in P1 represent the number of tasks with dependencies that the application would like the system to manage. Additionally, the number of marks in P11 represents the number of task descriptors of dependence-less tasks that have already been enqueued to the Submission Queue. Analogously, the number of marks in P10 encodes the number of tasks with dependencies that the system may already start to process. The transition weights of T5 and T6 define the simulated workload ratio between dependence-free and dependence-full tasks, while the transition rates of T13 and T12 define the simulated rate at which the RT may fill the Submission Queues with task descriptors for tasks with or without data dependencies, respectively.

(3) Task dependence inference at the accelerator is represented in our model by places {P6, P7} and transitions {T7, T8}. The number of marks in P6 represent the number of dependence-less tasks on the task graph, while P7, the number of dependence-full tasks on the graph. The transition rate for T7 represents the rate at which the accelerator may add dependence-less tasks to the graph, while the rate for T8, the rate at which it may process dependence-full tasks.

(4) Writing to the Runnable Queue is represented in our model by P2 and the transitions T2 and T15. The number of marks in P2 represents the number of tasks on the Runnable Queue, while the identical rates of T2 and T15 represent the rate at which the queue may be written to.

(5) Assignment of tasks to the processor cores is represented by P0 and T0. The number of marks in P0 represent the number of processors that are currently running a task. The rate for T0 represents the rate at which task descriptors may be read from the Runnable Queue and assigned to an available core.

(6) Task execution is represented by transition T1. Its rate represents the rate at which tasks are executed, which is a function of the number of currently active cores and of task granularity. Such relationship is depicted in Fig. 8 below T1.

(7-8) Task retirement is modeled by places {P3, P8, P9, P12, P13} and transitions {T1, T3, T9, T10, T14}. The number of marks in P9 represent the number of tasks on which other tasks depended that have recently finished executing, while the number of marks on P8 represent the number of tasks on which no other task depended that were recently finished. The number of marks in P12 represent the number of task IDs that have been recently written to a HW-register that receives IDs of retiring tasks. The rate for T14 represents the rate at which packets may be written to such register. The rate for T3 represents the rate at which the system may detect two tasks to be ready to execute as a result of the acknowledgement that one more task has finished executing.

4.1.2. Reasoning behind used rates and place occupancy restrictions

All rate parameters of our Petri Net model may be changed for better reflecting its target (TS System, workload) pair. In the specific configuration depicted in Fig. 9, we consider that tasks are $10\text{K}/3 \approx 3.3\text{K}$ cycles long. The amount of cycles NC_{action} related to each

of the remaining modeled actions may be calculated with the following formula:

$$NC_{action} = \frac{10000}{R_{action}}, \tag{9}$$

where R_{action} is the rate of the related timed transition.

The rates for transitions representing memory operations correspond to reasonable memory access latencies for modern x86 machines. The SW and ACC models abstract several sources of overheads that would make their actual implementations less efficient, while pessimistic latency values are set for the Native-TS model. Furthermore, we even consider the Runtimes behind SW and ACC systems to be able to perform several different scheduling actions in concurrent fashion, which, although desirable, would be difficult to implement for real systems. By doing so, our comparison between the Native and the other configurations is made more favorable towards the latter ones, making any perceived advantages of the Native TS system more reliable. The weights for the instantaneous transitions, on their turn, are set in such a way that a workload comprising 99% of tasks with dependencies and 1% of dependence-less tasks was simulated. Such ratio between the two task classes correspond to a very taxing task parallel workload, which goes in line with our goal of comparing the performance of the three system organizations under the most stressful circumstances. Finally, we set the limit on the number of marks on P0 to 64, which goes to represent that our system has a 64-cores CPU.

4.2. Models for SW and Native systems

We now showcase two petri nets that may be used to simulate the execution of the workload of Fig. 9 by SW and Native task scheduling systems. They share the same topology of the former model, differing only by the rates of some of their timed transitions.

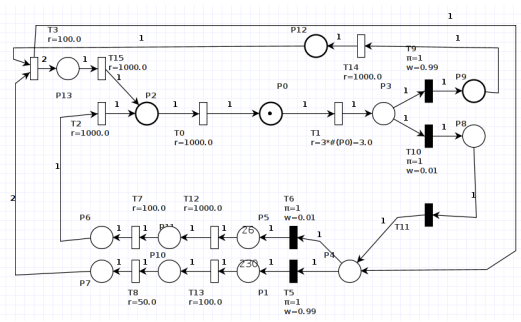


Fig. 9. SW-TS model

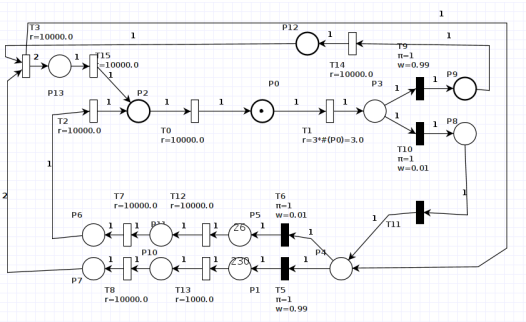


Fig. 10. Native-TS model

4.3. Experimental evaluation

We now present information regarding the number of cores that each of these solutions was found to be able to feed, as well as the subsystems that we discovered to be the bottlenecks of each organization. Results for both analysis are in Table 3.

Organization	TCO	System Bottleneck	C. Places
SW	17	Task-scheduling kernel	P11
ACC	33	SW-Queues Comm. interface	P1, P13, P12
Native	64	Limited number of CPU cores	P7, P0, P13, P2

Table 3. Typical Core Occupancy, System Bottlenecks and Congested Places for each organization

As discussed in Section 4.1.1, the number of cores currently running tasks during simulation relates to the number of marks in P0. Hence, the Typical Core Occupancy (TCO) of each scenario corresponds to most frequent value of marks of P0 after steady state has been achieved. Such metric represents a close upper-bound for program speedup.

System bottlenecks were evaluated by identifying, for each simulation model, the modelled system elements corresponding to places with highest typical occupation.

The conducted set of experiments suggests that Native-TS systems should bring a large leap in performance with respect to SW and ACC systems. In fact, the simulated ACC system was only able to increase TCO with respect to the SW system by a factor of 1.9 times, while the Native system was able to effectively saturate the available processor cores with work. These findings agree with our expectations that, by (1) circumventing the high overhead ratios of SW-TS systems and (2) avoiding the communication latencies of the FPGA-CPU buses and the queue interfaces of ACC-TS systems, the Native-TS organization should enable significantly higher performance.

Similar conclusions can be derived from experiments involving a higher ratio between dependence-less and dependence-full tasks (not depicted). These scenarios lead to significantly better performance for SW-TS systems. Concretely, if the dependence-less/dependence-full task ratio is increased to 50/50, Typical Core Occupation rises by about 50% for the latter organization. All the same, the performance of other configurations is not considerably altered by this change - given that the bottlenecks of ACC and Native TS systems are not related to slow task-graph management - and the SW-based systems keep their place at the bottom of the performance scale.

5. Conclusion

In this work, we have shown that the development of Native CPU support for Task Parallelism should enable efficient execution of task parallel workloads comprising tasks on the fine 1-100us size-range, for which existing task scheduling systems have presented only poor results. In order to do so, we developed a parametric Petri Net model and theoretical upper bounds for predicting performance characteristics of Task Scheduling systems under arbitrary workloads. With that, we believe to have motivated further research on Native Task Scheduling Systems - that is, systems where full support for Task Parallelism is provided by the CPU itself - for realizing the potential of the Task Parallelism Paradigm as not only a very productive, but also as a highly efficient parallelization tool.

6. Acknowledgements

The authors thank the support of FAPESP (2017/02682-2) for this research undertaking.

References

Asanovic, K., Wawrzynek, J., and Patterson, D. (2009). A view of the parallel computing landscape. In *Communications of the ACM*, pages 56–67.

Bamnote, R. and Nerkar, R. P. (2015). Review on Dynamic Task Scheduling to Support OoO Execution in an MPSoC Environment. In *Int'l Journal of Computer Applications*.

Borkar, S. and Chien, A. (2011). The Future of Microprocessors. In *Comm. of the ACM*.

César, D. (2016). MTSP: Multicore Task Scheduling Platform. https://bitbucket.org/lgeunicamp/mtsp/.

Dallou, T., Elhossini, A., and Juurlink, B. (2013). FPGA-Based Prototype of Nexus++ Task Manager. In *6th Wksh. on Many-Task Computing on Clouds Grids and Supercomputers*.

Dallou, T. and Engelhardt, N. (2015). Nexus#: A Distributed Hardware Task Manager for Task-Based Programming Models. In *IEEE 29th Int'l Parallel and Distributed Processing Symposium (IPDPS)*.

Dallou, T., Lucas, D. S., Araujo, G., Morais, L., Frank, M., and Ferreira, E. (2016). Task Parallel Programming Model + Hardware Acceleration = Performance Advantage (poster). In *Hot Chips*.

Dingle, N. J., Knottenbelt, W., and Suto, T. (2009). PIPE2: A Tool for the Performance Evaluation of Generalised Stochastic Petri Nets. In *ACM SIGMETRICS Performance Evaluation Review*, pages 34–39.

GNU (2013). An OpenMP implementation for GCC. http://gcc.gnu.org/projects/gomp.

Gupta, G. and Sohi, G. (2011). Dataflow Execution of Sequential Imperative Programs on Multicore Architectures. In *Proc. 44th IEEE/ACM Int'l Symp. on Microarchitecture*.

Intel (2013). Intel OpenMP Runtime Library. https://www.openmprtl.org.

Kumar, S., Hughes, C., and Nguyen, A. (2007). Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *Proc. 34th annual Int'l Symp. on Computer architecture*.

Meenderinck, C. and Juurlink, B. (2010). A Case for Hardware Task Management Support for the StarSS Programming Model. In *Proc. 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*.

Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., and Gautier, T. (2014). Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In *10th Int'l Workshop on OpenMP*, pages 16 – 29.

Vuduc, R., Chandramowlishwaran, A., and Choi, J. (2010). On the Limits of GPU Acceleration. In *Proc. 2nd USENIX conference on Hot topics in parallelism*, page 13.

Wang, C., Li, X., and Zhang, J. (2013). MP-Tomasulo: A Dependency-Aware Automatic Parallel Execution Engine for Sequential Programs. In *ACM Transactions on Architecture and Code Optimization (TACO)*, pages 1–9.

Yazdanpanah, F., Álvarez, C., and Jiménez-González, D. (2015). Picos: A hardware runtime architecture support for OmpSs. In *Future Generation Computer Systems*.

Análise de Zonas Térmicas em *Data Center* Não-CRAC

Ademir Camillo Junior, Charles C. Miers, Guilherme P. Koslovski, Maurício A. Pillon

[1]Programa de Pós-Graduação em Computação Aplicada (LabP2D/PPGCA – DCC)
Universidade do Estado de Santa Catarina (UDESC) – Joinville/SC – Brasil
ademir.junior@edu.udesc.br,
{mauricio.pillon,charles.miers,guilherme.koslovski}@udesc.br

Resumo. *A elevada concentração de equipamentos em Data Centers (DCs) é objeto de estudo para administradores, fabricantes (processadores, servidores e sistemas de refrigeração), entre outros. Dentre os desafios da área, destaca-se o melhoramento da eficiência energética destes ambientes. No contexto de DC, a Power Usage Effectiveness (PUE) é uma referência na mensuração da eficiência energética. Este trabalho apresenta a arquitetura MonTerDC, um sistema de monitoração de temperatura de DC baseado em sistemas de refrigeração não-CRAC. Como resultados, o trabalha apresenta o uso do MonTerDC na identificação de zonas térmicas indesejáveis (fora da norma). Com este mapeamento térmico em zonas, o administrador do DC pode aplicar práticas à melhor distribuição física dos nós de computação e, consequentemente, reduzir a temperatura das zonas térmicas.*

Introdução

O fornecimento crescente de serviços de Tecnologia da Informação (TI) gera a necessidade de incremento de poder computacional, atualmente, concentrados em DCs. Independente do número de equipamentos compondo um DC (pequeno, médio ou grande porte), o gerenciamento energético é um desafio recorrente. Pesquisas indicam que aproximadamente 1,3% de toda a energia elétrica gerada atualmente é consumida por DCs [Song et al. 2015]. Esse valor é composto, principalmente por dois elementos: infraestrutura de TI, representando 52%, e sistemas de suporte, constituindo 48% [Ahuja et al. 2011]. Dentre os componentes dos sistemas de suporte, a climatização representa aproximadamente 39% desta fatia e afeta diretamente os custos de manutenção [Song et al. 2015].

Além de quantificar o consumo energético por categoria, determinar uma métrica que considere a relação entre os componentes de um DC, é essencial para quantificar a eficiência energética destes ambientes. O consórcio *Green Grid*[1] estabeleceu métricas e políticas tanto para reduzir o consumo energético quanto para aumentar a eficiência energética. A métrica PUE é uma referência neste contexto e pode ser obtida pela razão entre o consumo total do DC e o consumo total dos equipamentos de TI. Quanto menor for o PUE, mais eficiente é o DC [Avelar et al. 2012]. Recentemente, DCs de grande porte divulgaram valores das suas médias trimestrais de PUEs, *e.g.,* Google e Facebook informaram 1,10 e 1,07, respectivamente, como melhores casos [Horner and Azevedo 2016]. Por outro lado, a média mundial em 2014 foi de aproximadamente 1,7 [Sverdlik 2014].

O consumo de energia dos equipamentos de TI e sistemas de suporte podem variar, segundo as condições de temperatura do ambiente, afetando o grau de eficiência do

[1]https://www.thegreengrid.org/

DC mensurado pelo PUE. Um dos componentes do sistema de suporte mais afetado com a variação de temperatura do ambiente é o sistema de refrigeração. No que se refere a temperatura externa, algumas organizações decidiram implantar seus DC em regiões com temperaturas médias anuais inferiores as médias globais, embora estes ambientes fiquem geograficamente distantes de seus clientes [Google 2012] [Facebook 2017].

Por outro lado, a temperatura interna de DCs sofre influência de outros fatores, tais como o projeto de sistema de refrigeração, a localização física dos equipamentos de processamento e de rede, ou ainda da carga de trabalho dos equipamentos de TI em geral. DCs de médio e grande portes são, normalmente, implantados de acordo com projetos complexos e específicos de alvenaria, elétrica e TI. Estes ambientes, em sua maioria, são climatizados por *Computer Room Air Conditioner* (CRAC), nos quais o ar frio é injetado por baixo, através de piso elevado perfurado, conduzido por dentro de *racks* de servidores, principal fonte geradora de calor em um DC, e expelido pela parte superior. O controle do fluxo de ar frio direcionado a fontes de geração de calor torna o modelo CRAC eficiente [Arghode and Joshi 2013], entretanto, exigem um elevado custo de implantação, se comparados com sistemas simples. Em resumo, o custo de implementação é um dos principais fatores para baixa utilização em DCs de pequeno e médio porte. No Brasil, apenas 8% dos DCs utilizam CRAC [Schneider 2014].

Neste trabalho, DCs constituídos de sistemas de refrigeração formados por condicionadores de ar e poucos pontos de injeção de ar, se comparado com CRAC que possui algumas dezenas de entradas/saídas direcionadas e projeto de condução pressurizada das correntes de ar frio e quente, são nomeados de *não-CRAC*. Sistemas de refrigeração *não-CRAC*, maioria dos DCs brasileiros, são mais suscetíveis a má distribuição das correntes de ar e a formação de zonas térmicas indesejáveis, não respeitando as normas de padronização [ASHRAE 2016] [Fulpagare et al. 2016].

Alguns fatores que podem influenciar na formação de zonas térmicas quentes que excedam as especificadas nas normas, são: a má distribuição físicas dos equipamentos de TI, a ausência de condução das correntes de ar frio / quente ou a variação da carga de processamento dos equipamentos. Os dois primeiros fatores referem-se ao projeto físico de concepção do DC, são fatores estáticos, pois o administrador não costuma deslocar seus servidores após a instalação física e tampouco direcionar as aletas do condicionador de ar de acordo com a carga do DC. O último fator, variação da carga de processamento, pode ser considerado pelo administrador no momento de provisionamento de recursos ou balanceamento de carga. Neste caso, o administrador define o servidor hospedeiro de acordo com a necessidade de processamento solicitada e a distribuição térmica atual entre as zonas. Portanto, ao provisionar um novo recurso, ele pode escolher um servidor X, localizado em uma zona Y, em detrimento a um servidor W, de igual poder computacional, localizado em uma zona Z, pelo fato da zona Z ter temperatura mais elevada do que a zona Y. Desta forma o equilíbrio térmico do DC é mantido.

O monitoramento térmico em tempo real é uma importante ferramenta para auxiliar a tomada de decisão do administrador do DC. A arquitetura *Monitoramento Término de Data Centers* (MonTerDC), proposta deste trabalho, é constituída de três módulos independentes e pode ser integrada a diversos *frameworks* gerenciadores. A MonTerDC foca em ambientes de DC de pequeno e médio porte, sem sistemas de refrigeração complexos que seguem o modelo CRAC. Os resultados da monitoração das zonas térmicas

no DC do LabP2D, um DC de pequeno porte e uma única fonte de ar frio, permitem quantificar o impacto do provisionamento de recursos na formação de zonas indesejáveis. Com o equilíbrio térmico no ambiente, evitando zonas térmicas indesejáveis, o sistema de refrigeração é menos exigido e, consequentemente, o consumo energético total do DC é reduzido.

Este artigo está organizado em quatro seções. Trabalhos correlatos são discutidos na Seção 2. A arquitetura MonTerDC é apresentada na Seção 3, com a descrição dos componentes e da ferramenta para monitoração de zonas térmicas. A Seção 4 apresenta os experimentos, descrevendo o cenário, o método de testes e a análise dos resultados coletados. A Seção 5 finaliza o trabalho e apresenta perspectivas para continuação.

Trabalhos correlatos

O monitoramento térmico é uma prática reconhecida em DCs de grande porte, cujo sistema de refrigeração segue o modelo CRAC. A Tabela 1 apresenta a comparação entre os trabalhos relacionados. Dentre eles, as soluções diferem quanto ao objetivo, a aplicação e ao componente observado. Alguns trabalhos observam o comportamento do processador, do servidor, do *rack* ou da totalidade do DC.

Autor	Nível	Coleta de dados	Aplicação	Objetivo
[Tang et al. 2008]	*data center*	modelo / CFD	simulação	circulação de ar
[Pakbaznia et al. 2010]	*data center*	-	real	balanceamento de carga
[Lei et al. 2011]	*rack*	sensoriamento	real	predição
[Ahuja et al. 2011]	*rack*	CFD	simulação	contenção no *rack*
[Alkharabsheh et al. 2014]	*data center*	modelo / CFD	simulação	saída de ar quente
[Bottari 2014]	*data center*	sensoriamento	real	predição / prevenção
[Wibron 2015]	*data center*	CFD	simulação	localização física
[Zhang et al. 2015]	*data center*	sensoriamento	real	pressão do ar
[Gao et al. 2016]	*rack*	-	real	fluxo de ar

Tabela 1. Trabalhos relacionados cronologicamente ordenados.

Com relação ao nível de atuação do monitoramento e refrigeração pode-se identificar soluções a nível de *rack* ou para o DC como um todo. Gao et al. 2016 abordam o comportamento do fluxo de ar internamente ao *rack* através de um experimento real, entretanto, não se preocupam com o posicionamento dos equipamentos e como a localização interna afeta o aquecimento. Lei et al. 2011 fazem uma análise semelhante, utilizando sensores para realizar uma predição de cenários em tempo real. Por outro lado, através de simulação, Ahuja et al. 2011 analisam o fluxo do ar internamente ao *rack* utilizando *Computational Fluid Dynamics* (CFD), porém não aborda as diferentes cargas de trabalho em diversos cenários.

Outro conjunto de trabalhos busca fornecer informações, através de simulações, para ambientes de DCs controlados. Neste caso, o DC não necessariamente está operacional, portanto, pode ser interessante para a concepção de projetos de DC. A simulação através da Dinâmica dos Fluidos Computacionais, CFD é utilizada para prever as velocidades dos fluídos e suas respectivas temperaturas ou para modelar as correntes de ar

frio/quente em ambientes de DC [Marshall and Bemis 2011, Wibron 2015]. Através de simulações, Ahuja et al. 2011 indicou como utilizar o CFD para analisar um ambiente de DC de alta disponibilidade para identificar a diferença de temperatura quando é utilizado contenção nos *racks*, evitando que o ar quente re-circule no DC. Por sua vez, Tang et al. 2008 apresentou um modelo de circulação de ar para DC visando diminuir o custo com climatização, diminuindo entre 2 e 5 graus a temperatura média, com economia de 20% a 30% no consumo de energia. Entretanto, ambos não relacionaram as cargas de trabalho dinâmicas que ocorrem nestes ambientes de alto desempenho. De forma complementar, [Alkharabsheh et al. 2014] utilizou CFD em um DC para prever a temperatura de saída do ar quente para ajustar o sistema de refrigeração.

Analisando o método de coleta dos dados, quando observada a utilização de sensores, [Lei et al. 2011], propuseram o ThermoCast, uma solução de monitoramento com sensores que realiza o mapeamento de zonas térmicas em um *rack*, para identificar as zonas de calor e predizer o superaquecimento individualizado em cada estrutura. Entretanto, nesta solução, uma visão macro do ambiente não é possível, atendo-se apenas ao nível de *rack*. Também utilizando sensores, Bottari 2014 apresenta uma arquitetura para monitoração de DCs através de *Internet of Things* (IoT), com objetivo de identificar e prever o superaquecimento, porém, não se preocupando com as zonas de calor ou a influência entre elas.

Por outro lado, para gerenciar a temperatura do ar frio injetado, Zhang et al. 2015 apresentam um sistema para utilização em CRAC, que através da coleta de informações de temperatura em tempo real e mapeamento de zonas, permite regular a quantidade de ar frio injetado no sistema em cada área gerenciada. Ainda, Gao et al. 2016 propõem uma unidade de refrigeração e circulação de ar individualizada para cada *rack* que permite a coleta do ar frio do sistema de ventilação CRAC e envio direto ao *rack*, gerando um fluxo de ar frio específico para cada área.

Como alternativa para a redução do consumo de energia, pode-se realizar a migração de máquinas virtuais para outros servidores ou locais físicos. Em [Pakbaznia et al. 2010] o objetivo é a redução de energia através do balanceamento de carga, distribuindo processos entre vários servidores e mantendo suas cargas equivalentes. A técnica utilizada por [Alkharabsheh et al. 2014] segue a lógica oposta, utiliza-se da consolidação de processos no menor número de servidores possível. Ambos convergem no ambiente de aplicação, DCs de produção e CRAC, entretanto, não se preocupam com as novas zonas térmicas originadas pelo balanceamento das cargas e variação da temperatura.

Por fim, é importante ressaltar que os trabalhos relacionados focam em ambientes CRAC, sobretudo na pressão do ar injetado no sistema com objetivo de otimização do resfriamento e menor consumo de energia, mas não discutem os locais que precisam um maior fluxo de ar frio devido a uma temperatura maior e variável. Outro aspecto relevante é o foco no consumo de energia dos equipamentos de TI, sem preocupação dedicada ao consumo de energia com climatização, diretamente influenciado pelas zonas de calor geradas pelo aquecimento dos equipamentos de TI que estão sendo gerenciados.

Monitoramento Térmico de *data center* não-CRAC (MonTerDC)

A concentração de potência de cálculo em DCs é uma realidade. A monitoração destes ambientes é essencial tanto para administradores quanto para usuários que buscam

eficiência no uso dos recursos em prol de suas aplicações ou, simplesmente, otimização de custos. Neste contexto, este trabalho descreve uma proposta de arquitetura para um ambiente de monitoração de DCs de pequeno e médio porte. A proposta de arquitetura do MonTerDC, ambiente de Monitoramento Térmico de *data centers*, tem por objetivo fornecer um mapa térmico dinâmico em tempo real de DCs não-CRAC. Este ambiente consiste da definição de módulos de hardware e de software que, integrados, permitem a identificação e mapeamento de zonas térmicas.

As zonas térmicas podem ser formadas por uma ou mais fontes geradoras de ar frio ou quentes e não possuem dimensões máximas ou mínimas. Em ambientes de DCs com sistemas de refrigeração não-CRAC, as correntes de ar frio e quente não são guiadas por projetos de canaletas, tubos, pisos elevados ou injeção de ar. A formação de zonas indesejáveis, que excedam as temperaturas máximas especificadas nas normas, são afetadas pela intensidade do gerador de calor ou pelo número de geradores concentrados, pela pressão da injeção de ar e pela umidade. Nestes ambientes, onde o controle dos fatores que influenciam na formação de zonas indesejáveis não são controlados, a identificação de zonas térmicas indesejáveis torna-se mais complexa. A característica dinâmica dos fatores, o deslocamento das zonas e a interferência entre elas dificultam ainda mais a identificação. Esta realidade é específica de DCs com sistemas de regrigeração não-CRAC, pois as zonas térmicas em DCs CRAC são bem definidas e controladas, com temperaturas baixas na parte inferior e quentes superiores, variando apenas os valores de acordo com a carga de trabalho dos servidores ativos [Athavale et al. 2016].

A arquitetura do MonTerDC é constituída de três módulos principais: Central de Coleta (C), Central de Processamento (CP) e o 3D Viewer. A relação entre estes módulos está ilustrada na Figura 1. O primeiro módulo, a Central de Coleta é composta de sensores especializados na captura dos fatores externos de formação das zonas térmicas, tais como temperatura, umidade e pressão atmosférica. Com capacidades de armazenamento e processamento limitadas, a Central de Coleta limita-se a aquisição das informações do ambiente de acordo com as especificações parametrizadas pelo administrador. O administrador define o intervalo de captura, tipo da informação, a faixa de erro tolerada nas capturas e a necessidade ou não de pré-processamento das informações. Caso o pré-processamento seja ativado, o administrador deve informar o método estatístico de sua preferência (média, mediana, etc) que deve ser aplicado a estes dados.

A Central de Processamento cabe a manipulação dos dados coletados em tempo real, associando-os a infraestrutura física do DC. Os dados individualizados por sensor são etiquetados (*tag*) e armazenados em um banco de dados. Este módulo dispõe da estrutura física do DC, localização de servidores, das saídas do sistema de refrigeração e dos sensores. Neste trabalho, uma zona térmica não tem uma dimensão física mínima ou máxima e deve ser formada por pelo menos um sensor. O número de sensores por zona térmica varia em função do número de fontes geradoras de calor ou frio e da precisão dos sensores. A dimensão de uma zona térmica é determinada pelo espaço físico com temperatura homogênea considerando o erro de precisão dos sensores. De posse destas informações, o submódulo Simulador CFX gera o cenário atual do ambiente de DC determinando as correntes de ar frio e quente, a intensidade destas correntes no ambiente e as zonas térmicas com suas respectivas temperaturas.

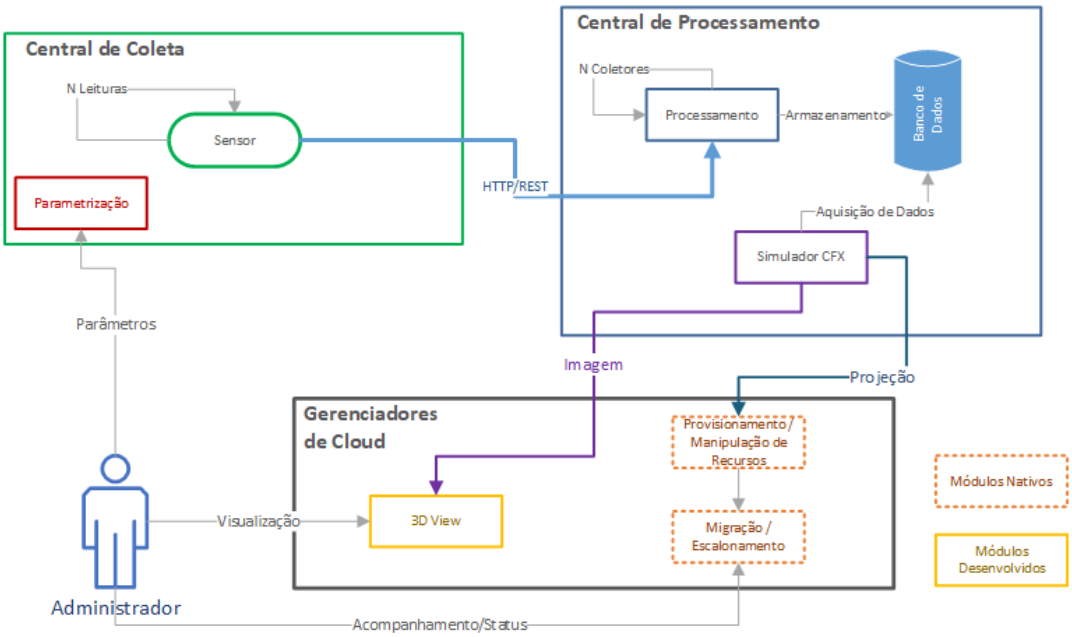


Figura 1. Arquitetura MonTerDC.

O último módulo, 3D View, é o responsável pela visualização do mapa térmico do ambiente. Ele está integrado a um gerente de recursos, de nuvem como ilustrado na Figura 1, ou de *data centers*. Com a visualização em tempo real no 3D View das zonas térmicas em ambiente de *data center* não-CRAC, a arquitetura MonTerDC auxilia o administrador de *data center* de pequeno e médio porte.

Estudo de Caso: LabP2D

O Laboratório de Processamento Paralelo e Distribuído (LabP2D) está vinculado a Universidade do Estado de Santa Catarina (UDESC), campus Joinville. O LabP2D possui um espaço físico no andar térreo de um prédio de três andares, originalmente projetado para acolher uma sala de aula. O ambiente possui sistema de climatização tradicional do tipo Split (Electrolux - 30.000 btus), fonte geradora fria, com uma única saída. Ele encontra-se localizado na parte superior da sala e o sistema de direcionamento do fluxo de ar é feito através de aletas. Neste ambiente, o LabP2D abriga o seu DC de pequeno porte que conta atualmente com duas fileiras de máquinas servidoras do tipo torre e um *rack* de chão com três *switches* e três servidores de produção do tipo *rack*. Cada fila de servidores do tipo torre contam com 10 máquinas HP Proliant (processador AMD Phenom II X4 B93 2,8GHz, 4 núcleos, 8GB RAM, 500GB HD, Fonte 150W). Os servidores de produção instalados no *rack* possuem a seguinte configuração: 2 processadores Intel Xeon E5-2600, 24 núcleos, 292GB RAM, 2TB HD, fonte 740W redundante. Atualmente, este DC hospeda a Nuvem Tche cujo o gerenciador de recursos é o OpenStack (*Ocata Release*). Na Figura 2, pode-se observar a modelagem do ambiente físico em 2D, representando as duas fileiras de servidores torre HP, na horizontal, e o *rack* com os servidores de produção e *switches* no meio a esquerda.

A implantação do sistema de Monitoramento Térmico de DC (MonTerDC) no LabP2D respeitou as restrições impostas pelo ambiente. Na Figura 2(a), ilustra-se o posicionamento físico dos sensores numerados de 1 a 14. Na Figura 2(b) é possível identificar

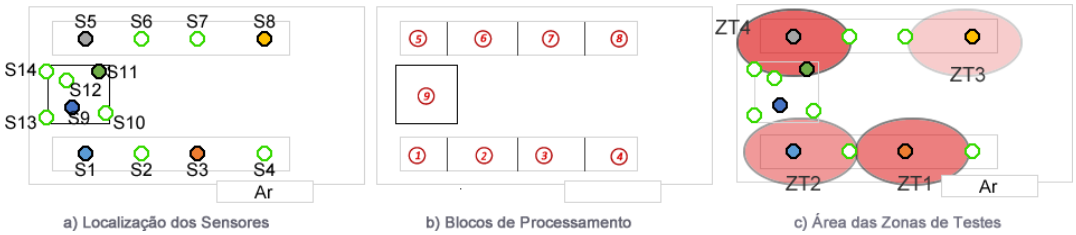


Figura 2. Estudo de caso LabP2D

os blocos de processamento. No caso das fileiras superior e inferior, cada bloco (de 1 à 4 e de 5 à 8) é constituído de 4 servidores HP. O bloco 9, localizado no meio a esquerda, é constituído dos 3 servidores de produção e 3 *switches*. Finalmente, na Figura 2(c) estão representadas as zonas térmicas identificadas pelo MonTerDC. Os sensores de temperatura são do tipo LM35DZ, possuem precisão de 1º Celsius e estão posicionados a uma distância máxima entre eles de 1 metro. Desta forma, foi possível identificar as mudanças de temperatura entre duas ou mais zonas de calor simultaneamente.

Os nós de processamento foram agrupados em blocos de processamento, com o objetivo de alternar as fontes geradoras de calor variando a carga de trabalho destas máquinas entre 0% e 100%. Como o objetivo do teste é observar o impacto das fontes de calor nas zonas, definiu-se que o bloco de processamento central, número 9 formado pelo *rack*, permanecesse com 100% de carga de trabalho. Os quatro cenários de testes realizados seguem a descrição da Tabela 2 que varia a carga de processamento por bloco.

Tabela 2. Configuração da carga de processamento no cenário de teste.

Cenário	Carga de Processamento				
	Bloco 1	Bloco 3	Bloco 5	Bloco 8	Bloco 9
Cenário 1	0%	100%	0%	0%	100%
Cenário 2	100%	0%	0%	0%	100%
Cenário 3	0%	0%	0%	100%	100%
Cenário 4	0%	0%	100%	0%	100%

O objetivo da variação de cenários (Tabela 2) é observar o comportamento das zonas térmicas. A geração de calor é intensa e constante no bloco 9 para todos os cenários. A escolha de intensificação e suavização alternada dos blocos 1, 3, 5 e 8 buscam identificar o comportamento das zonas nas fileiras de servidores HP. Observa-se que a posição do sistema de refrigeração favorece a circulação de ar frio em alguns blocos, por exemplo bloco 8, não atuando na mesma forma no bloco 1.

A bateria de testes com duração de 15 minutos cada foi aplicada a todos os cenários. Ela compreende de uma fase de aquecimento, próximo a 3 minutos e de estabilização/observação da temperatura do bloco (12 minutos). A cada nova bateria, aplica-se uma fase de resfriamento do ambiente, onde todos os blocos tem carga de 0% por 15 minutos. Finalmente, como saída do sistema (Figura 3), os dados coletados pelo ambiente MonTerDC geram figuras do ambiente de DC em 3D com o auxílio do software $ANSYS\ CFX$ 17.2.

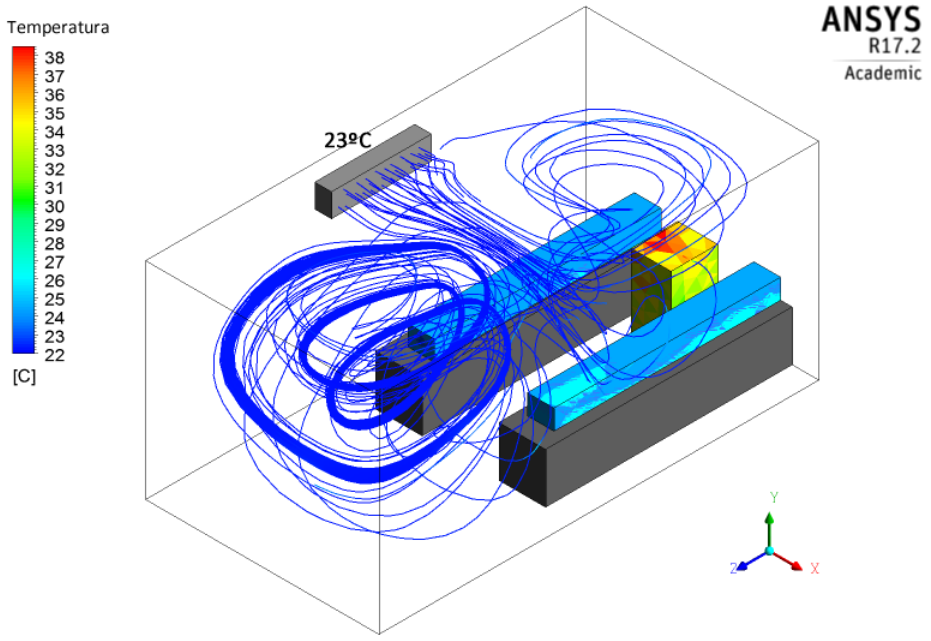


Figura 3. Resultado de saída da simulação do ambiente de testes.

Na Figura 3, tem-se um exemplo da modelagem do fluxo de corrente de ar frio do ambiente do LabP2D modelado com o software $AnsysAcademic$ 17.2. A integração do MonTerDC com o $ANSYS$ que permite a visualização das zonas sobrepostas ainda está em fase de testes.

Resultados

Nesta seção são descritos os resultados da observação do mapeamento térmico do LabP2D, através do sistema MonTerDC, nos quatro cenários propostos. Os aspectos analisados foram: formação de zonas térmicas indesejáves, o impacto de um bloco de processamento a seus vizinhos e a diferença de comportamento das zonas térmicas entre blocos com a mesma carga, porém em localizações físicas diferentes.

A execução da carga de processamento de acordo com a definição de blocos e cenários, atuou no ambiente do LabP2D com a formação de quatro zonas de observação, uma para cada bloco de processamento envolvido. As zonas foram nomeadas de Zonas Térmicas (ZT) de $ZT1$ à $ZT4$ de acordo com os blocos, respectivamente, identificados com 3, 1, 8 e 5 (Figura 2(c)). Os resultados analisam o comportamento nas zonas formadas em função do processamento dos blocos envolvidos nos cenários e o impacto do processamento destes blocos na temperatura do bloco 9, bloco central de *rack* com carga constante.

A Figura 4, identifica as temperaturas obtidas nos sensores $S1$, $S3$, $S5$, $S8$, $S9$ e $S11$, eixo y, e o horário de início e térmico de cada bateria de testes, eixo x. A escolha dos sensores se deu em função dos objetivos da análise destes testes. Os primeiros quatro correspondem aos sensores centrais dos blocos de processamentos envolvidos e, os últimos dois, localizados no bloco 9, são os afetados pela zona de término de seus vizinhos. A mensuração dos testes seguiu um período contínuo de observação, iniciando-se as $9{:}42$

e terminando as 11:38. Presume-se que as condições de temperatura, umidade e pressão externas, que podem influenciar nos resultados, são constantes para todos os cenários. Finalmente, os retângulos tracejados observados ao longo do eixo x demarcam o período de aplicação dos cenários.

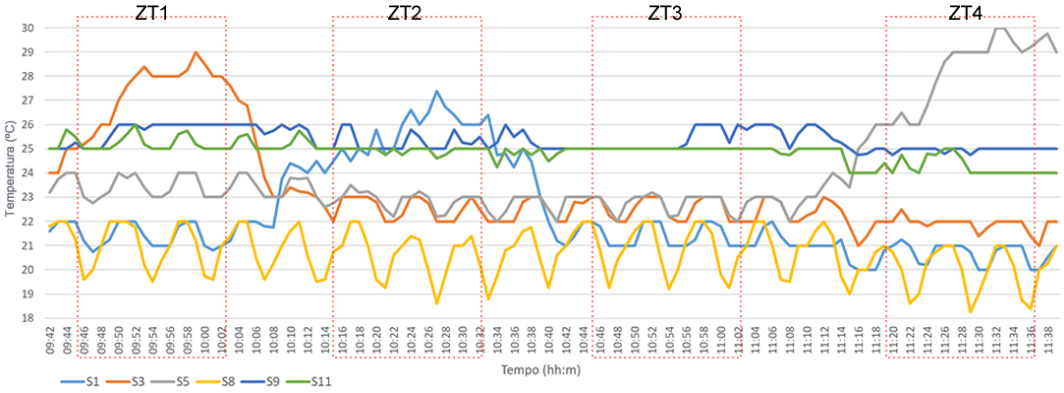


Figura 4. Mapeamento das Zonas Térmicas (ZT) no ambiente de testes.

Observando o comportamento das temperaturas (Figura 4), pode-se constatar que o impacto da carga de processamento não afeta da mesma forma todos os blocos. O bloco 8, medido pelo sensor $S8$, é o menos afetado pela sua própria carga de processamento ou pela carga de seus vizinhos. A sua temperatura varia entre pouco mais de 18º e 22º, com ondulações regulares, que correspondem as oscilações do fluxo de ar do sistema de refrigeração. Observando-se as correntes de ar frio, apresentadas na Figura 3, constata-se que a localização do bloco 8 recebe correntes de ar frio com maior intensidade.

O bloco de processamento 5 indica um comportamento totalmente oposto, medido pelo sensor $S5$. A faixa de temperatura deste bloco é entre 22º e 30º, atingindo portanto temperaturas superiores as especificadas nas duas normas abordadas. No período de repouso, com carga de processamento 0%, este bloco atinge sua temperatura mínima de 22º nas fases de resfriamento e nos cenários em que o fluxo de refrigeração não é afetado ($ZT2$ e $ZT3$). Porém, a ativação do bloco de processamento 3, incrementou a temperatura do bloco 5 em 1º. Embora o bloco 3 não seja seu vizinho, a sua ativação alterou o fluxo de corrente de ar frio central que refrigera o bloco 5. Com esta nova fonte geradora de calor, o bloco 5 já prejudicado, se comparado com os demais, foi ainda mais afetado. Finalmente, com a ativação da sua carga de processamento, o bloco 5 atingiu as temperaturas mais altas do teste, chegando a 30º, e formando uma zona indesejável.

Com temperaturas mais amenas, se comparadas as do bloco 5, o comportamento dos blocos 1 (sensor $S1$) e 3 (sensor $S3$) são semelhantes ao do bloco 5. As temperaturas do bloco 1 variam entre 21º e pouco mais de 27º e as do bloco 3 entre 21º e 29º, ambos formando zonas indesejáveis quando a carga de processamento é aplicada a seu respectivos blocos. No entanto, dadas as suas localizações físicas e a distribuição das correntes de ar frio, eles não são afetados por nenhuma outra fonte geradora de calor, mas afetam outras. A única diferença observada entre os blocos 1 e 3 é que o bloco 1 afeta a zona térmica de seu vizinho direto (bloco 9). Já o bloco 3 alterou o fluxo da corrente de ar influenciando a temperatura do bloco 5.

Os últimos dois sensores, $S9$ e $S11$, monitoraram o bloco 9 cujo a carga de processamento não variou. As temperaturas neste bloco tiveram pouca variação, ficando entre 24^{o} e 26^{o}. No entanto, suas maiores temperaturas, mesmo dentro da norma, ocorreram quando o bloco 1, vizinho direto, foi ativado ou quando o bloco 8, distante físicamente, ao intensificar sua geração de calor, afetou a corrente de ar frio que chegava ao bloco 9.

A aplicação do ambiente MonTerDC ao estudo de caso do LabP2D, um DC de pequeno porte não-CRAC, revelou a importância da proposta deste trabalho na identificação da formação dinâmica das zonas térmicas. A formação de zonas térmicas indesejáveis, ou mesmo de zonas térmicas dentro da norma mas com temperaturas altas, tem influência direta no consumo de energia e na durabilidade dos equipamentos de TI e sistemas de suporte. A análise dos resultados, através do MonTerDC, permitiram identificar as zonas térmicas visualmente e, principalmente, os fatores de interferências entre estas zonas. Normalmente, a influência térmica entre vizinhos é considerado por administradores, porém a alteração do fluxo de refrigeração, devido a intensificação ou suavização de fontes geradoras de calor, é um fator mais complexo de identificar sem o auxílio de ferramentas como o MonTerDC.

Considerações & Trabalhos futuros

A concentração de equipamentos de TI em DCs facilita a gerência destes recursos, contando com equipes especializadas e infraestrutura dedicada. Porém, o comportamento dos equipamentos de TI varia conforme a carga de processamento e é influenciado por fatores, tais como: temperatura, pressão e umidade. A monitoração destes fatores em tempo real, por si só, já é um desafio, mas compreender a relação entre eles não é trivial. Para o administrador, entender esta relação é essencial, pois a simples decisão de alocar um bloco de processamento em uma zona A pode afetar diretamente a temperatura da zona vizinha B ou a corrente de ar que refrigera a zona C. Enfim, estas relações são dinâmicas, não necessariamente evidentes e impactam no consumo de energia total do DC e a vida útil dos equipamentos.

Este trabalho apresentou a arquitetura do MonTerDC, um sistema de Monitoramento Térmico para *data centers* de pequeno e médio porte não-CRAC. Embora, DCs com sistemas de refrigeração não-CRAC sejam a realidade de $\sim 92\%$ dos DCs implantados no Brasil [Schneider 2014], poucos trabalhos científicos tratam do problema. O MonTerDC apoia-se em uma malha de sensores de temperatura, vinculando a estrutura física estática, localização dos equipamentos e a natureza dinâmica do fluxo de refrigeração formado por geradores de correntes de ar frio e quente. Com estas informações, o MonTerDC modela as zonas térmicas, identificando zonas indesejáveis (com temperaturas superiores as especificadas nas normas) e, através de um simulador de CFD, gera uma imagem 3D com as zonas térmicas em tempo real.

Os resultados reforçam a importância de ambientes de monitoração térmica neste contexto. Comprovou-se a influência entre vizinhos, por exemplo, a variação da carga de processamento do bloco 1 afetou a temperatura do bloco 9, mas também identificou-se o impacto no fluxo de ar frio gerado pela variação da carga do bloco 8 ou do bloco 3. A alteração do fluxo de ar frio afetou a temperatura tanto do bloco 9 quanto do bloco 5. Embora os blocos de processamento das fileiras serem formados pelo mesmo número de máquinas, no caso 4 servidores de torre HP, e terem a mesma carga de processamento,

ora 0% ora 100%, as zonas térmicas formadas em seus entornos possuem diferenças de até 7^oC. Observou-se claramente a existência de blocos suscetíveis às variações de cargas de outros blocos, tais como blocos 1, 3, 5 e 9, e outros, no caso dos testes, somente o bloco 8, totalmente indiferente a atuação dos demais. O protótipo da arquitetura MonTerDC está operacional e mostrou a sua importância na identificação de zonas térmicas no contexto a que se propõe. No âmbito do protótipo, os principais pontos de melhora focam na integração do MonTerDC ao gerente de recursos em nuvem OpenStack e na automatização do módulo de simulação CFX.

Agradecimentos

Os autores agradecem o apoio do Laboratório de Processamento Paralelo e Distribuído (LabP2D) no Centro de Ciências Tecnológicas (CCT) da Universidade do Estado de Santa Catarina (UDESC) e a Fundação de Amparo à Pesquisa e Inovação do Estado de Santa Catarina (FAPESC).

Referências

Ahuja, N., Rego, C., Ahuja, S., Warner, M., and Docca, A. (2011). Data center efficiency with higher ambient temperatures and optimized cooling control. In *2011 27th Annual IEEE Semiconductor Thermal Measurement and Management Symposium*, pages 105–109.

Alkharabsheh, S., Sammakia, B., Shrivastava, S., and Schmidt, R. (2014). Implementing rack thermal capacity in a room level CFD model of a data center. In *2014 Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM)*, pages 188–192.

Arghode, V. K. and Joshi, Y. (2013). Modeling Strategies for Air Flow Through Perforated Tiles in a Data Center. *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 3(5):800–810.

ASHRAE (2016). Ashrae tc9.9, data center networking equipment – issues and best practices.

Athavale, J., Joshi, Y., Yoda, M., and Phelps, W. (2016). Impact of active tiles on data center flow and temperature distribution. In *2016 15th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm)*, pages 1162–1171.

Avelar, V., Azevedo, D., and French, A. (2012). PUE^{TM}: A Comprehensive examination of the metric.

Bottari, G. D. (2014). *Monitoramento Térmico Responsivo para Centros de Processamento de Dados*. Master Degree, Universidade Federal Fluminense, Niteroi/RJ - Brasil.

Facebook (2017). Lulea datacenter. In *https://www.facebook.com/LuleaDataCenter*.

Fulpagare, Y., Shirbhate, P., and Bhargav, A. (2016). Design and testing of prototype data center. In *15th IEEE ITHERM Conference*, Indian Institute of Technology Gandhinagar.

Gao, T., Kumar, E., Sahini, M., Ingalz, C., Heydari, A., Lu, W., and Sun, X. (2016). Innovative server rack design with bottom located cooling unit. In *2016 15th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm)*, pages 1172–1181.

Google (2012). Google datacenters - from paper mill to data center. In *https://www.google.com/about/datacenters/inside/locations/hamina/*.

Horner, N. and Azevedo, I. (2016). Power usage effectiveness in data centers: Overloaded and underachieving. In *The Electricity Journal - 29*, pages 61–69.

Lei, L., Liang, C., and Liu, J. (2011). Thermocast: A cyber-physical forecasting model for data centers. In *17th ACM SIGKDD Conference on Knowledge Discovery*, San Diego, California, USA.

Marshall, L. and Bemis, P. (2011). Using CFD for data center design and analysis. *Applied Math Modeling White Paper*.

Pakbaznia, E., Ghasemazar, M., and Pedram, M. (2010). Temperature-aware dynamic resource provisioning in a power-optimized datacenter. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 124–129.

Schneider, E. (2014). Soluções em climatização para data center. Brasília/Brasil. XIV Encontro Nacional de Empresas Projetistas e Consultores da Abrava.

Song, Z., Zhang, X., and Eriksson, C. (2015). Data Center Energy and Cost Saving Evaluation. *Energy Procedia*, 75:1255–1260.

Sverdlik, Y. (2014). Survey: Industry average data center pue stays nearly flat over four years. Uptime Institute.

Tang, Q., Gupta, S. K. S., and Varsamopoulos, G. (2008). Energy-Efficient Thermal-Aware Task Scheduling for Homogeneous High-Performance Computing Data Centers: A Cyber-Physical Approach. *IEEE Transactions on Parallel and Distributed Systems*, 19(11):1458–1472.

Wibron, E. (2015). *CFD Modeling of an Air-Cooled Data Center*. Master Degree, CHALMERS University of Technology, Gothenburg/Sweden.

Zhang, S., Liu, X., Ahuja, N., Han, Y., Liu, L., Liu, S., and Shen, Y. (2015). On demand cooling with real time thermal information. In *2015 31st Thermal Measurement, Modeling Management Symposium (SEMI-THERM)*, pages 138–146.

Analyzing and Estimating the Performance of Concurrent Kernels Execution on GPUs

Rommel Cruz[1], **Lucia Drummond**[1], **Esteban Clua**[1], **Cristiana Bentes**[2]

[1]Institute of Computing, Federal Fluminense University, RJ - Brazil

[2]Department of System Engineering, State University of Rio de Janeiro, RJ - Brazil

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

Abstract. *GPUs have established a new baseline for power efficiency and computing power, delivering larger bandwidth and more computing units in each new generation. Modern GPUs support the concurrent execution of kernels to maximize resource utilization, allowing other kernels to better exploit idle resources. However, the decision on the simultaneous execution of different kernels is made by the hardware, and sometimes GPUs do not allow the execution of blocks from other kernels, even with the availability of resources. In this work, we present an in-depth study on the simultaneous execution of kernels on the GPU. We present the necessary conditions for executing kernels simultaneously, we define the factors that influence competition, and describe a model that can determine performance degradation. Finally, we validate the model using synthetic and real-world kernels with different computation and memory requirements.*

1. Introduction

Graphics Processing Units (GPUs) have emerged as a cost-effective platform for high performance computing and has become ubiquitous as accelerator devices. Modern GPUs contain thousands of computing cores, very large register files, hardware thread management, and access to fast on-chip and high-bandwidth external memories. Programming models like CUDA exploit this processing power using massive thread-level parallelism, where applications are offloaded to the GPU as *kernels*.

As the GPU resources continue to increase, sharing these resources between different applications becomes imperative. However, GPUs need to be able to efficiently handle a variety of applications and provide compatible throughput to be used as shared devices. Still, GPUs are not multiprogrammed devices with an operating system as are the current CPUs. NVIDIA GPUs have hardware support for simultaneous execution of kernels. However, the hardware policy used in scheduling the kernels is proprietary, and no explicit information has been made available. Previous experiments indicate that the resource partitioning is not even among kernels, but the scheduling policy follows a *leftover* strategy [Pai et al. 2013, Aguilera et al. 2014]. The first kernel allocates all the resources needed and, then, the leftover resources are distributed to the next kernel. So, even if the kernels are independent and submitted to run concurrently, the first kernel may consume too many resources and contrain the concurrent execution with another kernel.

Although some spatial multitasking mechanism have been proposed to improve the GPU throughput [Adriaens et al. 2012, Janzén et al. 2016], they are not currently implemented in the hardware. In the current scheduling policy, the order in which kernels

are submitted for execution and their resource usage have great impact on the system throughput, occupancy rate, and GPU utilization.

This work presents an in-depth study on the simultaneous execution of kernels in the GPU. We studied the necessary conditions for real simultaneous execution, proposing an algorithm that identifies whether the hardware will actually execute two kernels simultaneously. We also analyze the effects of the concurrent execution on the performance of the two kernels and propose a model for slowdown estimation. Our concurrency algorithm and slowdown estimation model are validated with both synthetic and real-world kernels that have different computation and memory requirements.

The remainder of this paper is organized as follows. Section 2 presents previous work on concurrent execution on the GPU. Section 3 explains the concurrent execution environment of a NVIDIA GPU. Section 4 shows the algorithm proposed to determine whether two kernels will execute concurrently on the GPU. Section 5 presents the slowdown estimation model. Section 6 validates our algorithm and model with synthetic and real-world applications execution. Finally, section 7 presents our conclusions and directions for future work.

2. Related Work

In contrast to CPU multiprogramming, GPU multiprogramming is a relatively new trend, and still largely unexplored. There are only a few works that address the interference on concurrent kernel execution. Most of these works focus only on memory interference. Hu *et al.* [Hu et al. 2016] introduced a slowdown estimation model for GPUs, whose focus is on memory contention of concurrent kernels. They applied two CPU interference models known as MISE and ASM [Subramanian et al. 2015] to some GPU applications. These multicore models are based on the observation that the concurrent access of applications to memory resources is correlated to their overall slowdown compared to their sequential execution. However, this approach resulted in predictions with low accuracy. In a similar direction, and also seeking a balanced execution, Jog *et al.* [Jog et al. 2015] proposed a low-level memory scheduling mechanism that extends the hardware memory scheduler to a more fair policy relying on the bandwidth and L2 behavior of kernels. In contrast to our work, these interference studies consider an ideal case where the GPU resources are statically assigned to each kernel, while we show the behavior of the actual thread block scheduler of modern GPUs. Consequently, we perform our experiments in real hardware instead of validating our proposal using a GPU simulator.

There are also some works that focus on the CPU-GPU memory interference. Ausavarungnirun [Ausavarungnirun 2017] analyzed three types of memory interference in a CPU-GPU system, and propose an application-aware CPU-GPU memory request scheduler. Jeong *et al.* [Jeong et al. 2012] proposed a memory scheduler that guarantees the performance of GPU applications by prioritizing graphics applications over CPU applications. Our work, on the other hand, focuses on the interference of concurrent kernels execution.

On a different direction, improving GPU utilization with concurrent execution was studied in several previous works. Software techniques, such as reordering [Wende et al. 2012, Li et al. 2015, Breder et al. 2016] focus on the order in which GPU kernels are invoked on the host side. Hardware techniques, such as pre-

emption [Tanasic et al. 2014, Park et al. 2015] control the resource usage by applying preemptive multitasking on the GPU. GPU virtualization techniques [Li et al. 2011, Suzuki et al. 2014] allow multiple VMs to share the GPU resources among the cores in a heterogeneous system.

Several studies on GPU benchmark characterization [Che et al. 2010, Goswami et al. 2010, Lal et al. 2014, Ukidave et al. 2015] contribute to demonstrate that applications with irregular memory access patterns and complex control flow behaviors usually produce kernels that do not take advantage of all the GPU resources. According to Pai *et al.* [Pai et al. 2013], the Parboil2 benchmark suite uses only from 20% to 70% of the Fermi GPU resources. Adriens *et al.* [Adriaens et al. 2012] perform similar studies for 12 real-world applications, and show that most of them exhibit unbalanced GPU resource utilization.

3. Concurrent Kernel Execution

In CUDA, the parallel task submitted to run on the GPU is called a *kernel*. Each kernel can have tens of thousands of threads organized into *blocks* that are assigned to run on Streaming Multiprocessors (SMs).

The first generations of NVIDIA GPUs were not able to execute more than one kernel at a time, and the hardware resources were exploited using only thread-level parallelism. Concurrent kernel execution was introduced in the Fermi architecture, with the concept of *streams*. CUDA streams allows the programmer to express execution independence. Kernels that belong to different streams do not depend on each other and can execute concurrently. On the Fermi architecture, the hardware was responsible for multiplexing the kernels into a single work queue, where two successive kernels can execute concurrently if they were assigned to different streams. This earliest form of kernel concurrency in the GPU, however, can cause false-serialization. If two successive kernels on the queue belong to the same stream, they have to complete before additional kernels in different streams can be executed. The Kepler architecture provides the Hyper-Q technology, where 32 hardware work queues were introduced. False serialization can still occur for more than 32 streams. Recently, NVIDIA introduced the Multi-Process Service (MPS) that enables kernels from different applications to share the GPU resources. The MPS server filters the work from different processes and submit to concurrent execution.

The GPU scheduler assigns blocks to run on a specific SM, without the possibility of runtime migration. Within each SM, the threads are scheduled in the GPU scheduling unit called *warp*. The warp scheduler multiplexes the warps to execute in the CUDA cores of the SM. All the threads that belong to the same warp are executed simultaneously. A warp is considered *active* from the time its threads begin executing to the time when all threads in the warp have exited from the kernel. There is a maximum number of warps which can be active on a SM described by the compute capability of the device.

The NVIDIA scheduling policy is, however, proprietary. No published material describes the policy used for block and warp scheduling. Some previous work performed microbenchmark experiments to disclose it [Hu et al. 2016]. The speculation is that the hardware uses a *leftover* policy that assigns as many resources as possible for one kernel and then assigns the remaining resources to another kernel, if there are sufficient leftover resources. Using this policy, a resource-hungry kernel can prevent the concurrent execu-

tion of other small kernels. According to Pai *et al.* [Pai et al. 2013], around 50% of the kernels from the Parboil2 and Rodinia 2 benchmark suites consume too many resources and prevent concurrent execution of other kernels.

Another important aspect of the concurrent kernel execution is the interference that one kernel cause in the other due to simultaneous execution. Inter-kernel interference can have a negative impact in the application execution time. In the following sections, we propose a thorough analysis of concurrent execution in the GPU. We first identify whether the concurrency can occur between two independent kernels. Then, we model the slowdown caused by inter-kernel interference. The correct characterization of the concurrency capability may be an important step for maximizing GPU usage and kernels throughput.

4. Analyzing Concurrency in GPUs

Although concurrent kernel execution can be easily expressed using streams or the MPS tool, the actual simultaneous execution depends on the hardware leftover policy. We propose here an algorithm to identify whether the hardware will actually execute simultaneously two kernels submitted for concurrent execution.

Suppose two kernels k_1 and k_2 were submitted to concurrent execution in this order in a particular GPU. Basically, each attempt to run k_1 and k_2 concurrently may fall into three cases: (a) the two kernels are executed concurrently from the beginning, (b) the second kernel start its execution when the first kernel begins to release resources, (c) the two kernels are executed sequentially. Figure 1 illustrates these cases.

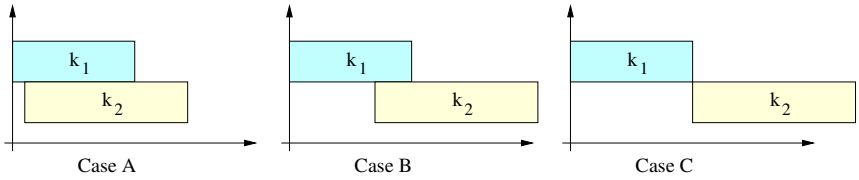


Figure 1. Concurrent execution possibilities.

We present in Algorithm 1 what we identified experimentally as the guidelines to actual concurrent execution. Suppose that the number of SMs in the GPU is nSM and that the overhead of launching a kernel is $Launch\_overhead$. Also suppose that $Blocks_{k1}$ is the total number of blocks of k_1, $ExecTime_{k1}$ is the execution time of k_1.

Since the GPU scheduler will first allocate the available resources for k_1 and, if there are leftover resources, will allocate resources for k_2, the number of blocks which can execute concurrently with k_1 on an SM is limited by: (i) the number of thread blocks available on each SM, (ii) the maximum active thread blocks imposed by the hardware, (iii) the number of thread blocks that the shared memory can accommodate given the consumption of each thread block, (iv) the number of thread blocks that the registers can accommodate given the consumption of each thread block. We define $ActiveBlocks_{k1}$ as the number of blocks from k_1 that can be active in one SM, considering these restrictions.

Initially, the algorithm tests if k_1 execution time is greater than the overhead of launching a kernel on the target device. If k_1 executes for less than the launching overhead time, the time the GPU takes to launch k_2, k_1 has finished, and no concurrency is achieved.

Algorithm 1 Concurrent scheduling

function AREEXECUTEDCONCURRENTLY(k_1, k_2, G)
  if $ExecTime_{k1} > Launch\_overhead$ **then**
    $cond1 \leftarrow Blocks_{k1} < (ActiveBlocks_{k1} \times nSM)$
    $cond2 \leftarrow GetAllocatableBlocksPerSM(k1, k2, G) > 0$
    if ($cond1 = true$) and ($cond2 = true$) **then**
      return $Case\ A$ ▷ Kernels are executed concurrently from beginning
    else if $(Blocks_{k1} \bmod (ActiveBlocks_{k1} \times nSM)) > 0$ **then**
      return $Case\ B$ ▷ Kernels are executed concurrently but not from beginning
    end if
  end if
  return $Case\ C$ ▷ Kernels are executed sequentially
end function

For k_1 and k_2 to run concurrently from the beginning (Case A), two conditions must be satisfied. First, k_1 blocks must not occupy all the SMs entirely, so the number of blocks of k_1 must be smaller than the number of blocks of k_1 that can be active in all SMs ($ActiveBlocks_{k1} \times nSM$), which means that k_1 is leaving space for another kernel execution. The second condition tests if at least one block of k_2 can be allocated in the SMs. The function $GetAllocatableBlocksPerSM$ returns the number of blocks from k_2 that can be allocated in a SM after the blocks from k_1 have been already allocated.

The function $GetAllocatableBlocksPerSM$ is shown in Algorithm 2. It examines if there is space for k_2 blocks, according to the amount of leftover resources from k_1 in terms of registers, number of threads and shared memory. First the algorithm computes the unused resources, $freeRegs$, $freeThreads$, $freeShMem$, by subtracting k_1 allocation from the hardware limits for all these resources. After that, the algorithm computes k_2 allocation on these resources dividing the amount of free resource by the k_2 request on each resource. The number of blocks allocated for k_2 is the minimum of all the possible allocations.

Algorithm 2 Calculating allocatable blocks according free resources

function GETALLOCATABLEBLOCKSPERSM(k_1, k_2, G)
  $freeRegs \leftarrow limitRegsPerSM - (regsPerBlock_{k1} \times activeBlocks_{k1})$
  $freeThreads \leftarrow limitThreadsPerSM - (threadsPerBlock_{k1} \times activeBlocks_{k1})$
  $freeShMem \leftarrow limitShMemPerSM - (shMemPerBlock_{k1} \times activeBlocks_{k1})$

  $allocBlByRegs \leftarrow freeRegs / regsPerBlock_{k2}$
  $allocBlByThreads \leftarrow freeThreads / threadsPerBlock_{k2}$

  if $shMemPerBlock_{k2} = 0$ **then**
    $allocBlByShMem \leftarrow limitBlocksPerSM$
  else
    $allocBlByShMem \leftarrow freeShMem / shMemPerBlock_{k2}$
  end if

  return $min(allocBlByRegs,\ allocBlByShMem,\ allocBlByThreads)$
end function

There is also a possibility of concurrent execution, when the number of blocks of k_1 exceeds the amount of blocks that can be active in all SMs ($(Blocks_{k1} \bmod (ActiveBlocks_{k1} \times nSM)) > 0$). In this case, k_1 must be allocated in *rounds*. A round represents an execution of part of the blocks of k_1. For example, suppose that k_1 has 128 blocks, and the GPU allows 8 active blocks to execute on one SM. For a GPU with 2 SMs, k_1 will execute in 8 rounds. The number of rounds is calculated as shown in equation (1).

$$Rounds = \left\lceil \frac{Blocks}{ActiveBlocks \times nSM} \right\rceil \tag{1}$$

We distinguish the *last round* as the round at which there maybe resources left for concurrent execution. In the last round, if $(Blocks_{k1} \bmod (ActiveBlocks_{k1} \times nSM)) > 0$, it means that k_2 can run concurrently with the remaining blocks of k_1 (Case B).

When k_1 execution time is smaller than the overhead of launching a kernel, or k_1 blocks occupy all the SMs, or the last round of k_1 does not leave space for k_2 execution, the kernels are executed sequentially (Case C).

5. Slowdown Estimation

The algorithm explained in Section 4 exposes the conditions under which two kernels actually execute concurrently in the GPU. In this section, we analyze the effects of the concurrent execution on the performance of the two kernels. We propose a slowdown estimation model that quantifies the slowdown in k_2 when it is executed concurrently with k_1 since the beginning (Case A).

According to the leftover policy, we assume that k_1 execution is not affected by the scheduling of k_2 blocks. Our slowdown estimation model considers only the performance reduction due to the lack of SM resources to the second kernel. It does not consider the interference caused by contention in memory access, which is quite difficult to quantify. Usually it requires a heavily instrumented GPU simulator and also requires a fair partition of the SMs across the concurrent applications [Jeong et al. 2012, Ausavarungnirun 2017]. GPU architectures evolve rapidly, and there is no available GPU simulator for the current architecture.

We define the estimated slowdown of k_2 as follows:

$$slowdown = \frac{RoundsLimRes}{Rounds} \tag{2}$$

$RoundsLimRes$ accounts for the number of rounds in k_2 execution, considering that there are limited resources available. The idea is to account for the allocatable blocks from k_2 according to the resources leftover by k_1. This is computed according to equation (3).

The computation of the number of rounds with limited resources has to consider two cases of the allocation of k_1 blocks on the SMs. In the first case, k_1 blocks are allocated to $nOccupied$ SMs, and leave $nFree$ SMs completely free. In the second case, k_1 blocks use all the SMs but do not fill them, leaving space for k_2 blocks. The number of k_2 blocks that can be allocated per SM in this case is $AllocBl_{k2}$, that is computed by the

function $GetAllocatableBlocksPerSM$ presented in Algorithm 2. So, the computation of $RoundsLimRes$ is performed by dividing the number of blocks of k_2 by $nFree$ multiplied by the number of blocks from k_2 that can be active in one SM, $ActiveBlocks_{k2}$, or divided by the number of SMs occupied by k_1, $nOccupied$, multiplied by the number of k_2 blocks that can be allocated per SM, $AllocBl_{k2}$.

$$RoundsLimRes = \left\lceil \frac{Blocks_{k2}}{(nFree \times ActiveBlocks_{k2}) + (nOccupied \times AllocBl_{k2})} \right\rceil \quad (3)$$

6. Experimental Results

In this section, we validate our slowdown model through the execution of pairs of concurrent kernels on the GPU. We first show the results using synthetic applications whose size and resource usage can be varied experimentally. After that, we execute pairs of real-world kernels from the Rodinia benchmark suite.

6.1. Hardware Environment

The results were obtained by direct measurements on a GPU K40. Table 1 shows the device specifications. The profiling information was obtained using the NVIDIA command-line profiler. Each experiment was repeated 30 times and we measured the average slowdown.

Table 1. GPUs configurations

	K40
Number of cores	2,880
RAM	12GB
Memory Bandwidth	288 GB/s
Capability	3.5
Number of SMs	15
Shared Memory per SM	48KB
Number of Registers per SM	64K
Max number of threads per SM	2048
Max thread blocks per SM	16
Max registers per thread	255
Maximum thread block size	1024
Architecture	Kepler

6.2. Framework for Concurrent Execution

In most of the GPU applications, before launching a kernel, the application has to transfer data from the CPU to the GPU and after the kernel finishes, the data has to be copied back from the GPU to the CPU. So, if two applications are submitted to execute at the same time, there maybe no actual concurrency among their kernels depending on the time taken for memory transfers. Since we are interested in the actual kernel concurrency, we implemented a framework that isolates the execution of the kernels.

Our framework guarantees that the kernels are submitted to execute at the same time, generating potential concurrency. A set of kernels are placed in a waiting queue

until all of their associated initialization and memory transfers were performed. After that, the framework launches the kernels on different CUDA streams. The framework inserts a synchronization barriers before and after launching the kernels.

The framework was implemented in C++ and built with g++ version 4.8.4 together with the host codes of the benchmark applications. The GPU kernels were compiled with NVCC CUDA version 7.5.

6.3. Synthetic Applications

In order to evaluate scenarios where there is no memory contention between the kernels, we created a set of synthetic kernels. These kernels evaluate the slowdown when different resource requirements are established.

Each synthetic kernel k_i performs a set of arithmetic operations on register values. The number of blocks, number of threads and shared memory requirements are created randomly.

In this experiment, two sets of 50 kernels were created. In the first set, we select k_1 kernels and in the second set, we select k_2 kernels. The kernels in the first set were created to allow concurrency from the beginning (Case A), so their number of blocks satisfies $Blocks_{k1} < (ActiveBlocks_{k1} \times nSM)$ (Algorithm 1). A huge number of experiments were set, but we show here only a random sample of these experiments. Table 2 shows 12 kernels, numbered from 1 to 12, where the odd-numbered kernels were derived from the first set and the even-numbered were derived from the second set.

Table 2. Synthetic applications characteristics

Kernel	# Blocks	# Threads	Sh Memory (B)
S1	110	256	1024
S2	450	256	0
S3	100	256	4096
S4	60	256	0
S5	42	512	256
S6	120	128	0
S7	90	256	896
S8	467	512	256
S9	35	256	2048
S10	130	512	1024
S11	35	256	0
S12	230	256	0

6.3.1. Results

Table 3 shows the results of the estimated and the actual slowdown for a set of pairs from the 12 first kernels. The estimated slowdown is computed by equation (2). The actual slowdown is the ratio between the execution time when the kernel is executed concurrently and the execution time when the kernel is executed alone. The percentage relative error is computed by equation (4).

$$error = \frac{estimated - actual}{actual} \times 100\% \quad (4)$$

Table 3. Estimated vs Actual slowdown (synthetic kernels)

Kernel Pair $k1 - k2$	Estimated	Actual	Perc. Error
S1-S2	11.25	11.312	0.55%
S3-S4	3.00	3.018	0.61%
S5-S6	2.00	1.998	0.12%
S7-S8	4.00	3.937	1.59%
S9-S10	1.30	1.333	2.49%
S11-S12	1.50	1.494	0.39%

We can observe in Table 3 that the combinations of S1-S2 and S7-S8 provided the highest slowdowns. This occurs because, in these cases, k_2 has a great number of blocks, but k_1 left almost no space for their execution. In the K40 GPU, the maximum number of active blocks per SM is 16.

Comparing the estimated and the actual slowdown obtained, we can observe that the percentage relative error is small, at most 2.49%. For the whole experiment, with the 100 kernels, we obtained an average of 3.49% of error and a standard deviation of 6.43%.

6.4. Real-World Applications

In order to evaluate real-world scenarios, we used 8 applications from the Rodinia benchmark suite: k-Nearest Neighbors (kNN), Path Finder (PF), Hotspot 3D (HS3), Breadth-First Search (BFS), Hotspot 2D (HS2), Speckle Reducing Anisotropic Diffusion version 2 (SRAD), LU Decomposition (LUD), and Particle Filter (PFL). Table 4 summarizes these applications resource requirements. We used the most relevant kernel (in terms of percentage of execution time) for each application in the experiments.

Table 4. Rodinia applications characteristics

App	**Kernel**	**#Registers**	**# Blocks**	**# Threads**	**Sh Memory (B)**
kNN	euclid	8	3840	256	0
PF	dynproc_kernel	13	463	256	2048
HS3	hotspotOpt1	36	1024	256	0
BFS	Kernel	19	1954	512	0
HS2	calculate_temp	38	1849	256	3072
SRAD	srad_cuda_2	21	16384	256	5120
LUD	lud_diagonal	32	1	16	1024
PFL	KernelFindIndex	13	47	128	0

6.4.1. Results

Among all possible pair combinations of these applications, we present the results of the pairs in which k_1 is the PFL application. PFL allows concurrency from the beginning (Case A), since $Blocks_{PFL} < (ActiveBlocks_{PFL} \times nSM)$.

Table 5 shows the comparison between the estimated and the actual slowdown for the combinations of PFL with all the other applications. We observed an average error of 10.6%. The highest error was produced by the combination PFL-HS3. In this case, our model estimated a smaller slowdown than the real achieved value. HS3 is the application with the highest percentage of usage of the memory bandwidth, around 60%. The experiments with the kernel HS3 suggest that the slowdown was increased by the memory contention. For the other applications, we can observe that the main source of performance reduction in concurrent execution is the amount of resources leftover by PFL kernel.

Table 5. Estimated vs Actual slowdown (Rodinia kernels)

Kernel Pair	Estimated	Actual	Perc. Error
PFL-kNN	1.250	1.185	5.46%
PFL-PF	1.250	1.252	0.17%
PFL-HS3	1.290	2.409	46.46%
PFL-BFS	1.240	1.357	8.60%
PFL-HS2	1.240	1.260	1.55%
PFL-SRAD	1.250	1.117	11.94%
PFL-LUD	1.000	1.004	0.36%

7. Conclusions

This work presented an in-depth study on the concurrent kernel execution in the GPU. Modern GPU architectures support concurrent sharing of the GPU resources among multiple kernels, which can unleash the power of the GPU for dynamic and highly virtualized environments. However, the GPU does not have an operating system and the hardware implements a *leftover* policy that assigns as many resources as possible for one kernel and then assigns the remaining resources to another kernel. Under this policy, a resource-hungry kernel can prevent the concurrent execution of other small kernels. Based on this, we studied the necessary conditions for simultaneous execution, and proposed an algorithm that describes when actual concurrency can occur. We also proposed a model for slowdown estimation. Our algorithm and slowdown model do not require simulation instrumentation. They use only the resource requirement data of the kernels.

We validated our slowdown model with synthetic and real-world applications. Our model was able to predict the slowdown in different resource requirements scenarios. For the synthetic applications, that do not present memory interference, our model was able to predict the slowdown with an average of 3.49% of error. For real-world applications, the average error was higher, around 10.6%. The application with the greatest memory bandwidth requirements was the one that provided the greatest error. Our results show that the GPU resources can be shared among the kernels, but the hardware does not provide a fair scheduling policy. In this sense, it is important to identify the kernel characteristics to co-schedule applications with complementary resource requirements. Further studies are necessary to evaluate the impact of memory interference in our model.

For future work, we intend to investigate memory contention in real-world applications and include in our model this study. We also intend to study the effects of distinct GPU architectures in our model.

References

Adriaens, J. T., Compton, K., Kim, N. S., and Schulte, M. J. (2012). The case for GPGPU spatial multitasking. In *2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA),*, pages 1–12.

Aguilera, P., Morrow, K., and Kim, N. S. (2014). Fair share: Allocation of GPU resources for both performance and fairness. In *32nd IEEE International Conference on Computer Design (ICCD), 2014*, pages 440–447.

Ausavarungnirun, R. (2017). *Techniques for Shared Resource Management in Systems with Throughput Processors*. PhD thesis, Carnegie Mellon University.

Breder, B., Charles, E., Cruz, R., Clua, E., Bentes, C., and Drummond, L. (2016). Maximizando o uso dos recursos de GPU através da reordenação da submissão de kernels concorrentes. In *Anais do WSCAD 2016 Simpósio de Sistemas Computacionais de Alto Desempenho*, pages 98–109. Editora da Sociedade Brasileira de Computação (SBC).

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. In *IEEE International Symposium on Workload Characterization (IISWC), 2010*, pages 1–11.

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

Hu, Q., Shu, J., Fan, J., and Lu, Y. (2016). Run-time performance estimation and fairness-oriented scheduling policy for concurrent GPGPU applications. In *45th International Conference on Parallel Processing (ICPP), 2016*, pages 57–66.

Janzén, J., Black-Schaffer, D., and Hugo, A. (2016). Partitioning GPUs for improved scalability. In *28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2016*, pages 42–49.

Jeong, M. K., Erez, M., Sudanthi, C., and Paver, N. (2012). A qos-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an mpsoc. In *49th Annual Design Automation Conference*, pages 850–855.

Jog, A., Kayiran, O., Kesten, T., Pattnaik, A., Bolotin, E., Chatterjee, N., Keckler, S. W., Kandemir, M. T., and Das, C. R. (2015). Anatomy of GPU memory system for multi-application execution. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 223–234.

Lal, S., Lucas, J., Andersch, M., Alvarez-Mesa, M., Elhossini, A., and Juurlink, B. (2014). GPGPU workload characteristics and performance analysis. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014*, pages 115–124.

Li, T., Narayana, V. K., El-Araby, E., and El-Ghazawi, T. (2011). GPU resource sharing and virtualization on high performance computing systems. In *International Conference on Parallel Processing (ICPP), 2011*, pages 733–742.

Li, T., Narayana, V. K., and El-Ghazawi, T. (2015). A power-aware symbiotic scheduling algorithm for concurrent GPU kernels. In *IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), 2015*, pages 562–569.

Pai, S., Thazhuthaveetil, M. J., and Govindarajan, R. (2013). Improving GPGPU concurrency with elastic kernels. In *ACM SIGPLAN Notices*, volume 48, pages 407–418.

Park, J. J. K., Park, Y., and Mahlke, S. (2015). Chimera: Collaborative preemption for multitasking on a shared GPU. *ACM SIGARCH Computer Architecture News*, 43(1):593–606.

Subramanian, L., Seshadri, V., Ghosh, A., Khan, S., and Mutlu, O. (2015). The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *48th International Symposium on Microarchitecture*, pages 62–75.

Suzuki, Y., Kato, S., Yamada, H., and Kono, K. (2014). Gpuvm: Why not virtualizing GPUs at the hypervisor? In *USENIX Annual Technical Conference*, pages 109–120.

Tanasic, I., Gelado, I., Cabezas, J., Ramirez, A., Navarro, N., and Valero, M. (2014). Enabling preemptive multiprogramming on GPUs. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 193–204.

Ukidave, Y., Paravecino, F. N., Yu, L., Kalra, C., Momeni, A., Chen, Z., Materise, N., Daley, B., Mistry, P., and Kaeli, D. (2015). Nupar: A benchmark suite for modern GPU architectures. In *6th ACM/SPEC International Conference on Performance Engineering*, pages 253–264.

Wende, F., Cordes, F., and Steinke, T. (2012). On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC), 2012*, pages 74–83.

Analyzing the I/O Performance of Post-Hoc Visualization of Huge Simulation Datasets on the K Computer

Eduardo C. Inacio[1,2], Jorji Nonaka[2], Kenji Ono[2,3], Mario A. R. Dantas[1,2]

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

[2]RIKEN AICS Advanced Visualization Research Team
Kobe, Hyogo – Japan

[3]Kyushu University
Fukuoka, Fukuoka – Japan

eduardo.camilo@posgrad.ufsc.br, jorji@riken.jp
keno@cc.kyushu-u.ac.jp, mario.dantas@ufsc.br

Abstract. *As computational science simulations produce ever increasing volumes of data, executing part or even the entire visualization pipeline in the supercomputer side becomes more a requirement than an option. Given the uniqueness of the high performance K computer architecture, the* HIVE *visualization framework was developed, focusing on meeting visualization and data analysis demands of scientists and engineers. In this paper, we present an analysis on the input/output (I/O) performance of post-hoc visualization. The contribution of this research work is characterized by an analysis of a set of empirical study cases considering huge simulation datasets using HIVE on the K computer. Results from the experimental effort, using a dataset produced by a real-world global climate simulation, provide a differentiated knowledge on the impact of dataset partitioning parameters in the I/O performance of large-scale visualization systems, and highlight challenges and opportunities for performance optimizations.*

1. Introduction

As the scale of computational science simulations grows to deal with increasingly complex problems, we can also verify a significant increase in the volume of data produced [Roten et al. 2016]. To derive meaningful information from these huge datasets, leading to scientific discoveries and breakthroughs, scientists and engineers rely upon large-scale visualization and data analysis systems [Nonaka et al. 2014]. Such data-intensive applications pose a great pressure on the shared backend storage system of modern high performance computing (HPC) environments. As a result, file I/O becomes a considerable bottleneck.

In order to mitigate performance degradation due to this extreme data movement, approaches have been proposed to execute part or even the entire visualization pipeline in the supercomputer side [Bennett et al. 2012, Dorier et al. 2016]. This scenario is specially verified in HPC environments employing data staging approaches, such as the K computer [Miyazaki et al. 2012]. This approach consists of moving applications' input data to a high throughput file system prior to job execution (*i.e.,* stage-in),

and moving generated output data to users' file system after job completion (*i.e.,* stage-out) [Tsujita et al. 2017].

Focusing on meeting large-scale visualization needs on the K computer environment, a visualization framework, named Heterogeneously Integrated Visual-analytics Environment (HIVE), has been developed [Nonaka et al. 2016]. The HIVE visualization framework offers a scalable approach for both post and in-situ visualization at heterogeneous computing environments, taking advantage of increasing parallelism of modern supercomputers as well as harnessing hardware acceleration capabilities when they are available. An example of the HIVE capabilities is illustrated in Figure 1, in which is presented a projection of a 16K resolution image produced by the HIVE framework on the K computer using a dataset of 1.1 TiB generated by a global climate simulation.

Figure 1. Projection of a 16K resolution image produced by the HIVE framework on the K computer. Data courtesy of JAMSTEC, AORI/The University of Tokyo (HPCI SPIRE3), and RIKEN AICS Computational Climate Science Research Team.

Although results achieved using the HIVE visualization framework have been promising, we have verified file I/O plays a significant role in the execution time. More specifically, when used for post-processing (*i.e.,* post-hoc visualization), normally, a smaller number of compute nodes is allocated for the visualization processing, compared to the number of compute nodes used for simulation, because of the difference in processing power demands of both applications. Considering that a common approach among simulation systems is to output data into multiple files, in a per process, per variable, and/or per time step basis, this difference in scale arises as a particular problem. Mostly because the original simulation output dataset needs to be repartitioned in order to balance the workload among visualization processes [Ono et al. 2014].

In this paper, we report results of an experimental analysis on the impact of dataset partitioning parameters on the I/O performance of the HIVE visualization framework executing on the K computer. Such experimental efforts have previously demonstrated

interesting behavior on parallel storage systems [Inacio et al. 2015, Inacio et al. 2017]. This study is part of an ongoing research work collaboration that focuses on optimizing parallel I/O and storage related parameters for large-scale visualization systems running on supercomputers. While the initial focus has been the execution of the HIVE framework on the K computer, results from this research work are expected to provide helpful insights for the enhancement of the data management features of the HIVE framework on next-generation systems, including the post-K supercomputer.

The remainder of this paper is organized as follows. Section 2 provides a brief overview of the HIVE visualization framework on the K computer environment. In Section 3, the dataset partitioning problem is described. The experimental methodology and environment employed in this study is detailed in Section 4, while experimental results are discussed in Section 5. Section 6 concludes this paper with a summary of the observed results and our future research directions.

2. HIVE – Post-hoc Visualization on the K Computer

The HIVE visualization framework [Nonaka et al. 2016] was designed to run on heterogeneous hardware platform environments found on traditional HPC infrastructures, such as the K computer operational environment. The development of HIVE was mainly motivated by the demand of visualizing huge datasets generated by large-scale computational science simulations executed on the K computer. Another reason, relies upon the unique architecture of the K computer, that makes it difficult to directly adopt existing visualization systems, such as PARAVIEW [Fabian et al. 2011] and VISIT [Childs et al. 2012].

The design of HIVE follows a client/server paradigm. This approach offers a great flexibility for users to select configurations that better suit their visualization needs. For instance, a client can be executed on a local machine, exploring interactivity, while a server could be deployed at a visualization cluster, fully utilizing abundant hardware resources. A web-based graphic user interface (GUI) permits easily definition of visualization pipelines, which are later exported as Lua scripts, allowing for the automation of the visualization workflow. Moreover, a command-line interface allows for batch execution of visualization workflows, which is particularly helpful for post-hoc visualization in the K computer.

An overview of a post-hoc visualization using the HIVE framework on the K computer environment is illustrated in Figure 2. The shared backend storage of the K computer has two layers: a Local File System (LFS), designed to support high throughput I/O; and a Global File System (GFS), that offers a large storage capacity (> 30 PB) for users' applications and data files; both implemented through the Fujitsu Exabyte File System (FEFS), an enhanced version of the LUSTRE parallel file system (PFS). Since only executing jobs have access to the LFS, simulation output data is available to users after the job completion, when this data is staged-out to the GFS.

In order to execute a post-hoc visualization on the K computer using HIVE, the user must provide a *visualization scene*, which contains the parameters for the visualization processing. This visualization scene can be produced manually, or using one of the editors provided with the HIVE framework. With the simulation output data available at the GFS, the user can submit a job script invoking HIVE rendering command (`hrender`), passing the visualization scene as argument. It is worth noting that, given

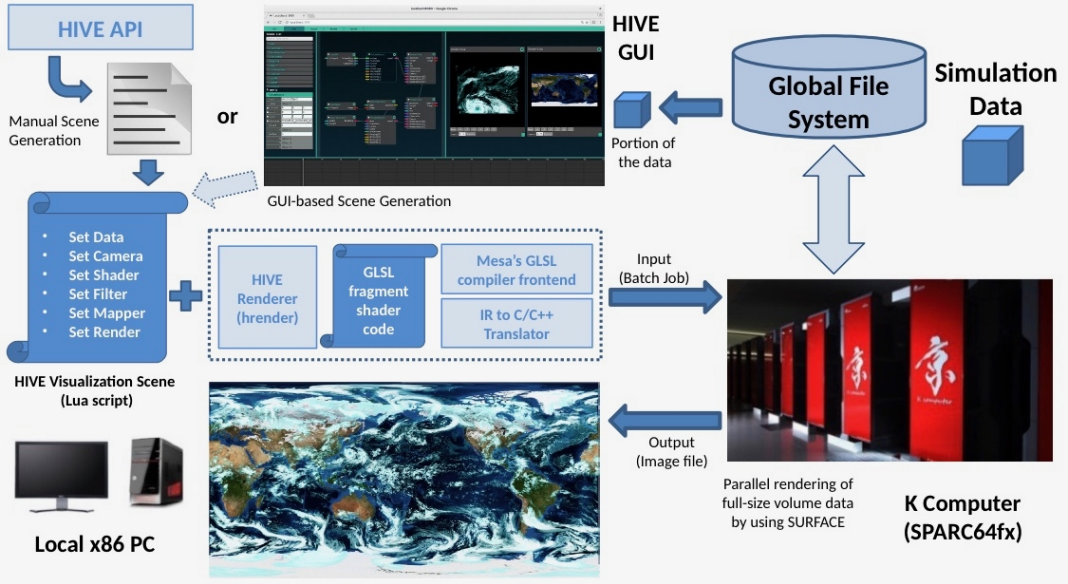


Figure 2. An overview of a post-hoc visualization using the HIVE framework on the K computer environment.

the staging approach of the K computer, simulation output data is staged-in again before the visualization job is launched for execution, which translates into a massive data movement. Once the visualization job is finished, image files stored in the LFS are staged-out, and can be analyzed by users.

3. The *M* x *N* Dataset Partitioning

The number of processes required by a visualization job for rendering a simulation output dataset is usually smaller than the number of processes used for generating the dataset. Also, computational science simulations executed on K computer usually adopt a file per processes approach when outputting data, mostly motivated by particular optimizations provide by the FEFS for such access pattern [Tsujita et al. 2017]. Consequently, in post-hoc visualization, the problem domain must be repartitioned in order to achieve load balance among visualization processes. As a result, a single visualization process can access data points in multiple files, and a single file can be concurrently accessed by multiple visualization processes.

The *M* x *N* dataset partitioning problem refers to these situations, in which a simulation generates *M* output files, and *N* visualization processes, with *N* usually smaller than *M*, will consume the dataset. The dataset partitioning options, which include the number of visualization processes and the number of partitions in the x, y, and z dimensions (for a Cartesian three dimensional problem space), are provided by the user through the visualization scene script. As previously stated, the way the output dataset is partitioned can have significant impact in the I/O performance of the post-hoc visualization, mainly due to conditions of concurrent access to shared files. Nevertheless, identifying the optimal options for the partitioning parameters can be a daunting task, considering the innumerous options available.

More specifically, the alignment of the visualization dataset partitioning with the

original partitioning of the simulation output could avoid concurrent file accesses conditions. Figure 3 provides an example of an aligned dataset partitioning. The simulation dataset consists of 12 files, partitioning the two dimensional problem domain into 4 x 3 grids. At post-hoc visualization, four processes were used, partitioning the simulation dataset into 4 x 1 grids. Under these partitioning parameters, it is possible to observe that each visualization process would independently access tree simulation output files.

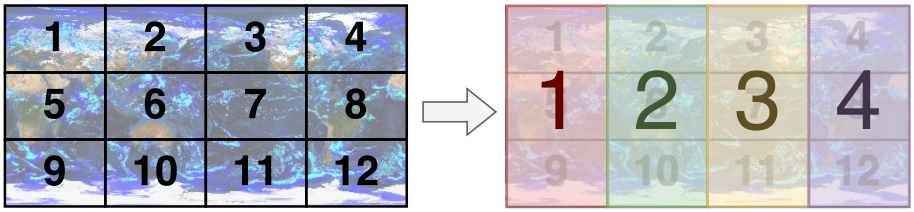


Figure 3. Example of a visualization dataset partitioning aligned with the partitioning used at simulation output files (M = 12, N = 4).

A very different situation is observed when the visualization partition parameters are defined as 2 x 2, even keeping the same number of visualization processes, as demonstrated in Figure 4. This configuration results in an aligned partition of the simulation output. Consequently, beyond visualization process accessing multiple files, it can be verified that some files are concurrently accessed by two processes. In this example, files 5 and 6 are concurrently accessed by visualization processes 1 and 3, while files 7 and 8 are accessed by visualization processes 2 and 4.

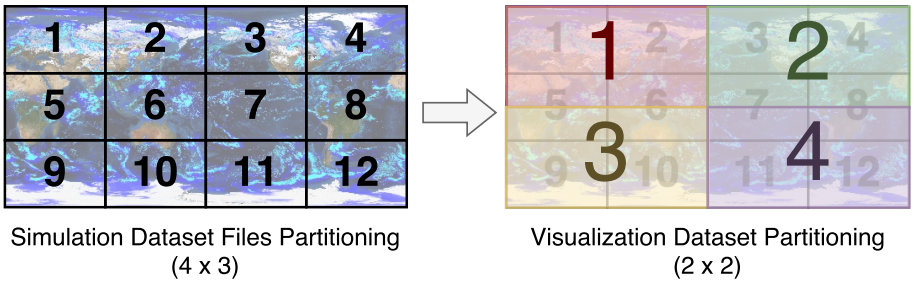


Figure 4. Example of a visualization dataset partitioning unaligned with the partitioning used at simulation output files (M = 12, N = 4).

Based upon these simple examples provided, it is possible to conceive that at larger scales, with hundreds to thousands of processes and files involved, the complexity of the *M* x *N* partitioning problem can increase drastically. Consequently, a significant impact in the overall performance of the post-hoc visualization on the K computer can be attributed to I/O performance degradation associated to dataset partitioning parameters. Next sections details the experimental effort carried out focusing on better understanding the magnitude of such impact.

4. Experimental Environment and Method

In order to observe the impact of using different partitioning parameters into the post-hoc visualization of a huge simulation dataset, several experiments were conducted processing a real-world simulation dataset using the HIVE visualization framework on the K computer. The K computer [Miyazaki et al. 2012] is a Japanese flagship-class supercomputer, developed by Fujitsu in collaboration with RIKEN and currently operated by the RIKEN Advanced Institute for Computational Science (AICS), consisting of $82{,}944$ compute nodes, with a SPARC64fx CPU and 16 GB RAM each, connected through a 6D Tofu interconnect [Ajima et al. 2011]. At full capacity, the K computer is capable of performing 10 Petaflops (10 quadrillion floating-point operations per second), which granted it the top position at the list of the 500 fastest supercomputers in the world for two consecutive times in 2011 [Dongarra et al. 2017], when it started operation.

A dataset generated by a real-world simulation was used in these experiments. This dataset corresponds to a sub-kilometer global simulation of deep moist atmospheric convection using the Nonhydrostatic Icosahedral Atmospheric Model (NICAM) [Miyamoto et al. 2013] executed on the full configuration of the K computer. The NICAM simulation outputs variables in a file per process for each time step. After simulation, variables in the dataset are remapped from the icosahedral grid to a geodesic (latitude-longitude) grid [Satoh et al. 2017]. In this simulation dataset, the x, y, and z dimensions have respectively $11{,}520$, $5{,}760$, and 94 points, and 48 time steps. For this analysis, a single variable and four time steps were considered, resulting in a dataset of 94 GiB. It is worth noting that time steps are processed by the visualization system sequentially. Therefore, using 48 times steps would mainly result in 12-fold larger execution times.

Furthermore, simulation data points were redistributed into 384 files per time step, as if they were generated by a simulation using 384 processes. As a result, files became larger in this configuration than in the original one, keeping the dataset size fixed. This approach was adopted in order to have a baseline performance (*i.e.,* an *M x M* mapping) for comparison of different *M x N* configurations. Executing an *M x M* post-hoc visualization using the full configuration of the K computer (*i.e.,* $82{,}944$) would not only be an unrealistic scenario, but also undesirable from an operational perspective.

Each process (MPI rank) is allocated to a different compute node. Before execution, the dataset is staged-in to the K computer LFS. It is worth mentioning that the K computer resource management system (RMS) allocates Object Storage Servers (OSSs) for a job accordingly to the number and shape of allocated compute nodes. Basically, OSSs in the same racks of allocated compute nodes are made available for jobs. This policy focuses making data closer to processes and mitigating cross-application interferences.

Table 1 presents dataset partitioning parameters considered in the experiments with post-hoc visualization of the NICAM simulation output dataset using the HIVE framework on the K computer. It can be observed that a wide range of parameters were evaluated. Depending on the number of processes used for visualization, which ranges from the same number of files in the dataset up to 16 times less processes, varying aligned and unaligned partitions were considered. The grid size refers to the number of points in each dimension (*i.e.,* x x y x z) per process and per time step.

Table 1. Dataset partitioning parameters considered in the experimental evaluation using the HIVE visualization framework in the K computer.

# Processes	# Partitions	Grid Size (Pts)	File Size	Alignment
384	32 x 12 x 1	360 x 480 x 94	62 MiB	Aligned
192	32 x 6 x 1	360 x 960 x 94	124 MiB	Aligned
	6 x 32 x 1	1920 x 180 x 94	124 MiB	Unaligned
	16 x 12 x 1	720 x 480 x 94	124 MiB	Aligned
	12 x 16 x 1	960 x 360 x 94	124 MiB	Unaligned
96	16 x 6 x 1	720 x 960 x 94	248 MiB	Aligned
	6 x 16 x 1	1920 x 360 x 94	248 MiB	Unaligned
	8 x 12 x 1	1440 x 480 x 94	248 MiB	Aligned
	12 x 8 x 1	960 x 720 x 94	248 MiB	Unaligned
48	4 x 12 x 1	2880 x 480 x 94	496 MiB	Aligned
	12 x 4 x 1	960 x 1440 x 94	496 MiB	Unaligned
	8 x 6 x 1	1440 x 960 x 94	496 MiB	Aligned
	6 x 8 x 1	1920 x 720 x 94	496 MiB	Unaligned
24	8 x 3 x 1	1440 x 1920 x 94	992 MiB	Aligned
	3 x 8 x 1	3840 x 720 x 94	992 MiB	Unaligned
	4 x 6 x 1	2880 x 960 x 94	992 MiB	Aligned
	6 x 4 x 1	1920 x 1440 x 94	992 MiB	Unaligned

In the context of this research work, an experiment run consists of the dataset loading phase of the HIVE visualization framework, using one of the partitioning parameters presented in Table 1. I/O performance metrics (*i.e.,* response variables) evaluated in this research work are the time each process takes to load the required data for each time step, and the time required to load the complete dataset, including all time steps. Each experiment run was replicated three times to account for experimental variance. The execution order of the experiment runs was completely random to assure response variables are independent and individually distributed.

5. I/O Performance Analysis

As discussed in Section 3, depending on the partitioning parameters, the visualization processes may need to read from multiple files. Furthermore, multiple processes may simultaneously access the same file, which can result in a contention translated into an I/O performance degradation. To verify such behavior, an analysis was carried out on the time that each process takes to load its designated data points per time step.

Figure 5 presents the load time per process according to different dataset partitioning parameters. Boxes delimits the first and third quantiles (*i.e.,* the 25th and 75th percentiles), thus, comprising 50% of the measurements. The horizontal line inside boxes denotes the median value (*i.e.,* 50th percentile), while vertical lines, known as whiskers, extend to largest and smallest values no further than $1.5\times$ the inter-quartile range (IQR) above or below boxes, respectively. Data points beyond whiskers denote outliers. Red boxes denote unaligned partitions, while blue boxes denote aligned partitions.

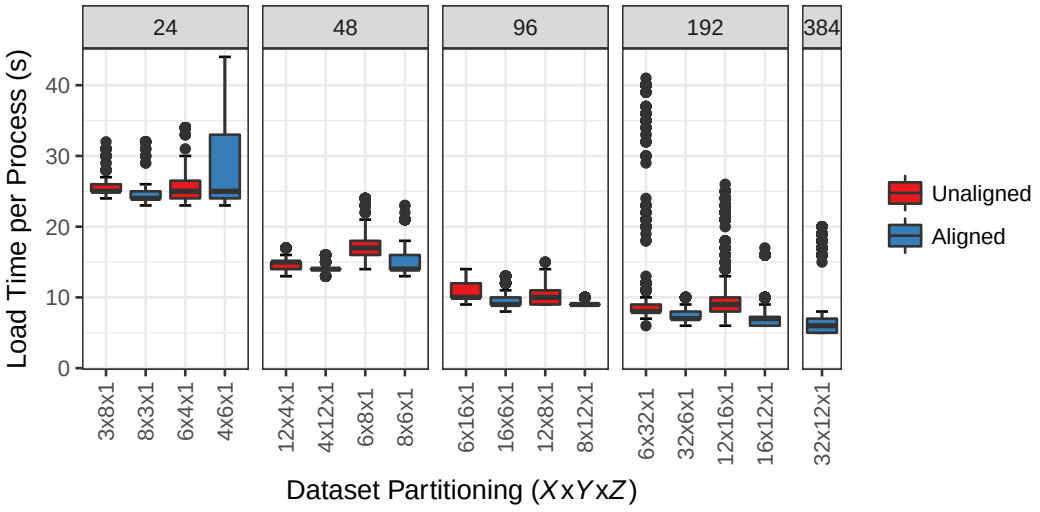


Figure 5. Load time of a single time step of the simulation dataset per process using different number of processes and partitions.

The most evident behavior in these results is the decrease of the load time per process with the increase in the number of processes used for visualization. This behavior can be mostly associated with the amount of data read by each visualization process. Considering the dataset size is fixed for the experiment, increasing the number of processes translates into each process loading a smaller number of points, and, thereby, reading a smaller fraction of the dataset. However, it is worth noting that the load time did not reduce in the same rate of the increase of the number of processes. Comparing the mean load time using 24 processes with the baseline case, with 384 processes, the I/O performance was approximately 3.8 times better at the cost of 16 times more compute nodes, an efficiency of 23.5%. For 96 processes, the compared efficiency increases to 35.4%, which can still be considered low. These results shed light upon this important trade-off faced by this large-scale visualization systems executing on the K computer.

Another prominent behavior observed in these results regards the variance of the load time per process across varying partitioning parameters. In general, it is possible to observe a larger variance for unaligned partitions compared to aligned ones; not only by the length of the boxes, but also by the number of outliers observed. Moreover, based upon the shape of the boxes and predominance of upper outliers, it is possible to conceive that the response variable has a skewed distribution with a long tail to the right. This means that most of the measurements are concentrated around smaller values, but with a number of measurements way above the average. Although many factors can contribute to this delay observed in some processes while loading their respective data points from the dataset, it is reasonable to infer that contention due to multiple processes accessing the same file has a non negligible impact in the I/O performance. Precisely controlling the computing environment for isolating such effect in an experiment is challenging, because of the number and variety of elements involved, including other concurrent jobs running at the supercomputer. A further investigation is in progress focusing on addressing this matter.

While the analysis of the time for a process to complete loading data points from a time step of the simulation datasets gives insight about the contention effects resulting from partition parameters, it is also important, from the visualization system perspective, to verify the overall performance behavior, since the delay of some parallel processes may overlap and have their direct impact in performance less perceived. In Figure 6, the time to load the complete dataset considering different partitioning parameters is presented. In other words, this response variable represents the time elapsed for all processes to read their respective data points from all time steps of the dataset in the LFS of the K computer. Red bars denote the mean value for unaligned partitions, while blue bars denote the mean value for aligned ones. Vertical error bars represent the standard error around the mean.

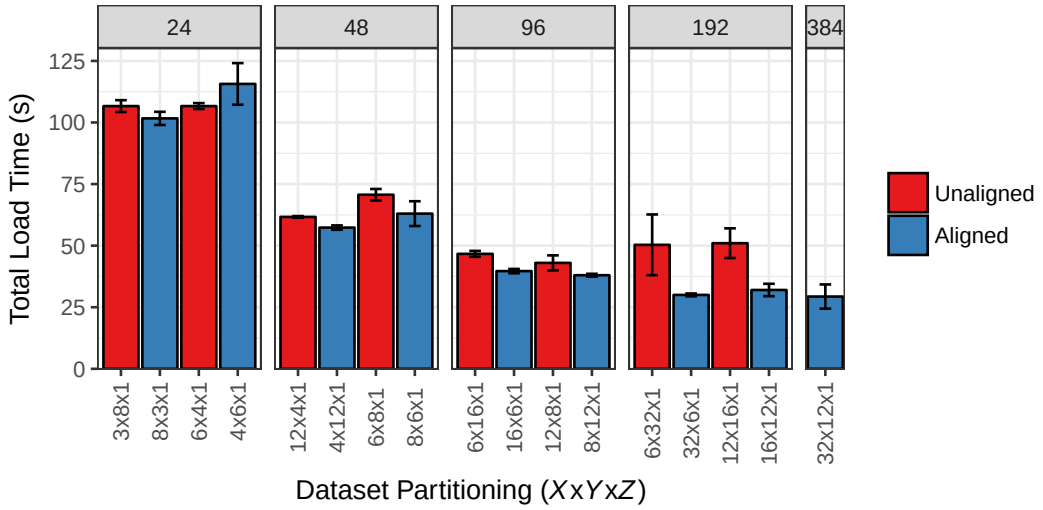


Figure 6. Time to load the complete dataset using varying number of processes and partitions.

Through these results, the comparison of the load time among different number of processes becomes more clear. It is noticeable that experiments with 24 processes had a total load time significantly higher compared to the other experiments. However, it can be observed that for 48, 96, 192, and the baseline case, 384 processes, the mean values became closer for some partitions. For the unaligned partitions 12 x 4 x 1 (48 processes), 6 x 16 x 1, 12 x 8 x 1 (96 processes), 6 x 32 x 1, and 12 x 16 x 1 (192 processes), both mean values and standard error values overlap. This indicates that it is not possible to discern the I/O performance among these partition parameters. In other words, using a partition 12 x 4 x 1 with 48 processes may outcome in the same total load time as using a 6 x 32 x 1 partition with 192 processes. The same interpretation is valid for the aligned partitions 32 x 6 x 1, 16 x 12 x 1 (192 processes), and 32 x 12 x 1 (384 processes). This means that, using half of the number of processes used in the simulation for HIVE post-hoc visualization in the K computer, with proper aligned partitioning parameters, a comparable I/O performance can be achieved, while saving computing resources.

Comparing aligned and unaligned partitions inside a fixed number of processes, it is possible to verify how the partitioning alignment can affect the total load time in the post-hoc visualization in the K computer. Except for one case, namely, the partition 4

x 6 x 1 (24 processes), unaligned partitioning resulted in a larger load time. The scale of the differences varies significantly, though. Results suggest the difference is affected by the number of concurrent processes in a file, which is a result from the partitioning parameters. For instance, in the 6 x 32 x 1 partition (192 processes), each file in the original dataset is accessed by at least three visualization processes, while some files are accessed by six processes. Based upon these results, it is possible to conceive the importance of properly choosing the partitioning parameters for post-hoc visualization combined with a more efficient use of the computing resources of the K computer.

6. Conclusions and Future Works

In this research work, we present a contribution that could be understood as the analysis of experiments carried out using the HIVE framework to visualize a huge dataset produced by a real-world computational science simulation on the K computer, and how significant was the impact of dataset partitioning in the I/O performance of a large-scale visualization system. The research presented in this paper will contribute to advance a collaborative research work in progress that aims at optimizing parameters available at the parallel I/O software stack of modern supercomputers for large-scale visualization purposes. Moreover, these results can provide insights on visualization requirements for the design of next-generation supercomputers, including the post-K computer.

Analyzing the time taken by each process to load its data points from the simulation output dataset, it was observed a strong influence of the volume of data transferred in the I/O performance. However, while increasing the number of process effectively reduced the load time per process, results indicate the efficiency of this approach can be low, pointing out an important trade-off between load time and computation resource utilization. On extreme cases, the load time was reduced by 3.8 times using 16 times more processes. Such observation provides compelling reasons for further research in the subject in order to more efficiently explore the available computing resources Furthermore, the variance of the load time across processes suggests while most processes complete loading a time step of the simulation output within a particular time window, a number of processes get significantly delayed. Based on the conditions of the experiment, we argue that such behavior can be associated, in part, to contention in file I/O access due to partitioning parameters.

Considering the time taken by HIVE to load all the simulation dataset, it was observed that the difference in I/O performance becomes less prominent. It is reasonable to consider that differences in load time per tasks get overlapped by parallel processing. Such observation gives insight on real opportunities for executing post-hoc visualization on K computer using a smaller number of processes than what was used for simulation execution, with comparable I/O performance. Nevertheless, dataset partitioning parameters can play a significant role in these situations, resulting in performance degradation if not properly addressed.

As part of this ongoing research work, we intend to, in short-term, perform a more detailed analysis, instrumenting the HIVE code to obtain fine-grained measurements of I/O operations performed during dataset loading. Furthermore, experiments should be conducted to assess the impact of factors related to the parallel I/O software stack of the K computer, such as file striping options of the FEFS, which could also be leveraged for

improved visualization performance. After this characterization effort, an optimization approach for the dataset management of the HIVE visualization system will be designed and developed, focusing on providing efficient large-scale visualization for extreme-scale supercomputers.

Acknowledgements

We would like to thank the Brazilian Federal Agency CAPES for supporting this research. Results reported in this paper were obtained by using the K computer at RIKEN Advanced Institute for Computational Science (AICS) in Kobe, Japan.

References

Ajima, Y., Takagi, Y., Inoue, T., Hiramoto, S., and Shimizu, T. (2011). The Tofu Interconnect. In *HOTI '11 Proceedings of the IEEE 19th Annual Symposium on High Performance Interconnects*, pages 87–94. IEEE.

Bennett, J. C., Abbasi, H., Bremer, P.-T., Grout, R., Gyulassy, A., Jin, T., Klasky, S., Kolla, H., Parashar, M., Pascucci, V., Pebay, P., Thompson, D., Yu, H., Zhang, F., and Chen, J. (2012). Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *SC '12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE.

Childs, H., Brugger, E., Whitlock, B., Meredith, J., Ahern, S., Pugmire, D., Biagas, K., Miller, M., Harrison, C., Weber, G. H., Krishnan, H., Fogal, T., Sanderson, A., Garth, C., Bethel, E. W., Camp, D., Rübel, O., Durant, M., Favre, J. M., and Navrátil, P. (2012). VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In Bethel, E. W., Childs, H., and Hansen, C., editors, *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, pages 357–372. Chapman & Hall/CRC, 1 edition.

Dongarra, J., Meuer, H. W., and Strohmaier, E. (2017). TOP500 Supercomputer Sites.

Dorier, M., Sisneros, R., Gomez, L. B., Peterka, T., Orf, L., Rahmani, L., Antoniu, G., and Bougé, L. (2016). Adaptive Performance-Constrained In Situ Visualization of Atmospheric Simulations. In *CLUSTER '16 Proceedings of the IEEE International Conference on Cluster Computing*, pages 269–278. IEEE.

Fabian, N., Moreland, K., Thompson, D., Bauer, A. C., Marion, P., Gevecik, B., Rasquin, M., and Jansen, K. E. (2011). The ParaView Coprocessing Library: A scalable, general purpose in situ visualization library. In *LDAV '11 Proceedings of the IEEE Symposium on Large Data Analysis and Visualization*, pages 89–96. IEEE.

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

Inacio, E. C., Pilla, L. L. L., and Dantas, M. A. R. (2015). Understanding the Effect of Multiple Factors on a Parallel File System's Performance. In *WETICE '15 Proceedings of the 24th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 90–92. IEEE.

Miyamoto, Y., Kajikawa, Y., Yoshida, R., Yamaura, T., Yashiro, H., and Tomita, H. (2013). Deep moist atmospheric convection in a subkilometer global simulation. *Geophysical Research Letters*, 40(18):4922–4926.

Miyazaki, H., Kusano, Y., Shinjou, N., Shoji, F., Yokokawa, M., and Watanabe, T. (2012). Overview of the K computer System. *Fujitsu Scientific and Technical Journal*, 48(3):255–265.

Nonaka, J., Ono, K., Bi, C., Sakurai, D., Fujita, M., Oku, K., and Kawanabe, T. (2016). HIVE: A Visualization and Analysis Framework for Large-Scale Simulations on the K Computer. In *PacificVis '16 Proceedings of the IEEE Pacific Visualization - Poster Session*. IEEE.

Nonaka, J., Ono, K., and Fujita, M. (2014). Multi-step image compositing for massively parallel rendering. In *HPCS '14 Proceedings of the International Conference on High Performance Computing & Simulation*, pages 627–634. IEEE.

Ono, K., Kawashima, Y., and Kawanabe, T. (2014). Data Centric Framework for Large-scale High-performance Parallel Computation. *Procedia Computer Science - Special Issue: International Conference on Computational Science, ICCS 2014*, 29:2336–2350.

Roten, D., Cui, Y., Olsen, K. B., Day, S. M., Withers, K., Savran, W. H., Wang, P., and Mu, D. (2016). High-Frequency Nonlinear Earthquake Simulations on Petascale Heterogeneous Supercomputers. In *SC '16 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE.

Satoh, M., Tomita, H., Yashiro, H., Kajikawa, Y., Miyamoto, Y., Yamaura, T., Miyakawa, T., Nakano, M., Kodama, C., Noda, A. T., Nasuno, T., Yamada, Y., and Fukutomi, Y. (2017). Outcomes and challenges of global high-resolution non-hydrostatic atmospheric simulations using the K computer. *Progress in Earth and Planetary Science*, 4(1):24.

Tsujita, Y., Yoshizaki, T., Yamamoto, K., Sueyasu, F., Miyazaki, R., and Uno, A. (2017). Alleviating I/O Interference Through Workload-Aware Striping and Load-Balancing on Parallel File Systems. In Kunkel, J. M., Yokota, R., Balaji, P., and Keyes, D., editors, *High Performance Computing*, volume 10266 of *Lecture Notes in Computer Science*, pages 315–333. Springer.

Optimizing the Decoding Process of a Post-Quantum Cryptographic Algorithm

Antonio Guimarães[1], **Diego F. Aranha**[1], **Edson Borin**[1]

[1]Institute of Computing – University of Campinas (UNICAMP)
Av. Albert Einstein, 1251 - 13.083-852 - Campinas - SP - Brazil

antonio.junior@students.ic.unicamp.br, {dfaranha, edson}@ic.unicamp.br

Abstract. *QcBits is a state-of-the-art constant-time implementation of a code-based encryption scheme for post-quantum public key cryptography. This paper presents an optimized version of its decoding process, which is used for message decryption. Our implementation leverages SSE and AVX instructions extensions and performs 3.6 to 4.8 times faster than the original version, while preserving the 80-bit security level and constant time execution. We also provide experimental data that indicates a further 1.4-factor speedup supposing the existence of instructions for vectorial conditional moves and 256-bit register shifts. Finally, we implemented countermeasures for side-channel security and showed that they do not affect the overall performance.*

1. Introduction

Recent developments on quantum computers and cryptanalysis create the need for new efficient and secure algorithms for public key cryptography to replace current standards. Conventional algorithms, mostly represented by elliptic curves [Koblitz 1987] and RSA [Rivest et al. 1978], depend on the hardness of integer factorization and computing discrete logarithms, which can be efficiently solvable in a quantum computer by Shor's algorithm [Shor 1999].

Public-key algorithms that do not rely on these problems are known since the late 70s and are today called post-quantum cryptographic techniques. Among them, code-based encryption is becoming one of the most promising. First proposed by McEliece using Goppa codes [McEliece 1978], it was never considered a reasonable alternative until the quantum computing rise, since it was clearly outperformed by other algorithms. Besides the difference in execution time, the very large public keys were also one of the main issues preventing it from becoming competitive.

Aiming to reduce the key size and improve performance while maintaining the security level, many different choices of codes and decoding algorithms were proposed in the past few years. Misoczki et al. [Misoczki et al. 2013] proposed the McEliece cryptosystem instantiation with quasi-cyclic structure with moderate density parity check codes (QC-MDPC). Although recent implementations show that the decoding process is slower than the originally proposed Goppa Codes, it allows a key size reduction from 20,480 bits to 4,801 bits, while keeping the 80-bit classical security level.

Side-channel attacks are also an important factor to consider, and secure implementations should not leak information correlated with critical data (keys and plaintext). More recently, Chou presented QcBits [Chou 2016], a constant-time implementation of a

QC-MDPC code-based encryption scheme. Using polynomial representation and a technique called bitslicing, the implementation is fully protected against timing attacks and two times faster than the previous speed record. This last accomplishment is specially important since the decoding process is probabilistic and executing it in constant-time may be very inefficient and lead to high decoding error rates.

The QcBits implementation was provided in two versions: `ref`, which uses only C code, and `clmul`, which performs arithmetic using the 128-bit carry-less multiplication instruction. Although the original implementations are fully constant-time, they are vulnerable against other side-channel attacks based on power consumption [Rossi et al. 2017].

In this paper, we present the following contributions:

- We optimized the decoding process for both the QcBits versions, achieving a speedup of 3.6 times over the `clmul` version and 4.8 times over `ref`.
- We estimated that gains could be as high as 5.06 times on `clmul` version if new instructions for conditional vectorial moves and 256-bit register shifts were added on the architecture.
- We mitigate all currently known power vulnerabilities found in the original implementation with an almost negligible ($< 1\%$) impact to the overall performance.

Our performance improvement comes from vectorization using AVX instructions, loop unrolling on hot spots and pre-calculation of vector rotations. All the performance measurements were executed on Haswell and Skylake architectures. We focus on the 80-bit security level for comparisons against related work, but higher security levels can be achieved with minimal changes to the implementation and all optimization techniques presented in the paper are still applicable. To the best of our knowledge, our paper is the first to present a fully vectorized software optimization of QcBits.

This work is organized as follows: Section 2 described code-based cryptography and the QcBits algorithm; Section 3 discusses and presents the results of our optimization on the decoding process of QcBits; Section 4 explains the countermeasure applied to mitigate a power channel vulnerability found in the original implementation; Section 5 presents some related work; and Section 6 discusses the final conclusions and future work.

2. Code-based Cryptography and the QcBits algorithm

The McElice cryptosystem [McEliece 1978] was the first code-based encryption scheme ever proposed and still remains as the most relevant one. Its security is based on the hardness of decoding linear codes, which is an NP-complete problem. The original scheme used Goppa Codes, which enabled great performance due to very efficient decoding algorithms, but keys took 460Kb at the 80-bit security [Bernstein et al. 2008], making the system not competitive in terms of viability among the alternatives.

Equation 1 shows the encryption in the McEliece Cryptosystem: m is a message of length k; z is an error vector with Hamming Weight t; and G' is a $k \times n$ matrix defined on Equation 2, where S is a scrambling matrix, G is the generator matrix for the chosen code (e.g. Goppa Code) and P is a permutation matrix. All these matrices are randomly generated and the last 3 of them compose the cryptosystem private key, while their product G' is the public key. The decryption is shown on Equation 3, where $Decode$ is the decoding algorithm for the chosen code.

Using Goppa codes at the 80-bit security, the parameters k, n, and t are chosen respectively as 1632, 1269 and 34 bits, resulting in the 460Kb public key size. Many techniques were proposed in order to reduce the key size of Goppa codes. Misoczki and Barreto [Misoczki and Barreto 2009] proposed a dyadic structure, but although they successfully presented a viable small-key alternative with just 20Kb, it resulted in structural vulnerabilities [Faugere et al. 2010].

$$c' = mG' + z \tag{1}$$

$$G' = SGP \tag{2}$$

$$m = Decode(cP^{-1})S^{-1} \tag{3}$$

As an alternative to Goppa codes, QC-MDPC codes [Misoczki et al. 2013] were first introduced in 2013, allowing the use of very compact keys. Table 1 shows a key length comparison between QC-MDPC codes and some of the previous alternatives.

Table 1. Key length in bits for different codes (from [Misoczki et al. 2013])

Security Level	QC-MDPC	QD-Goppa	Goppa
80	4,801	20,480	460,647
128	9,857	32,768	1,537,536
256	32,771	65,536	7,667,855

This great reduction in key size was achieved thanks to the quasi-cyclic (QC) structure. As discussed before, the public and private keys in the cryptosystem are all matrices composed of circulant blocks and this quasi-cyclic structure allows them to be represented by the first row only, as illustrated below:

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Besides exploiting the quasi-cyclic structure, another advantage of QC-MDPC codes is eliminating the need for scrambling and permutation matrices in the McEliece cryptosystem. The generator matrix G is the public-key and a parity check matrix H is the private key used in the decoding algorithm. Considering this, the decryption process boils down to the plain decoding, which is shown on Algorithm 1. For these codes, the generator matrix G is the row reduced echelon form of the parity check matrix H.

In Algorithm 1, H is the parity check matrix, c is the ciphertext and TH is an experimentally determined variable threshold, depending on the approach. The first line is the syndrome calculation, defined as the multiplication between the parity check matrix and the ciphertext. If it results in a zero vector, then there is no error in the message and therefore the decoding successfully finishes. Otherwise, the bit-flipping algorithm, represented by the `for` loop on Algorithm 1, must be executed. This algorithm works by calculating the Hamming Weight, which is the number of ones, of the `and` between each parity check matrix column and the calculated syndrome. If this Hamming Weight is greater than the threshold, the function $FlipBit$ flips the i-th bit (corresponding to the column) of the ciphertext c. Once the ciphertext is changed, the process restarts.

Algorithm 1: QC-MDPC Bit-flipping decoding.

Input : H, c and TH
Output: c

$s \leftarrow H \times c$
while $s \neq 0$ **do**
  foreach *column* h_i ***in*** H **do**
    $hd \leftarrow HammingWeight(h_i \wedge s)$
    if $hd > TH$ **then**
      $FlipBit(c, i)$
  end
  $s \leftarrow H \times c$
end

2.1. QcBits

QcBits [Chou 2016] is a state-of-the-art implementation of an encryption scheme based on QC-MDPC codes. It established new speed records and is the first fully constant-time implementation for this type of code. The speed improvement was achieved by representing the cryptosystem matrices, represented by its first row, as binary polynomials over $(x^r - 1)$, where r is a code size parameter. This was possible thanks to the quasi-cyclic structure. This way, all the syndrome calculations were done as polynomial multiplications instead of the less efficient general matrix multiplications. The polynomial view also helps with key generation, where the generator matrix was calculated using a polynomial inversion of the parity check matrix.

QcBits was presented in two versions: the C-only `ref` version, and the `clmul` version using the `PCLMULQDQ` instruction [Gueron and Kounavis 2010] to accelerate polynomial arithmetic. On both version, the bit-flipping in Algorithm 1 was implemented using constant-time vector rotations and bitslicing. Since it was the main target of our optimization, it will be further explained in the following subsection.

Aside from raw performance, constant-time execution was also an important implementation feature. It enabled the code to be resistant against timing side channel attacks, which was a problem for the previous implementation [Strenzke et al. 2008]. The decoding algorithm was the most challenging part of the implementation to protect. As shown in Algorithm 1, the original form of the algorithm is inherently variable time because the decoding only stops when all errors are corrected. To work around this problem, QcBits determined a maximum number of iterations for the decoding (6 at the 80-bit security level), and failure otherwise. There's no proof or strict estimate indicating that 6 iterations are enough for a practical secure use of the implementation, but empirical tests showed an acceptably low failure rate.

2.1.1. Bit-flipping algorithm

Algorithm 2 shows the implementation of decoding in QcBits. TH is the iteration threshold, s is the syndrome, c is the ciphertext and H' the sparse representation of the parity check matrix, which is an array of non-zero indices. The $BitSliceAdder$ function consists in adding each bit individually by positioning and storing each bit of the result in an array position (Algorithm 3), similarly to a half adder circuit. The $BitSliceSubtractor$ is implemented in the same way, but with a full adder or subtractor instead.

Algorithm 2: QcBits Bit-flipping implementation logic

Input : H', c, s and TH
Output: c

1 $N \leftarrow 1 + \lceil log_2(|H'|) \rceil$
2 $sum[N] \leftarrow 0's$
3 **foreach** *index* i ***in*** H' **do**
4 $\quad w \leftarrow s \lll i$
5 $\quad sum \leftarrow BitSliceAdder(sum, w)$
6 **end**
7 $sum \leftarrow BitSliceSubtractor(sum, TH)$
8 $c \leftarrow \neg sum[N-1] \oplus c$

Algorithm 3: BitSlice Adder Implementation Logic

Input : N, sum and w
Output: sum

1 **for** $i = 0$ ***to*** N **do**
2 $\quad c_{out} \leftarrow sum[i] \wedge w$
3 $\quad sum[i] \leftarrow sum[i] \oplus w$
4 $\quad w \leftarrow c_{out}$
5 **end**

Line 1 in Algorithm 2 calculates the number of bits necessary to represent the number of elements belonging to H', which is the maximum result that can be stored on the sum array by the $BitSliceAdder$. Line 2 initializes sum with zeros. The loop on line 3 iterates over the private key indices: for each index, the syndrome is rotated left on the index value (line 4) and the result is added to the sum array using the $BitSliceAdder$ function. This process is equivalent to calculating the Hamming Weight of the bitwise `AND` between each matrix column and the syndrome. However, for 80-bit security, instead of iterating over the 4801 rows of the parity check matrix, this method just needs to iterate over the 90 indices of the sparse matrix representation. At the end of the loop, the threshold is subtracted from the sum of each bit. If the most significant result bit is one on line 8, it indicates that the threshold is greater than the sum and the corresponding bit must not be flipped. Otherwise, the bit is flipped.

3. QcBits Optimization

We began our optimization by extending the vectorization to the whole code using SSE4 instructions for 128-bit registers, available since Intel Nehalem, and using AVX2 instructions for 256-bit registers, available since Haswell. Our initial expectation was obtaining a 2-factor speedup in the first case and 4 in the latter since these values correspond to the number of SIMD lanes found on these standards. Most of the code was composed of bitwise operations, such as `XOR` and `AND` of the bit slice adder, and were easily vectorizable, resulting in an immediate gain of 2.6 times when using the AVX2 instruction set. However, the absence of some instructions on the SIMD instruction sets prevented those expectations from materializing.

The main obstacle for vectorization was the implementation of 128-bit and 256-bit register shifts. These operations are necessary to perform the vector rotations shown on line 4 of Algorithm 2. For the 80-bit security level, the rotation target has 4801 bits and it

is implemented in two steps using C language: first, the words that compose the vector are permuted following the rotation logic; next, the rotation is done inside each word, shifting its bits and inserting next word bits in the shifted area. For registers with size lesser or equal to 64 bits there's a single instruction to shift all the register bits, which facilitates the implementation. For larger registers we had to perform a custom multi-instruction logic, making the implementation slower and more complex.

Listing 1 shows our implementation of a shift right with carry on AVX2 registers, used in the vector rotation shown on line 4 of Algorithm 2. The code is composed by 10 intrinsics for vector instructions. It works by permuting 64-bit sets to reduce the shift amount to less than 64, then the Carry In is inserted using the `BLENDV` instruction and the shift is finished using instructions that shifts inside the 64-bit lanes. Some of the used instructions are very expensive, like the `PERMUTEVAR` instruction on line 12 and 19, which has 3-cycle latency in Skylake, according to Agner Fog's instruction tables [Fog 2011].

```
1  unit bitShiftRight256xmmCarry (unit data, int count, unit * carryOut, unit carryIn){
2    unit innerCarry, out, countVet;
3    unit idx = _mm256_set_epi32(0x7, 0x6, 0x5, 0x4, 0x3, 0x2, 0x1, 0x0);
4    const unit zeroMask = _mm256_set_epi64x(-1, -1, -1, 0);
5    unit zeroMask2 = _mm256_set_epi8(0x80, 0x80, 0x80, 0x80, 0x80, 0x80, 0x80, 0x80,
6                                     0x82, 0x82, 0x82, 0x82, 0x82, 0x82, 0x82, 0x82,
7                                     0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84,
8                                     0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86);
9
10   countVet = _mm256_set1_epi8((count >> 5) & 0xE);
11   idx = _mm256_add_epi8(idx, countVet);
12   data = _mm256_permutevar8x32_epi32(data, idx); // rotate
13   *carryOut = data;
14   zeroMask2 = _mm256_sub_epi8(zeroMask2, countVet);
15   data = _mm256_blendv_epi8 (carryIn, data, zeroMask2);
16   // shift less than 64
17   count = (count & 0x3F);
18   innerCarry = _mm256_blendv_epi8(carryIn, data, zeroMask);
19   innerCarry = _mm256_permute4x64_epi64(innerCarry, 0x39); // >> 64
20   innerCarry = _mm256_slli_epi64 (innerCarry, 64 - count);
21   out = _mm256_srli_epi64 (data, count);
22   out = _mm256_or_si256 (out, innerCarry);
23   return out;
24 }
```

Listing 1: 256-bit register shift implementation

For the `clmul` version vectorized with AVX2 instruction, the syndrome calculation was also a problem. Executed at the beginning of the decoding process, it was originally implemented using the carry-less multiplication instruction which is only available for 128-bit size registers. Therefore, this code snippet, which takes approximately 20% of the code execution time, is stuck at the 128-bit implementation.

3.1. Basic Vectorization Results

We compiled the implementations using the three industry-standard compilers: GCC 6.1.3, CLANG 3.9.1 and ICC 17.0.2. For all the compilers, the compilation optimization flags used were `-O3` and `-march=native`. The flag `-funroll-all-loops` was also used when compiling with GCC. Equivalent flags for aggressive loop unrolling on the other compilers were tested, but they didn't result in any performance improvement and therefore were removed. The implementations were executed in two machines: the first one, named Haswell, uses an Intel i7-4770 processor and the second, named Skylake,

uses an Intel i7-6700k processor. Both machines run a Fedora 25 operating system and, aiming at experiment reproducibility and cycle accuracy, had the Intel Turbo Boost and Hyper-Threading mechanisms disabled. The performance results of this first vectorization are shown in the graph of Figure 1.

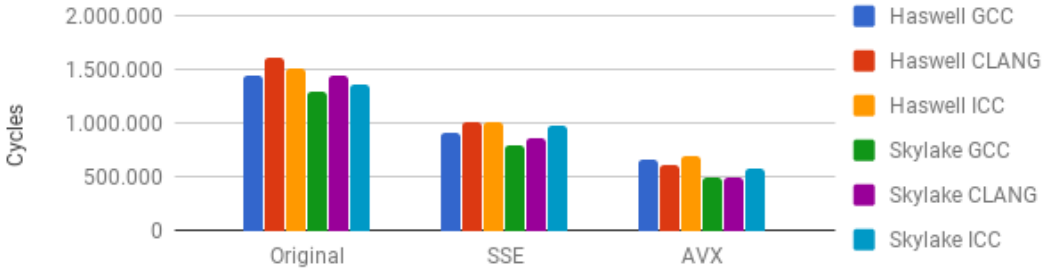


Figure 1. Initial vectorization results

As can be noted from the graph, the execution time, considering the compilation with GCC, reduced from 1,292,380 Skylake cycles and 1,441,220 Haswell cycles to respectively 803,970 and 912,498 cycles when using the SSE instruction set, which represents a speedup of 1.6 times; and to 501,473 and 669,596 cycles when using the AVX2 instruction set, which in turn represents a speedup of 2.6 times. The graph also shows the performance improvement between the two processors generations, especially for the vectorial versions: The Skylake processor is 10% faster than the Haswell processor on the original 64-bit version and on the SSE version, while for AVX2 version Skylake is 21% faster than Haswell. These conclusions are based on the average results obtained with the three compilers.

3.2. Vector Rotation Table

Although there is a likely more efficient implementation for Listing 1, it will probably be always inefficient without special hardware support. Instead of trying to optimize further our implementation, we focused on reducing the number of its executions. The word permutation of the vector rotation, which is shown on line 4 of Algorithm 2 and is composed of conditional copies and register shifts, represented almost 40% of the code execution time and 90 of them were calculated in the decoding implementation, one for each parity check matrix index. However, the permutation is done based on the first bits of each index and, using 256-bit registers and considering the 80-bit for the security level, there are only 32 possible permutations of words following the rotation logic.

Considering that, we construct a table of all possible word rotations in the beginning of the decoding process and just query that table instead of calculating the permutations every time. The graph in Figure 2 shows the correlation between the number of word rotations that were calculated and the number of possible rotations for each word size. As can be seen, the pre-calculated table of rotations would not be worth for the original 64-bit, but it is faster for all our optimized versions.

This approach, however, has some obstacles to be used in a constant time implementation. The table access pattern cannot depend on the private key because it would be leak cache-timing information that could be exploited on a side-channel attack [Strenzke et al. 2008]. Thus, every time the implementation needs to access the table

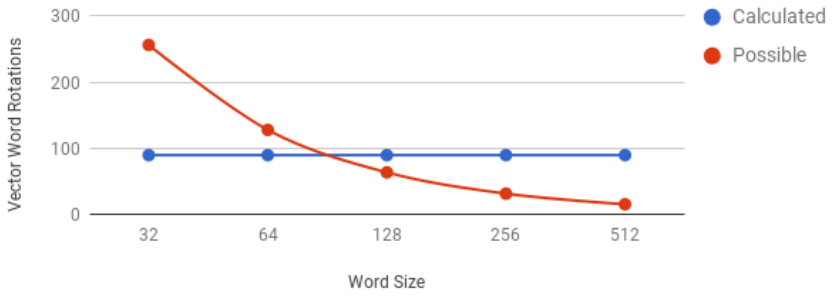


Figure 2. Number of word rotations computed and possible for each implementation

it must iterate over all the table elements conditionally copying each one of them. These extra memory accesses add a great performance penalty and the table of rotation alone became slower than the calculations even on 256-bit registers version.

Despite that, we were able to improve the rotation table by doing a trade-off between the calculation and the table access. Basically, we construct a table with just a small subset of the possible rotations. Then, when a rotation is needed, the implementation iterates over the table, picks the nearest rotation and calculates the pending rotation amount starting from the pre-calculated value. Since the rotation calculation is done based on each bit of the rotation amount, its performance is proportional to the logarithm of the maximum rotation amount. This way, we achieved a 1.19-factor speedup on the AVX version, when comparing to the basic vectorization time, using tables with 3 stored rotations. The number of Skylake cycles, when compiling with GCC, was further reduced from 501,473 cycles to 420,397 cycles and the overall speedup increase from 2.6 times to 3.1 times. The use of the rotation table also drastically reduced the number of iterations necessary to calculate the rotations. This reduction allowed a manual loop unrolling which leads to a 1.17-factor speedup over the best time, bringing the number of Skylake cycles when compiling with GCC down to 358,499 cycles.

All the presented optimization techniques were also applied to QcBits `ref` version, which uses only C code. Table 2 shows the results for all versions. The speedups relatively to the Original Version execution are shown in Figure 3.

Table 2. Final optimization results (in cycles)

Machine	Version	Compiler	Original	SSE	AVX2
Skylake	CLMUL	GCC	1,292,380	574,136	358,499
		CLANG	1,443,992	646,430	377,218
		ICC	1,368,697	878,976	449,620
	REF	GCC	2,097,282	844,992	492,390
		CLANG	2,236,178	944,803	470,578
		ICC	2,221,606	1,360,744	608,560
Haswell	CLMUL	GCC	1,441,220	788,436	529,956
		CLANG	1,610,954	829,896	528,188
		ICC	1,506,562	918,084	555,844
	REF	GCC	2,216,498	1,132,052	679,122
		CLANG	2,391,762	1,205,842	651,032
		ICC	2,337,726	1,306,476	716,208

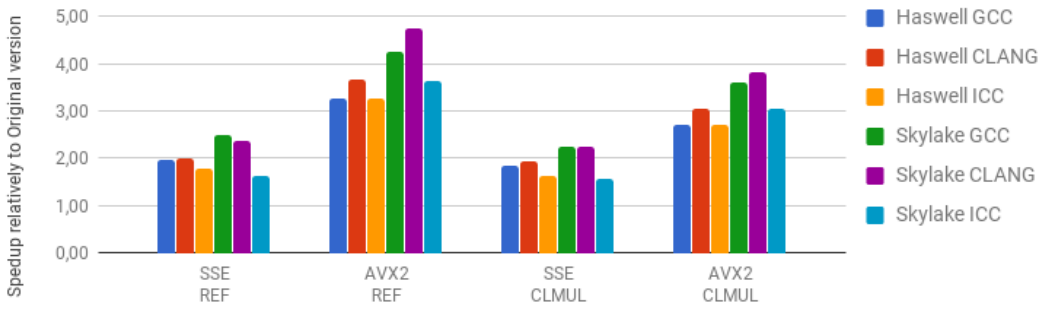


Figure 3. Final speedups achieved with the optimization (relatively to the corresponding Original version execution)

Analyzing the `clmul` version results in the graph, we can note that the results using SSE instructions overcome our initial expectation with a speedup of 2.25 times on Skylake with GCC. This was possible thanks to the rotation table use and the loop unrolling, previously explained. For the version vectorized with AVX2 instructions, however, the speedup is still below 4 times, being 3.6 times with GCC and 3.8 times with CLANG, both on Skylake. It happens mostly because of the absence of a 256-bit `clmul` instruction, which creates the need of the use of 128-bit register instructions in the syndrome calculation. This hurts the performance not only because of the use of small registers but also due to the transition between the instruction set extensions, which is known to be expensive [Lomont 2011].

The `ref` version uses the same constant-time rotation process showed in Algorithm 2 to calculate the matrix multiplications. The only modification in the algorithm is that the $BitSliceAdder$ is replaced by a simple `XOR`. Once this method doesn't rely on `clmul` instruction, the `ref` version could be better optimized, achieving a speedup of 4.75 times with a time of 470,578 cycles on Skylake with CLANG.

3.3. Estimations for possible improvements

As explained in the beginning of this section, the two main hindrances for the vectorization were the absence of vector instructions for register shifts and conditional moves. This last procedure is currently done by the `BLENDV` instruction, which is much more powerful and, hence, expensive than we need for this purpose. Although Fog [Fog 2011] reports a throughput of 1 cycle for this instruction, it is difficult to implement the instruction usage in a sequential way to achieve this time.

In order to estimate the possible gains if these two instructions exist, we experimented with the `clmul` version to suppose their existence. The experiment was done by replacing the `BLENDV` instruction with two addition instructions and the vector shift algorithm by a simple 64-bit lanes shift. This version, of course, doesn't result in the correct output, but it serves as a fair estimation. Testing on Skylake and compiling with GCC, we execute this version in 255.274 cycles, which represents a 1.4 times speedup compared to our best correct version and a total speedup of 5.06 times over the `clmul` version.

4. Power side-channel vulnerability

A simplified snippet of the original implementation code used for the word rotation is shown on Listing 2. As previously discussed, the rotation amount depends directly on the secret key bits and, therefore, must be executed mitigating all side channel leakages. On line 1, a mask is constructed using the variable `sk_bit`, which represents a secret key bit: If the bit is one, then the mask will be all ones, otherwise, the mask will be all zeros. Following this, on line 4, the vector is copied shifted or not depending on this mask.

```
1 mask = 0 - sk_bit;
2 us = 1 << i; // shift amount
3 for (j = 0; j < LEN; j++)
4     w[ j ] = (x[ j + us ] &  mask) ^ (x[ j ] & ~mask);
```

Listing 2: Vulnerable implementation of conditional copy for vector rotation

The problem lies on the fact that the power consumption of setting all bits in a register is perceptibly higher than keeping the register with all its bits zero. An attacker can exploit that fact and discover the secret key through a power measurement of the algorithm execution [Nascimento et al. 2016]. We are able to mitigate this vulnerability by using the instruction `BLENDV`, as shown in Listing 3. This vulnerability used to occur not only in the word rotation, but in all conditional copies implemented in the original version. We fix all of them in the same way and verified that this modification had very little impact on the overall performance ($< 1\%$). The performance results presented in Section 3 already include this modification.

```
1 mask = _mm_set1_epi8(sk_bit << 7);
2 us = 1 << i; // shift amount
3 for (j = 0; j < LEN; j++)
4     w[ j ] = _mm_blendv_epi8(x[ j ], x[ j + us ], mask);
```

Listing 3: Secure implementation of conditional copy for vector rotation

5. Related Work

Most of the work related to the QcBits implementation is research on side-channel attacks. Rossi et al. [Rossi et al. 2017] presented a side-channel power-based attack against the syndrome calculation of QcBits. The attack exploited a power-leakage at the store of the rotated code-word (line 4 of Algorithm 2). They also provided a simple countermeasure in order to prevent the attack. We did not apply this countermeasure in this paper since the use of registers greater than 128 bits makes the attack complexity much higher than the target security level. Guo et al. [Guo et al. 2016] presented an attack exploiting a relation between the parity check matrix bits and the decoding failure rate of the algorithm. The attack was named Reaction Attack and is capable of recovering the private key of QcBits in minutes. No effective countermeasure was proposed for this attack yet. Considering implementation work, Hu and Cheung [Hu and Cheung 2017] presented a hardware implementation of QC-MDPC codes partially based on QcBits implementation. Using a Xilinx Virtex-6 FPGA, they achieved a 53% area-time product gain comparing to the previous designs for QC-MDPC codes.

To the best of our knowledge, our paper is the first to present a fully vectorized software optimization of QcBits. The use of vector instructions for cryptographic algorithm optimization, however, is quite common. Chang et al. [Chang et al. 2015] presented an optimized implementation of the RSA algorithm which achieved speedups of 4.3 to 5.9 times using the AVX-512 instruction set, and Hamburg [Hamburg 2012] presented an implementation of Elliptic-Curve Cryptography using SSE and AVX instructions. Considering the code-based cryptography field, specifically, there are also some optimization work using vector instructions. Maurich et al. [Maurich et al. 2015] presented an implementation of QC-MDPC codes using the SSE instruction set which was considered the speed record for these codes before the QcBits publication.

6. Conclusion

In this paper, we presented an optimized implementation of decoding process in QcBits. We vectorized the entire algorithm, inserted a table of pre-computed vector rotations and unrolled the rotation calculation loop for the versions `ref` and `clmul`. In the `ref` version, using the SSE and AVX2 instruction sets, we achieved a maximum speedup of 2.48 and 4.75 times, respectively, while in the `clmul` version we achieved a speedup of 2.23 and 3.6 times when using SSE and AVX2 instructions and compiling with GCC. We also implemented countermeasures for some known side channel vulnerabilities without any significant performance penalty.

Our results clearly demonstrate the algorithm's aptitude for vectorization. The `ref` version, which does not rely on the `clmul` instruction, presented higher gains than the register size increment, showing the great impact of the rotation pre-computation technique. The same occurred with `clmul` version vectorized with SSE instructions. The use of the table could also be much more efficiently implemented if the hardware provided constant-time memory accesses. Besides that, we also demonstrated that some hardware improvements, such as shifting and conditional move instructions for 128-bit and 256-bit registers, can be very useful for the algorithm performance, as shown by our 1.4-factor speedup estimation considering these instructions. A 256-bit version of the `clmul` instruction would also provide significant performance gains.

Considering the current post-quantum cryptography scenario, the code-based cryptography field is just beginning its rise and, considering the latest performance improvements, it is shaping up as one of the most promising candidates for that end. As future work, we intend to implement an AVX-512 version of the decoding process and to optimize the key-pair generation and the encryption process of QcBits. Also, some countermeasure must be developed to mitigate the Reaction Attack.

Acknowledgements

The authors would like to thank Intel Labs and the São Paulo Research Foundation (FAPESP) for supporting this research under grant 14/50704-7.

References

Bernstein, D. J., Lange, T., and Peters, C. (2008). Attacking and defending the mceliece cryptosystem. Cryptology ePrint Archive, Report 2008/318. `http://eprint.iacr.org/2008/318`.

Chang, C., Yao, S., and Yu, D. (2015). Vectorized big integer operations for cryptosystems on the intel mic architecture. In *High Performance Computing (HiPC), 2015 IEEE 22nd International Conference on*, pages 194–203. IEEE.

Chou, T. (2016). Qcbits: constant-time small-key code-based cryptography. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 280–300. Springer.

Faugere, J.-C., Otmani, A., Perret, L., and Tillich, J.-P. (2010). Algebraic cryptanalysis of mceliece variants with compact keys. In *Eurocrypt*, pages 279–298. Springer.

Fog, A. (2011). Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. *Copenhagen University College of Engineering*.

Gueron, S. and Kounavis, M. E. (2010). Intel® carry-less multiplication instruction and its usage for computing the gcm mode. *White Paper*.

Guo, Q., Johansson, T., and Stankovski, P. (2016). A key recovery attack on mdpc with cca security using decoding errors. In *Advances in Cryptology–ASIACRYPT 2016*, pages 789–815. Springer.

Hamburg, M. (2012). Fast and compact elliptic-curve cryptography. *IACR Cryptology ePrint Archive*.

Hu, J. and Cheung, R. C. (2017). Area-time efficient computation of niederreiter encryption on qc-mdpc codes for embedded hardware. *IEEE Transactions on Computers*.

Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of computation*.

Lomont, C. (2011). Introduction to intel advanced vector extensions. *Intel White Paper*.

Maurich, I. V., Oder, T., and Güneysu, T. (2015). Implementing qc-mdpc mceliece encryption. *ACM Transactions on Embedded Computing Systems (TECS)*.

McEliece, R. J. (1978). A public-key cryptosystem based on algebraic. *Coding Thv*.

Misoczki, R. and Barreto, P. S. (2009). Compact mceliece keys from goppa codes. In *Selected Areas in Cryptography*, pages 376–392. Springer.

Misoczki, R., Tillich, J.-P., Sendrier, N., and Barreto, P. S. (2013). Mdpc-mceliece: New mceliece variants from moderate density parity-check codes. In *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*, pages 2069–2073. IEEE.

Nascimento, E., Chmielewski, L., Oswald, D., and Schwabe, P. (2016). Attacking embedded ecc implementations through cmov side channels. *IACR Cryptology ePrint Archive*.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*.

Rossi, M., Hamburg, M., Hutter, M., and Marson, M. E. (2017). A side-channel assisted cryptanalytic attack against qcbits. Cryptology ePrint Archive, Report 2017/596. `http://eprint.iacr.org/2017/596`.

Shor, P. W. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*.

Strenzke, F., Tews, E., Molter, H. G., Overbeck, R., and Shoufan, A. (2008). Side channels in the mceliece pkc. In *International Workshop on Post-Quantum Cryptography*, pages 216–229. Springer.

Propostas de Otimização de uma Implementação do Algoritmo de Análise Diferencial de Potência

Rodrigo Bazo[*], Diego Poleto[†], Gerson Geraldo H. Cavalheiro, Rafael Soares

[1]Programa de Pós-Graduação em Computação
Universidade Federal de Pelotas
Pelotas – RS – Brasil

rbazo@unisinos.br
{diegopoleto, gerson.cavalheiro,rafael.soares}@inf.ufpel.edu.br

Resumo. *Ataques por canais laterais exploram vulnerabilidades de um sistema criptográfico por meio do monitoramento de grandezas físicas tais como consumo de energia e tempo de execução. Estes ataques exigem grande poder computacional por atuarem em uma grande quantidade de dados e por explorar diferentes variações do ataque durante a execução. Neste trabalho é apresentada uma implementação inicial de Análise Diferencial de Potência (DPA) e são propostos quatro modos de otimização a fim de explorar seu paralelismo e reduzir o tempo de execução. Os resultados obtidos em uma arquitetura multiprocessada mostram ganhos significativos de desempenho, reduzindo em até 3000 vezes o tempo de execução do algoritmo DPA.*

1. Introdução

A Internet disponibilizou à população facilidades na realização de transações como compras de produtos, pagamento de contas, operações bancárias e semelhantes. A ubiquidade desejada no acesso a estes recursos fez com que diferentes equipamentos, como terminais bancários para auto atendimento e máquinas leitoras de cartão, sejam disponibilizados em locais públicos. Com a utilização em massa destes serviços e equipamentos, há uma grande preocupação com o sigilo dos dados, sejam os dados relativos às transações efetuadas, sejam os dados relativos às senhas de acesso.

Para garantir sigilo dos dados, são utilizados protocolos de comunicação e criptografia com a finalidade de ocultar o conteúdo de uma determinada mensagem quando em trânsito entre sua origem e seu destino. Neste caso, o sigilo das mensagens fica condicionado a chave criptográfica, uma pequena palavra que deve ser conhecida apenas pelos entes comunicantes com chave simétricas. Apesar disso, existe a criptoanálise, ciência dedicada a encontrar vulnerabilidade em algoritmos criptográficos a fim de extrair informações sigilosas de sistemas que usam algoritmos criptográficos.

Os algoritmos criptográficos evoluíram significantemente nos últimos anos, muito em função do fato de os algoritmos serem públicos e da intensa atividade de criptoanálise. Considerando isto, existe um grande esforço da comunidade científica para identificar as vulnerabilidades de sistemas criptográficos a fim de encontrar soluções que possam resistir às ameaças provocadas por ataques.

[*]Bolsista IC Capes
[†]Bolsista IC Capes

De acordo com [Waddle and Wagner 2004], existem dois tipos de ataques aos sistemas criptográficos: aqueles que procuram falhas nos próprios algoritmos de criptografia e aqueles que exploram as vulnerabilidades físicas da implementação do sistema. Neste segundo caso, o ataque é realizado por meio do monitoramento do dispositivo físico que está executando o algoritmo criptográfico. Esta abordagem é conhecida como Ataque por Canais Laterais (do inglês, *Side-Channel Attacks* – SCA) [Kocher 1996].

A Análise Diferencial de Potência (*Differential Power Analysis* – DPA) é um ataque do tipo SCA que, por meio de análises estatísticas, estabelece uma dependência entre os dados processados por algoritmos criptográficos tais como (*Data Encryption Standard* - DES), (*Advanced Encryption Standard* - AES) e o consumo de energia do circuito que o executa [Kocher et al. 1999]. Ataques DPA são principalmente aplicados a sistemas criptográficos que utilizam o algoritmo DES ou sua versão moderna, o AES . O grande desafio é aumentar o nível de segurança destes sistemas, certificando-se a robustez de seus métodos de segurança ao vazamento de informações por meio de sua dissipação de potência antes de sua disponibilização ao mercado consumidor.

DPA é um tipo de análise que demanda alto poder de processamento por manipular uma grande quantidade de traços contendo o consumo de corrente do dispositivo atacado e consequentemente revelando a potência dissipada durante o processamento. Por se tratar de um método estatístico, sendo maior a quantidade de traços de medição de consumo energético, melhor será a qualidade da análise estatística e, portanto, melhor o resultado da criptoanálise. O estudo de caso realizado neste artigo explora a otimização deste algoritmo do ataque DPA explorando sua paralelização em uma arquitetura multiprocessada a fim de dinamizar as pesquisas relacionadas à área de criptoanálise pela redução do tempo necessário à obtenção de resultados de análises. O ponto de partida se dá pelo desenvolvimento de um código em Matlab para sua implementação e, na sequência, sua tradução para C++ e sucessivas otimizações de código visando desempenho por meio de paralelização em diferentes níveis.

O restante deste artigo está organizado como segue. A Seção 2 descreve o funcionamento do algoritmo DES, como processo usado para encriptar e decriptar traços da análise de potência, bem como alguns tipos de ataques existente, destacando aquele usado como estudo de caso. A Seção 3 fornece embasamento sobre o funcionamento dos ataques por canais laterais em meio a implementação das funções *DPA* e *KeySearch*, que se destacam pelo alto custo computacional, sendo assim alvos para otimização. A Seção 4 refere-se a trabalhos relacionados, menciona implementações que se destacam por explorar o paralelismo do algoritmo DPA e o uso de GPUs para melhoria do tempo de execução. Na seção 5 estão presentes as propostas de otimização do trabalho, assim como os resultados obtidos em cada etapa. As conclusões são apresentadas na Seção 6.

2. Algoritmo de criptografia DES

Os algoritmos DES e AES são *simétricos* por utilizarem a mesma chave criptográfica para cifrar e decifrar os dados. O algoritmo DES usado neste trabalho como estudo de caso, é composto de três etapas: permutação inicial, 16 rodadas de execução de uma função aplicada sobre a mensagem de entrada e chave criptográfica e uma permutação final. Os dados são efetivamente processados em blocos de 56 bits, sendo 8 bits de paridade. Os ataques geralmente ocorrem nas SBOXs, pois é onde a chave e os dados estão sendo

encriptados. Na Figura 1 a estrutura de uma rodada do algoritmo DES é representada, os passos das rodadas são descritos a seguir de acordo com [Stallings 2008]:.

1. Um bloco de dados de entrada de 64 bits, onde 8 bits são paridade, reduzindo-se a um bloco de 56 bits de informação propriamente dita; Em seguida, este bloco é dividido em dois blocos;
2. Paralelo a este fluxo, a chave criptográfica de 64 bits sofre uma permutação inicial e redução para 48 bits chamada *Round Key*, onda a cada rodada a chave sofre uma operação de escalonamento;
3. Aplicam-se operações lógicas OU-Exclusivo (do inglês, *Exclusive Or*) utilizando as *Round Keys*, que são calculadas a partir da chave do usuário;
4. Os resultados são divididos e encaminhados para 8 Sboxes (do inglês, *Substitution Boxes*); Novamente é aplicada a operação OU-Exclusivo; Ao fim da rodada é realizada a união dos blocos de dados para formar a saída.

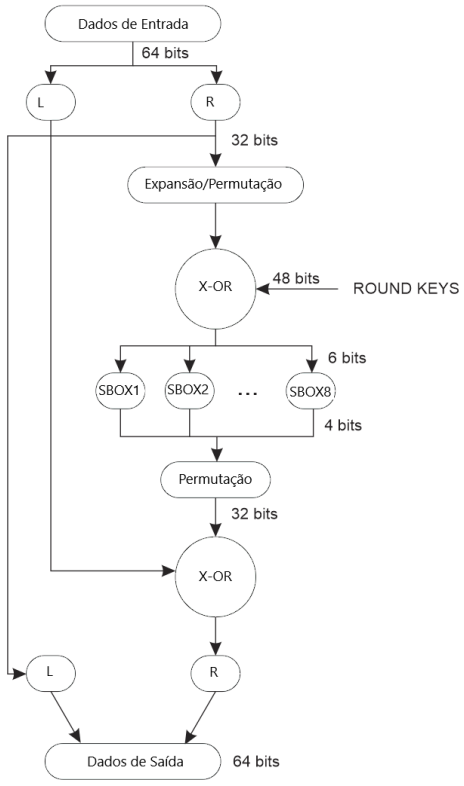


Figura 1. Fluxograma de execução das rodadas do DES.

Ao fim das 16 rodadas é executada uma permutação final e tem-se então o dado cifrado. Na Figura 2, está ilustrado o comportamento de um traço de consumo de um dado, sendo processado pelo algoritmo DES. Na figura é destacada a permutação inicial, as 16 rodadas do algoritmo e a permutação final.

3. Ataques por Canais Laterais

Os ataques por canais laterais visam explorar informações contidas em canais laterais, esses canais podendo ser, por exemplo, consumo de energia, tempo de execução e radiação

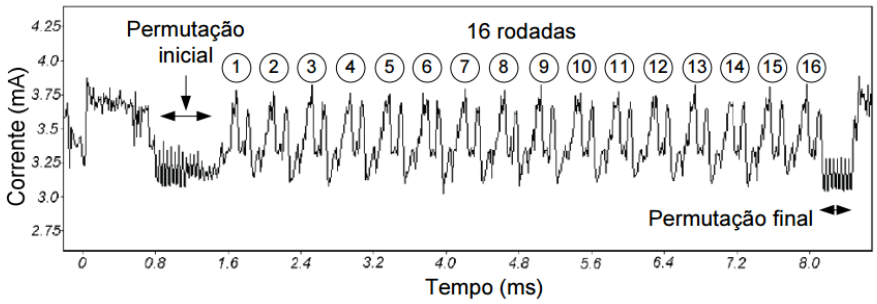


Figura 2. Modelo de traço de consumo de um processamento do DES .

eletromagnética. Os ataques por consumo de potência visam descobrir o dado processado analisando o consumo de energia do dispositivo executando o algoritmo criptográfico. Para isso é necessário o uso de um equipamento para aquisição do consumo de energia do dispositivo alvo do ataque, normalmente usa-se um osciloscópio para tal propósito conforme ilustrado na Figura 3. Os dispositivos atacados são normalmente smartcards, porém qualquer dispositivo poderia ser atacado como por exemplo um microcontrolador, FPGA ou um circuito integrado de aplicação específica (do inglês, Application Specific Integrated Circuit - ASIC).

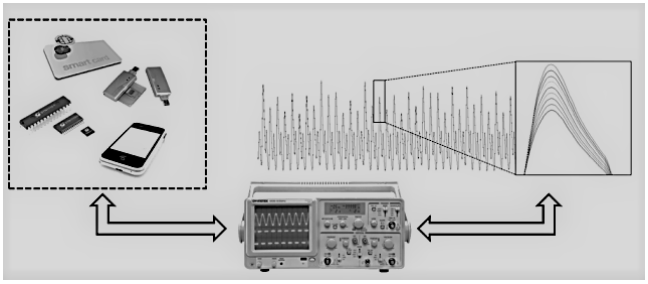


Figura 3. Alguns dispositivos que podem ser sujeitos à ataques por DPA.

3.1. Implementação do algoritmo DPA

O ataque visa analisar uma função alvo do algoritmo DES que relacione dado de entrada e chave criptográfica a fim de descobrir a chave criptográfica a partir dos traços de consumo de energia. No algoritmo DES as funções alvo normalmente são as SBOXs. Após obtidos os traços de consumo para um conjunto com diferentes dados de entrada e mesma chave criptográfica a etapa seguinte é realizar a análise propriamente dita. Para isso aplica-se uma função de seleção ao conjunto de traços adquiridos. Esta função divide o conjunto de traços disponíveis em dois grupos, a fim de identificar para cada hipótese de chave se o bit alvo do ataque é zero ou um.

O pressuposto do ataque é que o consumo de energia em circuitos digitais implementados com tecnologia CMOS difere para transições $0 \rightarrow 1$ e $1 \rightarrow 0$. Assim, o ataque classifica traços de consumo em dois grupos onde em um dado instante do tempo de processamento ocorre uma transição $0 \rightarrow 1$ e no outro, uma transição $1 \rightarrow 0$. A divisão correta de traços garante que, com o cálculo do traço médio de cada grupo, o consumo alvo do ataque seja evidenciado em um dado instante. Assim, a diferença entre os traços

médios obtidos produzirá uma diferença significativa entre os traços no dado instante alvo do ataque. A curva obtida pela diferença das médias de cada grupo é chamada de curva hipótese de chave. Uma curva para cada hipótese de chave é gerada durante a análise. A curva com maior amplitude no dado instante de processamento é considerada a curva de maior probabilidade de chave correta.

A função de seleção dos traços pode usar diferentes estratégias para desempenhar o ataque. Analisar individualmente os bits de saída de cada SBOX é estratégia mais comum. Neste caso é possível realizar ataques diferentes para cada um dos 4 bits de saída das SBOXs. Outra possibilidade de função seleção é analisar a soma dos 4 bits da SBOX (SUM). Uma terceira estratégia é avaliar o Peso Hamming (HW) da saída da SBOX. Além disso é possível executar um ataque que testa todas as estratégias de função seleção disponíveis (ALL). Tal como DPA, outras estratégias alternativas de ataques foram propostas, uma delas é CPA onde um modelo de consumo de energia é aplicado aos traços de consumo a fim de evitar influências de ruídos presentes nos traços e tornar o ataque mais eficiente [Brier et al. 2004].

A quantidade de traços envolvida no ataque impacta diretamente no custo computacional despendido, assim como a quantidade de amostras contidas nos traços. A implementação do algoritmo aninha alguns laços para realizar o ataque o que resulta no longo esforço computacional sem explorar o paralelismo do algoritmo. O ataque deve analisar todo o conjunto de traços que normalmente é superior a 50 mil. O algoritmo DES possui 8 SBOXs, logo são realizados 8 ataques para a obtenção de todos os segmentos da chave criptográfica. No entanto, cada SBOX pode ser atacada de diferentes maneiras conforme revisado, um ataque a cada bit, SUM, HW e CPA ou ainda ALL, a execução de todos eles. Isto é replicado para cada SBOX. Cada ataque gera uma curva hipótese para cada uma das 64 hipóteses de chave para a SBOX por meio da função *KeySearch*, uma das funções que mais demandam processamento do algoritmo, justamente por analisar e organizar as hipótese de chave para todos os tipos de ataque realizados. Logo, o uso do paralelismo juntamente com um acelerador gráfico deve reduzir o tempo de execução do algoritmo conforme [Swamy et al. 2014].

3.2. Função KeySearch

A função *KeySearch* mostrada na Figura 4 é responsável por percorrer a matrizes N e identificar as hipóteses de chave correta. No início da execução são inicializadas algumas variáveis como por exemplo a `matrixMax`, que armazena todos os valores máximos de cada curva obtida pela diferença das médias contida na matriz N a ser pesquisada.

Esta função pode usar dois métodos de busca da melhor hipótese de chave. Um método usa a maior amplitude da curva diferencial, enquanto o outro avalia a maior área sobre um intervalo pré-definido. Neste último caso, usa-se uma função que retorna a integração trapezoidal sob a curva. Ambos métodos são eficazes para a definição da melhor hipótese de chave na matriz N.

4. Trabalhos Relacionados

Em [Bartkewitz and Lemke-Rust 2011] é proposta uma implementação do ataque DPA dedicada para a execução em unidades de processamento de gráfico (GPUs). A implementação utiliza recursos avançados oferecidos pelo *CUDA Framework* para minimizar o

```
1: INITVARS()
2: for each keyHypothesis k do
3:     if method == maxPeak then
4:         Calculate matrixDPA maximum peak
5:     end if
6:     if method == maxIntegral then
7:         Calculate matrixDPA maximum trapezoidal integral
8:     end if
9: end for
10: keyGuessValue = MAXIMUM(matrixMax)
11: keyGuess = maximumValuePosition
12: matrixValuesIndexes = DESCENDINGSORT(matrixMax)
13: for each keyHypothesis k do
14:     if matrixValuesIndexes.Indexes(k) == roundKey then
15:         rankRealKey = k
16:         break
17:     end if
18: end for
19: keyGuessValue2 = GETSECONDMAXIMUM(matrixMax)
20: margin = CEIL(100 - ((keyGuessValue - keyGuessValue2)/keyGuessValue))
21: abscissa = matrixPoint[keyGuess] + xMin
```

Figura 4. Pseudocódigo genérico da função keySearch.

tempo de execução com base no coeficiente de correlação de Pearson. O modo de calcular este coeficiente poderia servir como uma referência para processamento de alto desempenho, pois como mostrado pelos autores é possível analisar um milhão de traços contendo 20 mil amostras em menos de dois minutos.

Em [Swamy et al. 2014], os autores propuseram a implementação de uma plataforma usando código aberto para execução de ataques a canais laterais, para avaliar a segurança de uma gama de dispositivos em relação a ataques SCA. Para isso foi desenvolvida uma estrutura de processamento paralelo com objetivo de acelerar cálculos intensivos explorando o paralelismo em nível de dados da aplicação, algo que é inerente aos algoritmos de ataque SCA. Para execução da implementação proposta utilizaram uma unidade de processamento gráfico (GPU), para acelerar a extração de chaves criptográficas.

No trabalho de [Brier et al. 2004] é proposta uma especialização de DPA, chamada de Análise por Correlação de Potência (*Correlation Power Analysis*). Esta análise utiliza o modelo de potência distância Hamming. Esta proposta visa reduzir o problema dos picos fantasmas em DPA. Atualmente existem alguns trabalhos que propõem modelos para acelerar a CPA, como no trabalho de [Gamaarachchi et al. 2014] onde os autores propõem a execução deste problema com GPUs utilizando CUDA.

Conforme revisado, os trabalhos encontrados na literatura exploram o paralelismo dos algoritmos de ataque DPA e execução em GPUs conhecidas por seu elevado número de elementos de processamento. No entanto, o uso de GPUs requer implementação dedicada com o uso de uma interface de programação para o uso dos recursos. Por outro lado, o uso de plataformas *multicore*, vários núcleos de processamento com possibilidade de execução de múltiplas *threads* não possui o mesmo desempenho em relação as executadas em GPUs, mas possuem a vantagem de manter a mesma estrutura do algoritmo sem a necessidade do uso de interfaces de programação específicas para cada tecnologia.

Neste trabalho é explorado o paralelismo de uma implementação do ataque DPA usando plataformas *multicore* para sua execução.

5. Propostas de Otimização da Implementação

Nesta Seção são apresentadas quatro propostas de otimizações aplicadas ao código que implementa o ataque DPA. A implementação disponível e validada em [Lomné et al. 2009] encontra-se codificada para o Matlab devido aos seus grandes recursos para operações e manipulações de matrizes. Entretanto, a execução de programas no Matlab é interpretada, sendo executada em uma máquina virtual o que não favorece implementações que exigem alto desempenho. Desta forma, otimizações são propostas e análises de desempenho são executadas para avaliar o tempo de execução das mesmas. As avaliações de desempenho consideram quatro estratégias da função de seleção de traços: Sum, HW, CPA e ALL, além dos seguintes parâmetros:

- Traço de consumo com 1.500 amostras;
- Métodos de busca da chave: maxPeak e maxIntegral;
- Modelo de consumo: HD (*Hamming Distance*).

Os experimentos foram realizados em um PC com a seguinte configuração: Processador Core i5 4690 com 4 cores físicos, 8 Gb de Memória RAM, executando o sistema operacional Ubuntu 14.04 e utilizando o compilador GCC versão 4.9. Os testes são realizados sobre um conjunto de 100.000 traços de consumo de energia provenientes de um sistema criptográfico alvo do ataque. Ataques executados sobre um conjunto desta ordem dispendem um tempo significativo de processamento. Um teste inicial mede o tempo de execução do ataque sobre o conjunto disponível. Em seguida, executa-se o mesmo ataque sobre um subconjunto de 5.000 traços, onde a cada 100 traços processados é medido e armazenado o tempo de execução. Ao final, calcula-se o tempo médio de execução de 100 traços e estima-se o tempo de execução para 100.000 traços. O tempo estimado é aproximadamente o mesmo obtido no teste inicial sobre todo o conjunto, garantindo a possibilidade das avaliações de desempenho apenas sobre um subconjunto dos traços e reduzindo os tempos de processamento.

5.1. Tradução do Algoritmo

O código desenvolvido em C++ respeita a estrutura do algoritmo apresentado em [Lomné et al. 2009]. No entanto, como a linguagem C++ não oferece os mesmos recursos matemáticos e funções específicas para cálculo de integral e manipulação de matrizes, algumas funções foram implementadas. Como esperado, o ganho de desempenho obtido nesta etapa de otimização foi significativo conforme mostrado na Figura 5. Deve-se observar que a estrutura inicial do algoritmo foi preservada a fim de avaliar-se apenas a mudança da linguagem de programação nesta primeira otimização proposta.

Os resultados obtidos nesta etapa de otimização destacam que apesar de oferecer diversas facilidades para implementação de operações com matriz, o Matlab não é uma solução interessante para aplicações que exigem alto desempenho. Por ser uma linguagem interpretada e oferecer um alto nível de abstração, o seu desempenho é inferior a implementações em linguagem compiladas tais com a linguagens como C++ gerando um código executável pelo processador e favorecendo aplicações de alto desempenho.

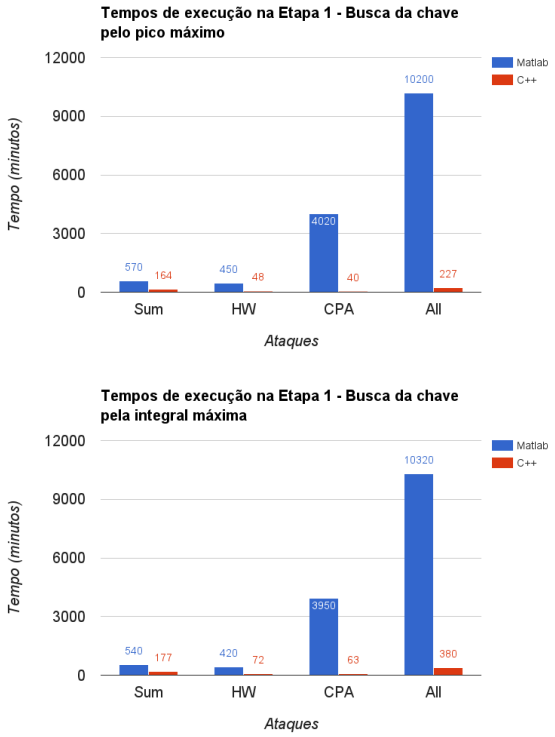


Figura 5. Comparação dos tempos de execução entre os algoritmos em C++ e em Matlab na Etapa 1.

5.2. Planificação de Matrizes

Analisando a execução do algoritmo observa-se que a parte de maior custo computacional é a exigência por percorrer as matrizes tridimensionais iterando por meio das colunas das matrizes e não em sua profundidade, ou seja, há um laço de iteração `for i`, seguido de um `for k` e um `for j`. O principal problema ocorre pelo fato do termo que está iterando ser o termo das colunas, ou seja, se tal vetor da matriz está localizado em uma posição de memória não carregada na memória cache ocorre um `cache mis` gerando um tempo de atualização da cache. Esse processo pode se repetir inúmeras vezes comprometendo o desempenho da execução do algoritmo. Para amenizar o custo da busca de tais vetores em memória propõe-se a planificação destas matrizes tridimensionais para vetores unidimensionais como mostra na Figura 6.

Nesta matriz pode-se notar que seus índices foram remapeados para um vetor que possui tamanho `numKeyHypothesis * numPontos * numSbox`. Os índices `kps`, `kPS` e `KPS`, onde k é a linha, p são as colunas e s é a profundidade, indicam respectivamente as posições `[0][0][0]`, `[0][numPontos][0]` e `[numKeyHypothesis][numPontos][numSbox]` estão destacados no vetor para indicar as posições em que foram remapeados. Para realizar o controle dos índices desta matriz tridimensional, dentro de um laço de iteração for i, j e k, foi utilizada a equação `[i * (largura * profundidade) + j * (profundidade) + k]`. Com esta equação foi mantida a coerência dos índices de cada dado da matriz original.

A Figura 7 mostra que o tempo de execução da busca pelo pico máximo não

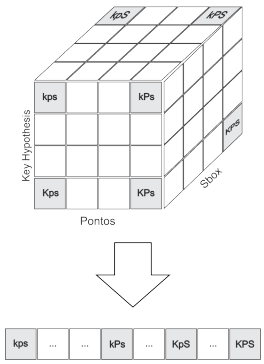


Figura 6. Planificação de matriz tridimensional em um vetor.

foi reduzido apenas no CPA. O Sum, ataque com maior consumo de memória dentre os avaliados, teve uma redução em torno de 39 minutos em seu tempo de execução. O HW teve uma redução de aproximadamente 10% (≈5 min) em relação a etapa anterior. Ao utilizar todos os ataques, o tempo de execução foi reduzido em torno de 25 minutos. Na busca pela integral máxima o comportamento se manteve, apesar de o ganho no Sum ter sido um pouco menor em relação a busca pelo pico máximo. Os demais resultados mantiveram a proporção de seus ganhos, onde o ALL teve uma redução em torno de 30 minutos em sua execução. Pode-se observar que o CPA foi o único ataque que não obteve ganho de desempenho nesta etapa, isso se deve pelo fato de este ataque não possuir matrizes intermediárias como os demais.

5.3. Diretivas SIMD

Após a planificação das matrizes, foi utilizado o OpenMP para fazer a implementação de diretivas SIMD neste algoritmo. Estas diretivas informam o compilador que determinado laço pode ser executado utilizando instruções do tipo SIMD (*Single Instruction, Multiple Data*). Estas instruções fazem com que diversas computações sejam executadas paralelamente, porém somente uma instrução por vez. Este tipo de diretiva também é conhecido como paralelismo de dados.

Em cada laço que percorre as matrizes intermediárias foram implementadas as diretivas `#pragma omp simd`. Entretanto, os resultados nesta etapa foram os que obtiveram menor ganho de desempenho em relação as demais etapas de otimização propostas, conforme mostrado na Tabela 1.

Em ambos os métodos de busca se observou um pequeno ganho de desempenho nos mesmos ataques da etapa anterior: Sum, HW e ALL. Novamente o CPA manteve o desempenho da etapa anterior por não ter matrizes intermediárias. Na Tabela 1 observam-se os ganhos de desempenho obtidos nesta etapa foram pequenos.

Tabela 1. Speedups da vetorização do código na Etapa 3.

	Sum	HW	CPA	ALL
maxPeak	1.03	1.04	1	1.03
maxIntegral	1.02	1.03	1	1.02

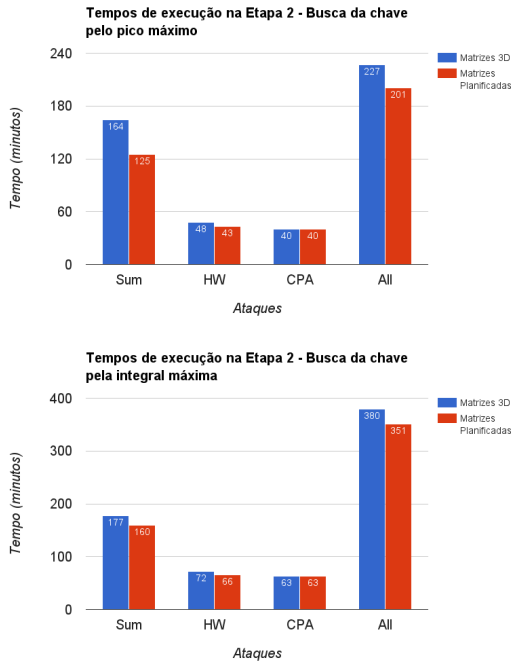


Figura 7. Comparação dos tempos de execução entre o algoritmo com as matrizes tridimensionais e as matrizes planificadas na Etapa 2.

5.4. Paralelismo com OpenMP

Na quarta e última etapa de otimização proposta é implementado o paralelismo utilizando OpenMP. Considerando que DPA possui matrizes globais que são incrementadas a cada vez que uma hipótese de chave é percorrida, não é possível aplicar um paralelismo em níveis mais externos do aninhamento dos laços `for` neste algoritmo. Desta forma a estratégia utilizada é paralelizar o laço das hipóteses de chave, considerando o grande número de dependência de dados existente no algoritmo.

As estruturas utilizadas para paralelizar o laço das hipóteses de chave são as diretivas `#pragma omp parallel for`, e também é especificado com a diretiva `firstprivate`, onde cada `thread` disparada pelo `parallel for` terá sua própria instância do vetor que possui o conteúdo do traço lido de entrada e seu próprio valor intermediário. Ao usar uma `thread` observa-se que em todos os ataques há um aumento em seu tempo de execução, isso demonstra que o custo da criação e sincronização das `threads` em OpenMP apresentam um custo significativo que não deve ser ignorado.

Diferentemente da etapa anterior, o paralelismo trouxe bons ganhos de desempenho na execução do algoritmo. Os tempos estão demonstrados na Figura 8 onde observa-se que o número de `threads` 0 representa o tempo de execução do algoritmo com otimizações na Etapa 3, os números de `threads` seguintes são os números de `threads` utilizados com o OpenMP.

A busca pela integral máxima possui o mesmo comportamento que a busca pelo pico máximo, porém somente em Sum e ALL houve perdas de desempenho ao utilizar as 4 threads. Novamente observou-se que CPA teve seu desempenho elevado e o HW também teve um pequeno ganho de desempenho ao utilizar 4 `threads`. Na Tabela 3 são

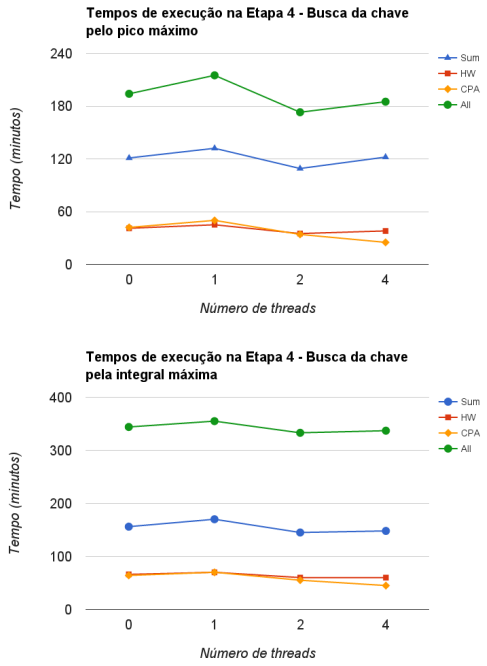


Figura 8. Comparação dos tempos de execução do algoritmo com as diretivas SIMD e do algoritmo paralelo na Etapa 4.

Tabela 2. Speedups do algoritmo paralelo na Etapa 4 com busca pela chave máxima e integral máxima, respectivamente.

	Sum	HW	CPA	ALL	Sum	HW	CPA	ALL
1 Thread	0,9	0,9	0,8	0,9	0,9	0,9	0,9	0,9
2 Threads	1,1	1,3	1,2	1,1	1,1	1,3	1,2	1,03
4 Threads	0,99	1,1	1,4	1,02	1,05	1,1	1,4	1,02

representados os speedups calculados para cada `thread` nesta última etapa de otimização do algoritmo em relação aos resultados obtidos na Etapa 3.

6. Conclusão

Neste trabalho são propostas e avaliadas quatro propostas de otimização para uma implementação do algoritmo de Análise Diferencial de Potência (DPA) desenvolvido em [Lomné et al. 2009]. O trabalho inicialmente revisa as principais motivações para sua execução bem como algumas ferramentas para programação paralela que podem explorar o paralelismo de execução do algoritmo DPA. Além disso, a revisão mostra que por lidar com problemas de natureza aninhada, muitas diretrizes disponíveis ao programador e um alto nível de abstração levam a escolha pela interface OpenMP.

Os experimentos avaliam a execução do algoritmo para cada nível de otimização proposto. Os resultados demonstram significativos ganhos em desempenho, principalmente no ataque CPA, onde houve uma redução de aproximadamente 160 vezes em seu tempo de execução, usando a busca pelo pico máximo. Além disso, os experimentos destacam relevantes ganhos de desempenho na ordem de 5 vezes para ataques realizados com

SUM e 13 vezes para ataques usando o parâmetro HW. É preciso enfatizar, que a maior parte desse ganho, entre todas as fazes de otimização, provém da tradução do algoritmo de Matlab para C++.

Tabela 3. Tempos de execução finais em minutos, com busca pelo pico máximo e integral máxima, respectivamente.

	Sum	HW	CPA	ALL	Sum	HW	CPA	ALL
Matlab	570	450	4.020	10.200	540	420	3.950	10.320
C++	109	35	25	173	145	60	45	333

Com as implementações otimizadas, é possível executar um ataque DPA em um número maior de traços, de forma a obter-se resultados em um intervalo menor de tempo. Isto permite explorar vários cenários de ataques em um tempo viável, avaliando ainda mais a fuga de informações por este canal lateral, reduzindo o tempo de espera pelos resultados, que pode ser na ordem de dias e até mesmo semanas para o fim de sua execução.

Agradecimentos

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

Referências

Bartkewitz, T. and Lemke-Rust, K. (2011). A High-Performance Implementation of Differential Power Analysis on Graphics Cards. In *CARDIS*, pages 252–265. Springer.

Brier, E., Clavier, C., and Olivier, F. (2004). Correlation Power Analysis with a Leakage Model. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 16–29. Springer.

Gamaarachchi, H., Ragel, R., and Jayasinghe, D. (2014). Accelerating Correlation Power Analysis using Graphics Processing Units (GPUs). In *Information and Automation for Sustainability (ICIAfS), 2014 7th International Conference on*, pages 1–6. IEEE.

Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. In *Advances in Cryptology—CRYPTO'99*, pages 388–397. Springer.

Kocher, P. C. (1996). Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, other Systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer.

Lomné, V., Maurine, P., Torres, L., Robert, M., Soares, R., and Calazans, N. (2009). Evaluation on FPGA of Triple Rail Logic Robustness against DPA and DEMA. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 634–639. European Design and Automation Association.

Stallings, W. (2008). *Criptografia e Segurança de Redes*. Pearson Prentice Hall.

Swamy, T., Shah, N., Luo, P., Fei, Y., and Kaeli, D. (2014). Scalable and Efficient Implementation of Correlation Power Analysis using Graphics Processing Units (GPUs). In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, page 10. ACM.

Waddle, J. and Wagner, D. (2004). Towards Efficient Second-Order Power Analysis. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 1–15. Springer.

Modelo de Recuperação Arquitetural QoE-QoS Híbrido para Bases de Dados Distribuídas Gerenciado por Middleware

Ramon Hugo de Souza, Mario Antonio Ribeiro Dantas

Programa de Pós-Graduação em Ciência da Computação (PPGCC)
Universidade Federal de Santa Catarina (UFSC)
Florianópolis - SC Brazil

ramon@securebay.com, mario.dantas@ufsc.br

Resumo. *O conceito de qualidade de experiência (QoE) não é uma novidade per se, mas sua definição em termos de métricas mapeáveis para qualidade de serviço (QoS), e em especial tal abordagem aplicada ao contexto de bases de dados distribuídas (BDDs), é um novo conceito. Novos trabalhos formalizando QoE como completude de QoS permitem se lidar com decisões diretamente relacionadas a correção de desempenho em nível de sistema, não mais considerando-se o conceito abstrato de QoE, usualmente ligado a satisfação de usuário com base em avaliações tácitas. Esta pesquisa tem como objetivo a demonstração do ganho temporal que pode ser obtido em considerações de replicação com gerenciamento via a abordagem de um middleware.*

1. Introdução

O conceito de *qualidade de serviço* (*QoS*) é bem fundamentado na área de *redes*, origem de suas definições em *ciência da computação*. Sendo o mesmo apresentado em [Wolter and van Moorsel 2001] como o conjunto de métricas de vazão, atraso e disponibilidade, é um conceito apresentado de maneira mais rígida em [Shenker et al. 1997] com a definição utilizada para *serviço garantido*, a qual firma que o mesmo "provê limites firmes e matematicamente prováveis de atraso fim-a-fim em enfileiramento de datagramas".

A definição conceitual de *QoS* em termos de *métricas matematicamente prováveis*, a qual pôde-se abstrair além da fundamentação de datagramas de redes, foi o passo fundamental que permitiu a aplicação de tal conceito a abstração de *bases de dados distribuídas* (*BDDs*) nos primeiros trabalhos sobre o assunto, publicados em meados dos anos 2010.

Além de tal, tem-se também a definição do conceito de *qualidade de serviço* aparecendo em diversos artigos publicados junto a área de *marketing* [Lewis and Booms 1983, Gronroos 1984, Parasuraman et al. 1985, Parasuraman et al. 1988, Parasuraman et al. 1994], onde o mesmo pode ser concisamente definido como "o resultado da comparação que clientes realizam entre suas expectativas sobre o serviço e sua percepção sobre a maneira como o serviço foi realizado" [Caruana 2002]. Sendo esta na verdade a conceitualização utilizada para *qualidade de experiência* (*QoE*), e não *QoS*, em *ciência da computação*, com suas origens provavelmente advindas destes conceitos de *marketing* qualificando a experiência do usuário sobre um serviço consumido.

Esta conexão com tais conceitos de *marketing* pode ser visualizada na definição dada em [ISO9241-210 2010], onde é definido que a *experiência do usuário* envolve "as

percepções de uma pessoa e suas respostas resultantes do uso, ou antecipação do uso, de um produto, sistema ou serviço".

Na explicação de fundo teórico da conexão entre *QoE* e *conceitos de QoS* dada em [ur Rehman Laghari et al. 2011], tem-se que *QoE* pode ser considerada como um mapeamento direto, porém ainda *difuso*, sobre como um usuário percebe um serviço em termos de *parâmetros de QoS*. Sendo citado como digno de nota em [de Souza and Dantas 2015] que "uma vez que os *parâmetros de QoE* são baseados em comportamento e expectativas humanas, é difícil se assumir o nível de precisão neste processo de mapeamento. Um problema que desaparece quando se considerando sistemas ao invés de usuários".

E então tem-se uma primeira definição que permite se lidar com avaliações de *QoE* em sistemas de *BDDs* quando se definindo métricas palpáveis de *QoS* em termos de médias e desvios padrão - considerado-se a combinação destes parâmetros como satisfatória para definição de métricas matematicamente prováveis sobre garantias dadas.

Na seção 2 apresenta-se: (i) a origem dos conceitos de *QoE* aplicados a *BDDs*; (ii) um primeiro modelo lidando com avaliação de base estritamente temporal; e (iii) um segundo modelo lidando com avaliação diretamente sobre completude de recursos. Na seção 3 é apresentado o novo modelo com gerenciamento de recursos utilizando um *middleware*, com os resultados de simulação demonstrados na seção 4. Finaliza-se com as conclusões e trabalhos futuros junto a seção 5.

2. QoE em BDDs e Modelo Arquitetural de Recuperação

2.1. QoE em Bases de Dados Distribuídas

No trabalho de levantamento bibliométrico [de Souza and Dantas 2015] é discutida a ausência de definições precisas de *qualidade* para *BDDs* num período de levantamento realizado no ano de 2014 e ainda válido em 2017. Neste trabalho é pontuado que conceitos em literatura focada em redes possuem natureza por demais holística, o que levou tal a lidar com definições conceituais para demonstração da importância de *QoE* para *BDDs*.

Sendo também demonstrado que uma melhor definição de *QoS* em termos de médias e desvios padrão de crucial importância para quando se lidando com garantias de *QoE*, seja para *BDDs* ou mesmo tecnologias relacionadas a redes. Formalizando assim métricas matematicamente prováveis utilizadas em trabalhos subsequentes.

2.2. Modelo Arquitetural de Recuperação Focado em RAS

Em [de Souza et al. 2016] é apresentado um primeiro modelo de recuperação baseado na chamada *capacidade RAS*[1] *(confiabilidade/disponibilidade/facilidade de manutenção do inglês reliability/availability/serviceability)*: com restrições de confiabilidade, disponibilidade e facilidade de manutenção - incluindo restrições de tolerância a falhas.

Sendo este modelo completamente baseado em verificação de restrições temporais, gerando o empecilho de se mapear medidas de recursos para seu efeito temporal.

[1]RAS: Um termo originalmente utilizado pela IBM para descrever a robustez de seus computadores *mainframe* [Siewiorek and Swarz 1998].

2.3. Modelo Arquitetural Estendido

Em [de Souza and Dantas 2017] é apresentada uma primeira solução para se evitar o mapeamento de recursos para sua representação temporal, sendo um primeiro modelo baseado somente em avaliação de recursos provistos.

Em tal artigo é apresentado um modelo lidando com dois anéis completos e tolerância a uma falta na comunicação entre anéis e na comunicação entre os nós replicantes no que é chamado de *pool provedor de serviços*. É apresentado também um *pool de backup* com um semi-anel de réplicas dormentes.

No algoritmo 1 é apresentado o procedimento principal de tal modelo, que é utilizado em explicação comparativa na próxima seção sobre o porquê da mudança em processo decisório.

Algoritmo 1 Procedimento principal do algoritmo de gerenciamento de recursos em sistemas distribuídos

1: **procedimento** $Principal(tempo, tipo, passo)$
2: **enquanto** $verdade$ **faça**
3: **se** $\frac{\sum_{i=1}^{\#replicas} replicas[i].c_{livre}}{\#replicas} \leq ANS^{min}_{RASGarantia}$ **então**
4: call $RAS_{min}(tempo, tipo, passo)$
5: **senão se** $c - 2 * ANS^{min}_{RASGarantia} \geq \frac{\sum_{i=1}^{\#replicas} replicas[i].c_{usado}}{\#replicas}$ **então**
6: call $RAS_{max}(tempo, tipo, passo)$
7: **fim se**
8: $espere(tempo)$
9: **fim enquanto**
10: **fim procedimento**

Este é um procedimento que checa constantemente se o valor mínimo de recursos definido em *Acordo de Nível de Serviço* (*ANS*), $ANS^{min}_{RASGuarantee}$, é garantido, chamando o procedimento de correção $RAS_{min}(tempo, tipo, passo)$ quando tal valor é alcançado. Sendo este procedimento de correção responsável por iniciar mecanismos que visam prover mais recursos.

O procedimento principal checa também se é possível se reduzir a capacidade de recursos provistos, chamando o procedimento responsável $RAS_{max}(tempo, tipo, passo)$ caso a redução de capacidade não apresente riscos ao sistema em relação a garantia dada sobre o valor definido em ANS.

Tal algoritmo trabalha com quatro modos de operação: (i) otimista: baseado no algoritmo apresentado em [de Souza et al. 2016], mas com mensuramento de recursos e não avaliações temporais; (ii) não-otimista: uma versão em contrapartida ao modo otimista; (iii) balanceado: uma tentativa de balanceamento entre os dois primeiros modos; e (iv) não-otimista otimizado: a evolução do modo não-otimista focada em economia de recursos.

Note, na linha 8, que este algoritmo tem como consideração um valor de tempo de espera como intervalo de verificação de se os recursos podem ser modificados.

3. Modelo Arquitetural Estendido Gerenciado por Middleware

O maior empecilho encontrado no algoritmo do modelo arquitetural estendido, apresentado na seção anterior, é a necessidade de verificação temporal. Tal verificação não é pontual, e o sistema lidando com passos discretos pode passar por estados indesejados desnecessariamente. Esta verificação temporal é legado do algoritmo apresentado em [de Souza et al. 2016], o qual tem por base avaliação somente temporal.

Na figura apresentada em tal artigo percebe-se uma pequena falha de consideração na representação gráfica da arquitetura quando se verificando o procedimento no algoritmo 1. De maneira que quem realmente deve realizar as chamadas de incremento ou decremento de recursos é o *middleware*, tanto no caso vertical quanto no horizontal.

E então o *stress* de replicação aplicado sobre os nós $n04$ ou $n07$ - a figura 1 apresenta um modelo com as mesmas especificações - deixa de existir. Note também que este *stress* de replicação não foi demonstrado na simulação apresentada em [de Souza and Dantas 2017].

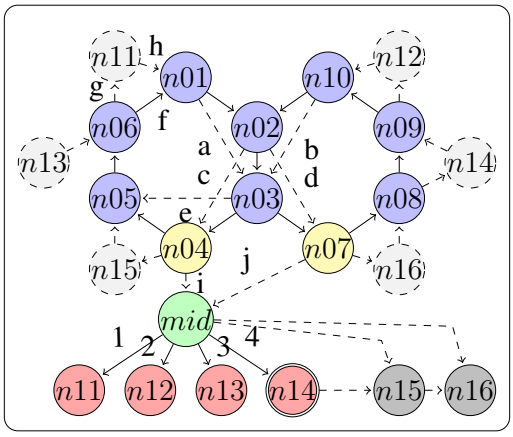


Figura 1. Modelo arquitetural estendido gerenciado por middleware com implementação de troca de mensagens utilizando o modelo multi-anel e tolerância a uma falta.

Para demonstração deste efeito de *stress* de replicação considera-se o consumo de uma unidade de tempo sobre o nó $n04$, responsável pela replicação, e sobre o nó a ser adicionado utilizando os quatro modos - nas figuras 3, 5, 7 e 9 com as mesmas considerações de simulação publicadas no referido artigo sem o *stress*. O efeito de *stress* de replicação é, em geral, muito maior que este simulado, mas para demonstração considerar-se-á que ocorre a sincronização de operações somente no momento de adição de nó. Note que este é o melhor cenário possível, ocupando apenas um tempo dos nós e somente quando um nó é adicionado ao *pool provedor de serviços*.

De maneira que a representação do modelo, evitando-se o *stress* de replicação sobre os *nós de borda*[2], deve ser gerenciada via *middleware*, como apresentado no modelo da figura 1.

[2]Nós referenciados em [de Souza and Dantas 2017] como os responsáveis pela gerência de replicação junto ao *pool de backup*.

Onde então os nós $n04$ e $n07$ não mais precisam armazenar uma sequência especial de operações a ser enviada as réplicas no *pool de backup*, somente enviando uma cópia das operações em seu *buffer* ao *middleware*, da mesma maneira como o nó $n04$ envia ao nó $n05$ em sua sequência de atualização padrão.

Note que neste modelo gerenciado por *middleware* a replicação deve ser *sequencial completa*, e não transação a transação, afinal deseja-se obter uma réplica funcional o mais rápido possível caso seja necessária adição ao *pool provedor de serviços*, mesmo durante a fase de replicação - a replicação poderia estar acontecendo junto ao nó $n12$ enquanto o nó $n11$ já estaria pronto para ser adicionado caso necessário.

Neste novo modelo tem-se também uma consideração especial quanto ao *último nó* parte do *pool de backup*. No exemplo apresentado na figura 1 tem-se que no momento em que o nó $n11$ é adicionado ao *pool provedor de serviços* o *middleware* acorda o nó $n14$ para se responsabilizar pela obtenção de dados necessária a instanciação do novo nó $n15$, a ser adicionado ao *pool de backup*. De maneira a explicitarmos que não é o nó $n14$ que se responsabiliza pela cópia, e sim o *middleware*.

4. Resultados Experimentais

Nesta seção apresenta-se os resultados relativos aos experimentos, visando demostrar o diferencial da proposta estendida.

Quanto aos modos de operação, tem-se inicialmente o modo otimista, que já fora demonstrado "problemático" em sua primeira utilização junto ao modelo de avaliação temporal. Na figura 2 pode-se verificar tal impacto sobre os valores de recursos provistos ainda desconsiderando-se o *stress* de replicação.

Note que este modo já apresenta o comportamento peculiar de lidar com picos abaixo da capacidade esperada, podendo-se observar o comportamento do mesmo com a consideração de *stress* de replicação junto a figura 3.

Quando lidando-se com o modo otimista fica claro que o efeito do *stress* de replicação é uma consideração que pode trazer sérias implicações quanto a utilização deste tipo de arquitetura, em um modo o qual já apresentava menos recursos que o necessário quando da replicação horizontal sem a consideração de tal efeito.

Os modos não-otimista, balanceado e não-otimista aprimorado apresentam comportamento estruturalmente parecido quando da replicação horizontal, mas mesmo não sofrendo como o modo otimista todos eles apresentam momentos em que a capacidade requerida não é provista pelo sistema a tempo. De maneira que fica claro que as considerações feitas no modelo anterior, sem considerar efeitos de manutenção quando não realizados estritamente pelo *middleware*, podem ser realmente problemáticas.

No comparativo entre as figuras 4, 6 e 8 com seus equivalentes com a inserção do efeito de *stress* de replicação, nas figuras 5, 7 e 9, respectivamente, nota-se que a consideração pontuada no artigo original pode ser *não realista*, de maneira que a existência do *stress* de replicação causa situações em que os recursos não são provistos como previsto em tal modelagem. Um problema que desaparece ao se considerar o *middleware* como o responsável por tal processo de replicação.

Os modos não-otimistas possuem uma queda menor mesmo em consideração de

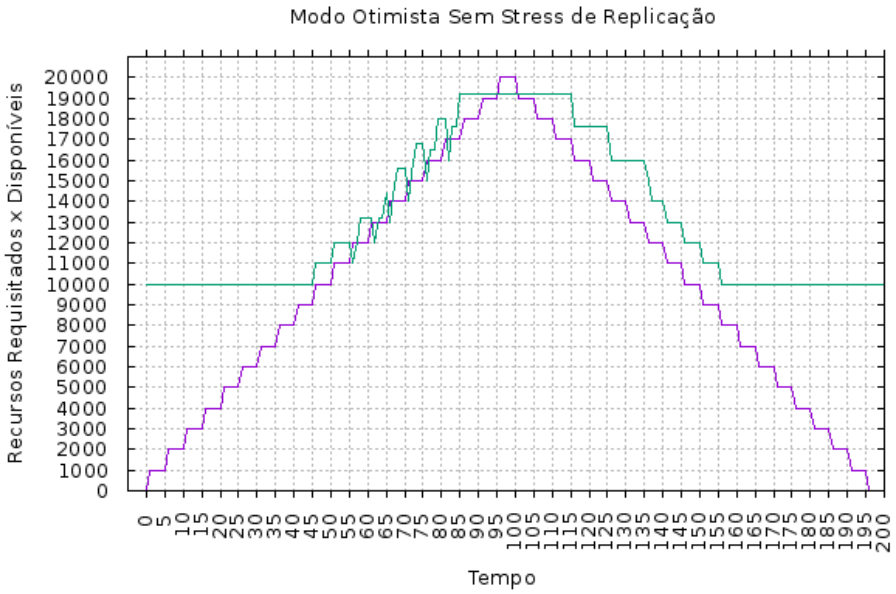


Figura 2. Modo otimista sem stress de replicação: Recursos requisitados (em roxo, iniciando em 0) contra disponibilizados (em ciano, iniciando em 10000).

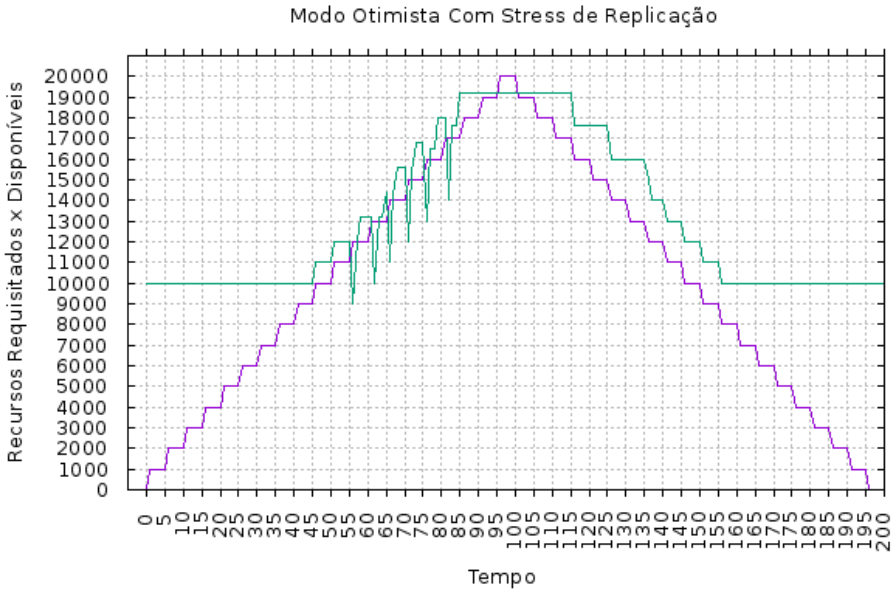


Figura 3. Modo otimista com stress de replicação: Recursos requisitados (em roxo, iniciando em 0) contra disponibilizados (em ciano, iniciando em 10000).

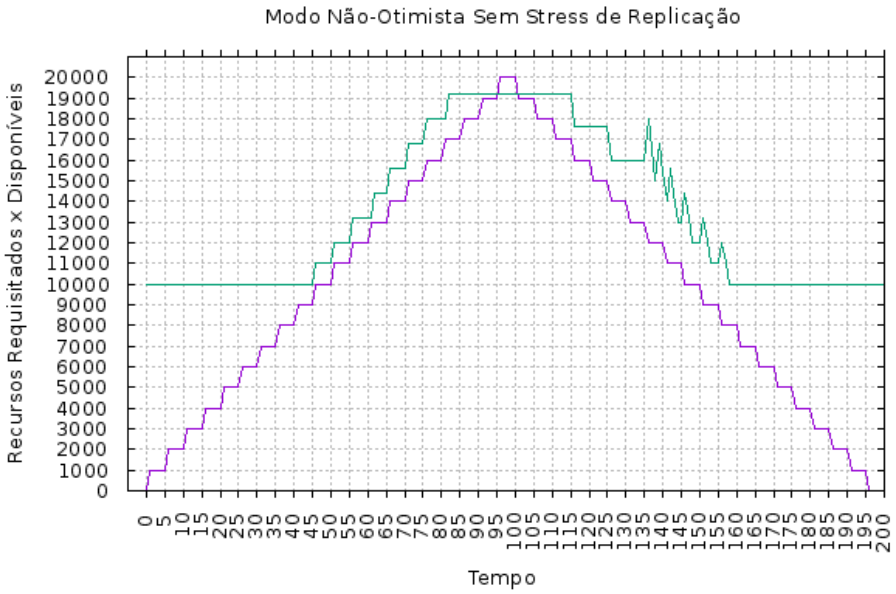


Figura 4. Modo não-otimista sem stress de replicação: Recursos requisitados (em roxo, iniciando em 0) contra disponibilizados (em ciano, iniciando em 10000).

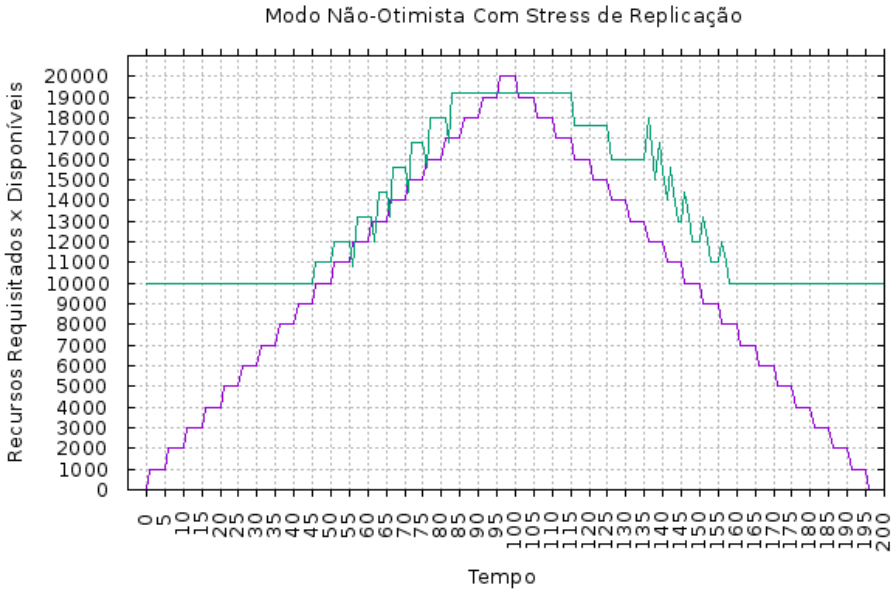


Figura 5. Modo não-otimista com stress de replicação: Recursos requisitados (em roxo, iniciando em 0) contra disponibilizados (em ciano, iniciando em 10000).

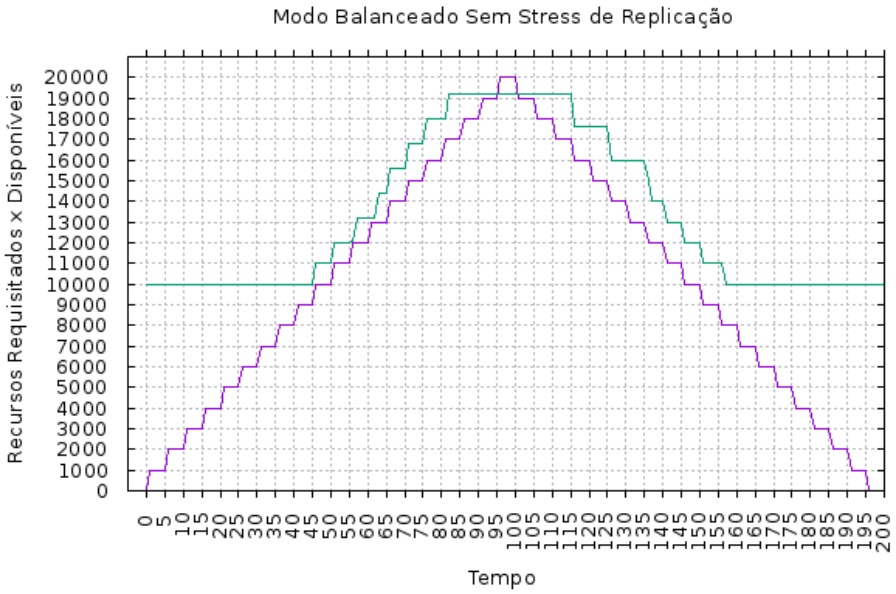


Figura 6. Modo balanceado sem stress de replicação: Recursos requisitados (em roxo, iniciando em 0) contra disponibilizados (em ciano, iniciando em 10000).

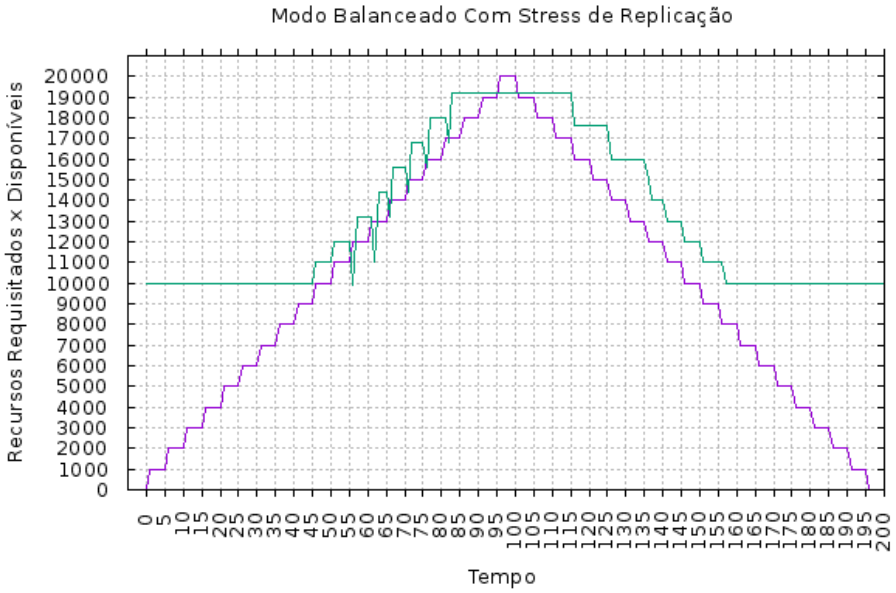


Figura 7. Modo balanceado com stress de replicação: Recursos requisitados (em roxo, iniciando em 0) contra disponibilizados (em ciano, iniciando em 10000).

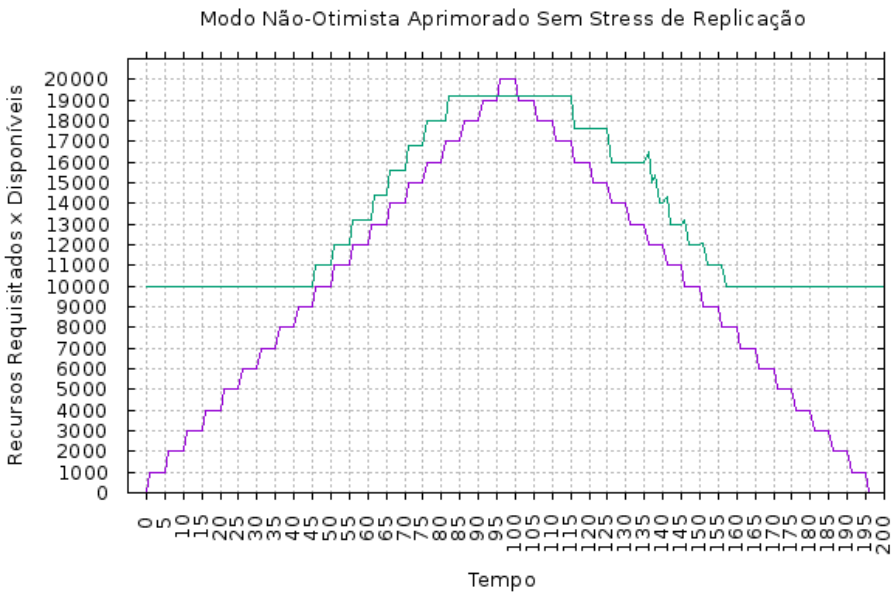


Figura 8. Modo não-otimista aprimorado sem stress de replicação: Recursos requisitados (em roxo, iniciando em 0) contra disponibilizados (em ciano, iniciando em 10000).

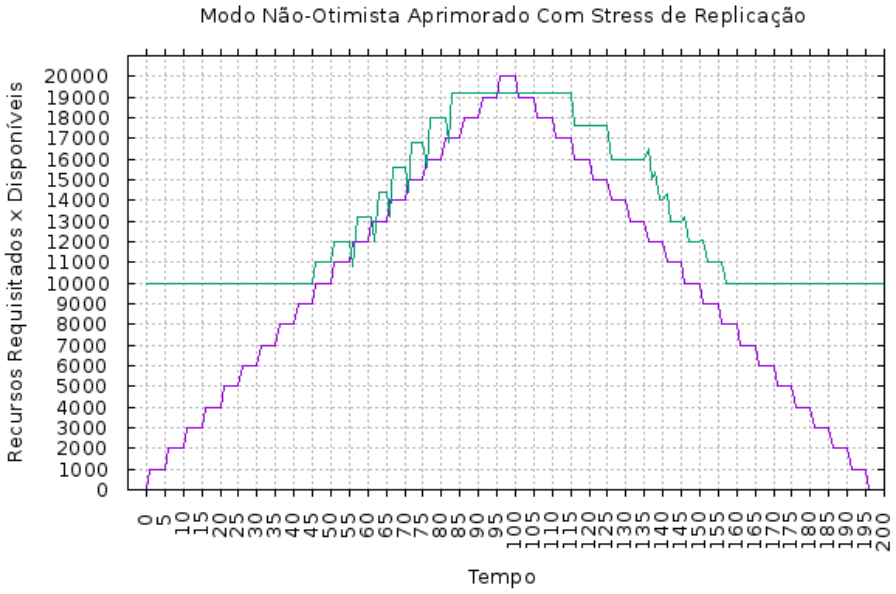


Figura 9. Modo não-otimista aprimorado com stress de replicação: Recursos requisitados (em roxo, iniciando em 0) contra disponibilizados (em ciano, iniciando em 10000).

stress de replicação, sendo o efeito sobre os dois modos de mesma magnitude. Tal efeito sobre o modo balanceado se encontra entre os modos não-otimistas e o modo otimista, sendo o modo otimista o mais afetado pelo efeito de *stress*, afinal o mesmo já possuía o problema de redução de recursos quando da replicação horizontal que dá nome ao modo.

E então apresenta-se o algoritmo 2 com as definições procedurais necessárias para que o *middleware* lide com este processo de replicação.

Tal algoritmo sendo somente um procedimento de adição de operações, a ser recebido pelo *middleware*, com o procedimento $Principal$, apresentado no algoritmo 1, sendo executado ao final do procedimento de adição de operações. Não existindo mais o a chamada a $espere(tempo)$ e sendo o intervalo de atualização agora o mesmo de atualização de mensagens entre nós.

Neste novo modelo os parâmetros são considerados globais, de maneira que os procedimentos RAS_{min} e RAS_{max} não necessitam mais de parâmetros em sua chamada. Para tais procedimentos agora considera-se também explicitamente a exclusão de seu loop de verificação com o intervalo temporal realizado pela chamada a $espere(tempo)$, excluindo-se completamente as considerações temporais e ainda assim mantendo válidos os valores simulados, afinal o tempo de espera simulado é justamente o intervalo de atualização de mensagens.

No algoritmo são explicitadas as operações *aloca(réplica)*, linha 19, e *desaloca(réplica)*, linha 26, para explicitar a existência de momentos em que o nó *réplica* está indisponível, origem da definição de tarefa de replicação a cargo do *middleware*.

5. Conclusões e Trabalhos Futuros

Neste trabalho de pesquisa foram apresentados resultados de um modelo diferenciado, o qual, diferente de uma proposta anterior que desconsiderava o consumo de tempo quando da realização de operação de replicação, leva em conta a consideração temporal de *stress* de replicação, a qual é necessária para uma melhor representação deste tipo de modelo quando aplicado a *BDDs*.

Com base em tais resultados comparativos entre os modelos com e sem consideração de consumo de tempo de replicação, apresentados com efeitos brandos sobre as possibilidades que tais considerações poderiam causar, concluiu-se que delegar o controle de replicação a um *middleware* lidando com o *Sistema de Gerenciamento de Banco de Dados* (*SGBD*) é o caminho para se evitar sobrecarga de recurso sobre nós replicantes.

Nesta nova representação tem-se agora um modelo discreto, baseado em passos de comunicação entre os nós da arquitetura. Em uma evolução, após a avaliação migrada da abordagem temporal para recursos, este modelo agora não lida mais com *threads* em *loop* com intervalos de verificação, sendo desta maneira um modelo completamente discreto.

Na atual abordagem, o *middleware* tem comportamento similar a um nó comunicante, tornando-se mais simples as verificações necessárias, além de evitar sobrecarga de processamento e armazenamento de fila de operações sobre os *nós de borda*.

Os dados de simulação apresentados demonstram como casos em que os recursos necessários não poderiam ser providos instantaneamente com a consideração real, envolvendo *stress* de replicação, poderiam ser resolvidos ao se utilizar um *middleware* anexo

ao *SGBD* responsabilizado pelo processo de replicação.

Como trabalho futuro espera-se formalizar completamente o modelo de replicação baseado em *nós virtuais*.

Algoritmo 2 Gerenciamento de Operações no Pool de Backup via Middleware

1: $opTempo[inserção|update|deleção] = \{tempo1|tempo2|tempo3\}$
2: $tempo \leftarrow 0$
3: $lista \leftarrow$ nova lista de operações vazia
4: $limite \leftarrow$ tempo limite para atingir cópia completa
5: $tipo \leftarrow$ modo do algoritmo
6: $passo \leftarrow$ passo de incremento do sistema
7: **procedimento** $AdicionaOperação(opNovas)$
8: $i \leftarrow 0$
9: **enquanto** $i < opNovas.tamanho$ **faça**
10: $opNova \leftarrow opNovas.operação(i)$
11: $tempo = tempo + opTempo[opNova.tipo]$
12: $lista.adiciona(opNova)$
13: $i \leftarrow i + 1$
14: **fim enquanto**
15: **se** $tempo \geq limite$ **então**
16: $i \leftarrow 0$
17: **enquanto** $i < réplicas.tamanho$ **faça**
18: $réplica \leftarrow réplicas.acorda(i)$
19: $aloca(réplica)$
20: $j \leftarrow 0$
21: **enquanto** $j < lista.tamanho$ **faça**
22: $op \leftarrow lista.operação(j)$
23: $replica.adicionaOp(op)$
24: $j \leftarrow j + 1$
25: **fim enquanto**
26: $desaloca(réplica)$
27: $i \leftarrow i + 1$
28: **fim enquanto**
29: $tempo \leftarrow 0$
30: $lista \leftarrow$ nova lista vazia
31: **fim se**
32: **se** $\frac{\sum_{i=1}^{\#replicas} replicas[i].c_{livre}}{\#replicas} \leq ANS^{min}_{RASGuarantee}$ **então**
33: chame $RAS_{min}()$
34: **senão se** $c - 2 * ANS^{min}_{RASGuarantee} \geq \frac{\sum_{i=1}^{\#replicas} replicas[i].c_{usado}}{\#replicas}$ **então**
35: chame $RAS_{max}()$
36: **fim se**
37: **fim procedimento**

Referências

Caruana, A. (2002). Service loyalty: The effects of service quality and the mediating role of customer satisfaction. *European Journal of Marketing*, 36(7/8):811–828.

de Souza, R. H. and Dantas, M. A. R. (2015). Mapping QoE through QoS in an approach to DDB architectures: Research analysis and conceptualization. *ACM Computing Surveys (CSUR)*, 48(2).

de Souza, R. H. and Dantas, M. A. R. (2017). An Architectural Recovering Model for Distributed Databases Focused on Resources Availability. Submitted to Distributed and Parallel Databases (DAPD).

de Souza, R. H., Flores, P. A., Dantas, M. A. R., and Siqueira, F. (2016). Architectural recovering model for distributed databases: A reliability, availability and serviceability approach. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 575–580, Messina, Italy.

Gronroos, C. (1984). A service quality model and its market implications. *European Journal of Marketing*, 18(4):36–44.

ISO9241-210 (2010). Iso 9241-210:2010 - human-centred design processes for interactive systems. Available at http://www.iso.org/iso/catalogue_detail.htm?csnumber=21197.

Lewis, R. C. and Booms, B. H. (1983). The marketing aspects of service quality. In Berry, L. L., Shostack, G., and Upah, G., editors, *Emerging Perspectives in Service Marketing*, pages 99–107, Chicago, IL. USA. American Marketing Association.

Parasuraman, A., Zeithaml, V. A., and Bery, L. L. (1985). A conceptual model of service quality and its implication for future research. *Journal of Marketing*, 49:41–50.

Parasuraman, A., Zeithaml, V. A., and Bery, L. L. (1988). Servqual: a multiple-item scale for measuring consumer perceptions of service quality. *Journal of Retailing*, 64(1):12–40.

Parasuraman, A., Zeithaml, V. A., and Bery, L. L. (1994). Alternative scales for measuring service quality: a comparative assessment based on psychometric and diagnostic criteria. *Journal of Retailing*, 70(3):201–30.

Shenker, S., Partridge, C., and Guerin, R. (1997). Rfc 2212: Specification of guaranteed quality of service. Technical report, Internet Engineering Task Force (IETF).

Siewiorek, D. P. and Swarz, R. S. (1998). *Reliable computer systems: design and evaluation*. Natick, Mass. : A K Peters, 3th edition. p. 508.

ur Rehman Laghari, K., Crespi, N., Molina, B., and Palau, C. (2011). Qoe aware service delivery in distributed environment. In *Advanced Information Networking and Applications (AINA), 2011 IEEE Workshops of International Conference on*, pages 837 – 842. Available at http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5763609.

Wolter, K. and van Moorsel, A. (2001). The relationship between quality of service and business metrics: Monitoring, notification and optimization. Technical Report 96, HP Laboratories Technical Report.

A Hybrid CPU-GPU-MIC Algorithm for the Hitting Set Problem

Danilo Carastan-Santos[1], **David C. Martins-Jr**[1], **Luiz C. S. Rozante**[1]
Siang W. Song[2], **Raphael Y. de Camargo**[1]

[1]Centro de Matemática, Computação e Cognição
Universidade Federal do ABC (UFABC)
Santo André, Brazil

[2]Instituto de Matemática e Estatística
Universidade de São Paulo (USP)
São Paulo, Brazil

{danilo.santos, david.martins, luiz.rozante}@ufabc.edu.br

song@ime.usp.br, raphael.camargo@ufabc.edu.br

Abstract. *We present a hybrid exact algorithm for the Hitting Set Problem (HSP) for highly heterogeneous CPU-GPU-MIC platforms. With several techniques that permit an efficient exploitation of each architecture, low communication cost and effective load balancing, we were able to solve large HSP instances in reasonable time, achieving good performance and scalability. We obtained speedups of up to 25.32 in comparison with using two six-core CPUs and exact HSP solutions for instances with tens of thousands of variables in less than 5 hours. These results reinforce the statement that heterogeneous clusters of CPUs, GPUs and MICs can be used efficiently for high-performance computing.*

1. Introduction

Many important theoretical or practical problems can be modeled, in part or as a whole, as an instance of the Hitting Set problem (HSP) whose main objective, informally, is to find a minimum set of variables that satisfies every element in a finite set of constraints. This satisfiability reasoning is present, for example, in problems from Systems Biology, such as genomic reversal distance [Kolman and Walen 2007], classification models [Hvidsten et al. 2003], polymerase chain reaction experiments [Pearson et al. 1996] and gene regulatory networks (GRN) inference [Ideker et al. 2000, Ruchkys and Song 2003]. A common characteristic of these problems is that the input size of the modeled HSP instances is often large, making the retrieval of the exact solutions impracticable for traditional algorithms.

Since the Hitting Set problem is NP-hard [Garey and Johnson 1999], there exist some solutions that try to circumvent this problem, either by imposing restriction on the problem, such as with non-polynomial exact algorithms [Shi and Cai 2010], or by using heuristics and approximations [Cendic 2014, Hochbaum 1997]. Most recent works in HSP algorithms focus only on multithread parallelism, with different approaches to tackle HSP such as hypergraphs [Murakami and Uno 2014] and MapReduce [Cardoso and Abreu 2013]. Gainer-

Dewar and Vera-Licona [Gainer-Dewar and Vera-Licona 2017] present an extensive review of recent HSP algorithms. Due to the absence of accelerator-aided HSP algorithms, we have previously proposed a highly parallel and efficient algorithm for GPU (Graphics Processing Unit) [Owens et al. 2008] clusters that finds exact solutions for HSP instances with thousands of variables [Carastan-Santos et al. 2015, Carastan-Santos et al. 2017]. In these previous works, however, we explored only a single architecture for finding exact HSP solutions.

With the advent of new accelerator technologies such as Many Integrated Core (MIC) architectures [Duran and Klemm 2012], hybrid heterogeneous platforms composed of CPUs, GPUs and MICs became a common occurrence in research centers. In order to fully exploit the computer power of these novel hybrid platforms, there is a growing effort to develop hybrid applications, that is, applications capable of using CPU, GPU and MIC processors together [Andrade et al. 2014, Wolfe et al. 2014, Reza et al. 2015, Sîrbu and Babaoglu 2016]. However, this growing effort is still small mostly due to two major hindrances: (i) these heterogeneous platforms demand hybrid algorithms to efficiently exploit all co-processor architectures in conjunction and (ii) the efficiency of the usage of CPU-GPU-MIC platforms in a wide range of applications is not clearly known. In this regard, in this paper we explore ways to circumvent these hindrances in the scope of the Hitting Set Problem, and thus we present the following contributions:

- We propose a hybrid exact Hitting Set algorithm for CPU-GPU-MIC heterogeneous platforms, which efficiently exploits the advantages of each individual architecture on the heterogeneous cluster, efficiently minimizes communication among nodes and properly balances the load among the processors;
- We report a performance evaluation of the heterogeneous CPU-GPU-MIC cluster usage with our hybrid HSP implementation. The experimental results show that if (i) the load balancing is properly set, (ii) each individual architecture advantage is exploited and (iii) the node to node communication cost is minimized, we can effectively use heterogeneous CPU-GPU-MIC clusters, which allow us to solve unprecedentedly large HSP instances with tens of thousands of variables in reasonable time.

The remaining parts of this manuscript are organized as follows: Section 2 presents a formal definition of HSP, followed by our exact hybrid algorithm design, including the load balancing procedure. In Section 3 our proposed MIC HSP implementation is presented. The experimental results showing the performance evaluation are presented and discussed in Section 4 and in Section 5 we present the main conclusions.

2. Hybrid CPU-GPU-MIC Hitting Set Algorithm

Formally, the Hitting Set Problem (HSP for short) can be defined as follows. Given a finite set X, a collection $\mathcal{S}$ of subsets of X, which we call a collection of clauses, and a positive integer k, the goal is to find a subset $H \subseteq X$ with the smallest cardinality such that:

$$|H| \leq k \text{ and } H \cap S \neq \emptyset, \forall S \in \mathcal{S}. \quad (1)$$

More than one subset of X may satisfy the conditions above. We call $\mathcal{H} = \{H_1, H_2, ..., H_{|\mathcal{H}|}\}$ the collection of possible solutions. In this work our goal is to obtain all possible solutions $H \in \mathcal{H}$.

2.1. Exact Hybrid HSP Algorithm

We adopted an enhanced exhaustive search algorithm [Carastan-Santos et al. 2017] that enumerates and evaluates all candidate solutions – which are encoded as combinations of variables of X – in increasing order of cardinality $i, 1 \leq i \leq k$ and stops when a solution is found. The evaluation process is a disjunction check with the clauses in a sorted collection SortS, which is the collection $\mathcal{S}$ whose clauses are sorted in increasing order of its sizes. Evaluating the candidate solution with SortS allows an efficient discarding of non solution candidates.

We define a heterogeneous CPU-GPU-MIC platform as a set of c machines connected by the network, in which the jth machine has g_j processing units (PUs), which can be CPUs, GPUs or MICs. We adopted a master-slave approach, where the master process is responsible for assigning work to the slave processes and each slave process is responsible for demanding work to be processed in its respective PU. Algorithm 1 shows the pseudo-code for the master process and Algorithm 2 shows the pseudo-code for the slave process. In such a setting, the total number of processes is then $np = (\sum_{j=1}^{c} g_j) + 1$. Each PU architecture has its own processing module, represented by the function `EvalSolutionsOnPU()` in Algorithm 2, which is an architecture specific implementation of the code that can be called by the slave processes as a kernel, offload or function call for GPU, MIC and CPU, respectively. For CPUs and GPUs we adopted the exact HSP algorithm developed by Carastan-Santos *et. al* [Carastan-Santos et al. 2017]. For MICs we adopted the exact MIC HSP algorithm presented in Section 3.

Algorithm 1 Exact Hybrid Parallel Algorithm for Hitting Set Problem (Master Process).

Input: set X of variables, collection S, integer $k > 0$, number of candidates per task batch $b \cdot v$, stop constant $STOP$, number of slave processes n_s

Output: solution vector H where each subvector of H represents a subset $H \subseteq X$ with smallest cardinality, such that $|H| \leq k$ and $H \cap S \neq \emptyset, \forall S \in \mathcal{S}$.

1: $i \leftarrow 1$
2: H $\leftarrow \emptyset$
3: SortS $\leftarrow$ `SortClauses(`$\mathcal{S}$`)`
4: `BroadcastToSlaves(`SortS`)`
5: **while** $i \leq k$ **do**
6: $u \leftarrow \binom{|X|}{i}$
7: $\tau = \lceil u/(b \cdot v) \rceil$
8: $sp \leftarrow 0$
9: **for** $j = 0$ to τ **do**
10: $sPid \leftarrow$ `RecvAvailableSlave()`
11: `SendStartPoint(`sp`,` $sPid$`)`
12: $sp \leftarrow sp + (b \cdot v)$
13: **end for**
14: `WaitForSlavesToFinish()`
15: H $\leftarrow$ `GatherSolutions()`
16: **if** H is not empty **then**
17: **break**
18: **end if**
19: $i = i + 1$
20: **end while**
21: **for** $j = 0$ to n_s **do**
22: $sPid \leftarrow$ `RecvAvailableSlave()`
23: `SendStartPoint(`$STOP$`,` $sPid$`)`
24: **end for**
25: **return** H

Algorithm 2 Exact Hybrid Parallel Algorithm for Hitting Set Problem (Slave Process).

Input: set X of variables, vector SortS, master process ID $mPid$, stop constant $STOP$.

Output: solution vector localH where each subvector of localH represents a subset $H \subseteq X$ with smallest cardinality, such that $|H| \leq k$ and $H \cap S \neq \emptyset, \forall S \in \mathcal{S}$.

```
1: sPid ← GetProcessID()
2: localH ← ∅
3: w = |SortS|
4: loop
5:    SendAvailableSlave(sPid, mPid)
6:    sp ← RecvStartPoint()
7:    if sp = STOP then
8:       break
9:    end if
10:   SeeD ← NSGetSeeds(sp)
11:   EvalSolutionsOnPU
      (SeeD, SortS, localH, w, κ)
12: end loop
```

For a given cardinality i, $\tau = \left\lceil \binom{|X|}{i} / (b \cdot v) \right\rceil$ *task batches* are created. For typical input sizes the number of task batches τ is high, and these task batches can be dynamically assigned to the available PUs by the master process and executed as either a kernel, offload or function call. The key factor is to transfer as little information as possible to each slave process so that they can start their processing immediately. The b and v variables are tuning parameters that are explained with more detail in Section 2.2.

To control which instances will be processed by each slave process, the master process initializes a starting point variable sp as 0. When a slave requests new work, the master sends the current sp value and increases it by $b \cdot v$. After receiving sp, the slave process generates the list of candidates that will help the processing unit to generate its candidate solutions. Let X be the set of variables of an instance of HSP. To generate this list of candidates in an efficient manner, we use a combinatorial numbering system (Equation 2), which establishes a unique correspondence between a combination of elements of X of cardinality i, denoted by $C_i = \{c_i, c_{i-1}, ..., c_2, c_1\}$, and an integer N, $0 \leq N \leq \binom{|X|}{i}$ [Knuth 2005]. In this way, the combinatorial numbering system – denoted by the function `NSGetSeeds()` in Algorithm 2 – is used in the CPU by the slave process to generate only a very small number of candidate solutions, with $N = sp + (j \cdot \kappa)$, $0 \leq j < \beta$, $\kappa = \lfloor (b \cdot v)/\beta \rfloor$, where β is either the number of threads to be launched (in the case of CPU or MIC), or the number of GPU blocks to be launched (in the case of GPU). These combinations generated by the combinatorial numbering system are stored in a vector called SeeD and this vector is transmitted to the PU in the offload call, kernel call or function call. The other candidate solutions are generated in sequence for each thread in the PU during the offload call, kernel call or function call, having as starting point a unique combination present in the vector SeeD that was generated outside of the processing unit.

$$N = \binom{c_i}{i} + \binom{c_i - 1}{i - 1} + ... + \binom{c_2}{2} + \binom{c_1}{1}, \quad c_i > c_{i-1} > ... > c_2 > c_1 \geq 0 \tag{2}$$

Figure 1 illustrates this distribution of task batches among PUs. Found solutions

are stored on local solution vectors localH by each slave process. When the master detects that all of the τ tasks batches were assigned to a slave process, it means that all candidate solutions of the current cardinality i are being tested and then it waits (function `WaitForSlavesToFinish()` of the Algorithm 1) for all slave processes to complete its computations. Once the computations of all slave processes are completed, the master gathers the solutions found by them (function `GatherSolutions()` of the Algorithm 1). If no solution is found, it increases the cardinality by 1 and the entire process is repeated, until i reaches the value k.

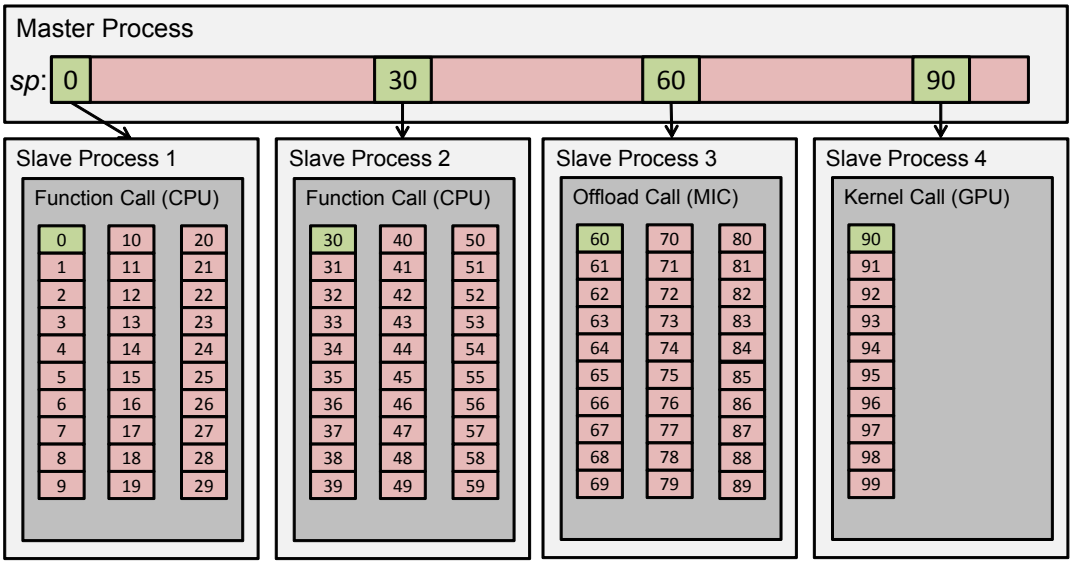


Figure 1. An example of the task batch distribution process among PUs, for 100 candidate solutions and task batch size $b \cdot v = 30$. Note that only information about four candidate solutions (the green ones) is sent through the network.

We should note that most of the data required by the algorithm are generated in parallel by different processes and the only large data structure sent using the network is the SortS (see line 4 of the Algorithm 1) structure, which is sent once at the beginning of the algorithm. During the algorithm execution, only small values, such as sp values, are transferred. At the end of the execution, the solution list is transferred. Therefore all the remaining calculations, including the enumeration and evaluation of candidate solutions, are done in parallel by the multiple available PUs and MPI processes. Consequently, this algorithm should present a good scalability with increasing numbers of PUs and machines.

2.2. Load Balancing

With heterogeneous configurations, we need to send task batch sizes according to the computational capabilities of the available PUs. To achieve such task, we use three tuning parameters b, v and t. The parameter v defines the number of threads per GPU block and its value is statically set according to the GPU architecture of the platform. A typical practice to set this value is to use the number of GPU cores per streaming processor. In its turn, the variable b defines the number of GPU blocks to be created in each kernel call. We defined these b and v variables to address the thread configuration in the case of GPUs,

which is significantly different from the other PUs. To address such a configuration for the case of CPUs and MICs, i.e., to define the number of threads t to be created on the CPU or MIC, t is set according to the hyperthreading factor (threads per core) multiplied by the number of cores of the CPU or MIC. It is important to note that, since the number of candidate solutions to be evaluated per task batch is defined by the product $b \cdot v$, the number of candidate solutions per GPU thread is set to 1, and for CPUs and MICs is set to $\lfloor (b \cdot v)/t \rfloor$.

A simple though efficient heuristic strategy is to generate many small task batches, which are deployed to the PUs as they become available. In this case, faster PUs will process more task batches than slower ones. One way to obtain good performance and keep all the PUs busy is by setting the aforementioned parameter b with a certain minimum value ($bMin$). This value can be empirically determined by executing the whole algorithm a few times using the same input size, but with increasing values of b. For small values of b, the MPI communication and kernel/offload launch overheads will dominate, but as b increases, this overhead decreases, resulting in smaller execution times. We select $bMin$ as the lowest b value for which the total execution time becomes constant with regard to b, which means that the overhead becomes negligible.

By performing this process on the fastest PU, we obtain a $bMin$ value that can be used as the value of b by all PUs in the platform. It's important to note that this process needs to be performed only once per cluster, with only a few tests with increasing values of b and with a small HSP instance. Once the $bMin$ value is set, multiple executions in the same platform, using different inputs, can be performed using the same $bMin$ value.

3. Exact MIC HSP Algorithm

We now describe the main characteristics of the proposed exact MIC HSP algorithm. As presented in Section 2.1, the part of our algorithm that is assigned to the processing units is the checking of the candidate solutions, which in the case of MICs, is executed in an offload call. Algorithm 3 shows the pseudo-code of the MIC version of the function `EvalSolutionsOnPU`.

Algorithm 3 Offload function EvalSolutionsOnPU (MIC version)

Input: Array SeeD, array SortS, solution vector localH, number of clauses w and number of candidates per MIC thread κ
Output: solution vector localH where each subvetor of H represents the subset $H \subseteq X$ with smallest cardinality, such that $|H| \leq k$ and $H \cap S \neq \emptyset, \forall S \in \mathcal{S}$.

1: $tr \leftarrow threadId$
2: **for** $c = 1$ to κ **do**
3: **if** $c = 1$ **then**
4: $C^{tr} \leftarrow SeeD^{tr}$
5: **else**
6: $C^{tr} \leftarrow$ `GetSubsequentComb(`C^{tr}`)`
7: **end if**
8: $solution \leftarrow true$
9: **for** $j = 1$ to w **do**
10: **if** C^{tr} and $SortS^{j}$ are disjoint sets **then**
11: $solution \leftarrow false$
12: **end if**
13: **end for**
14: **if** $solution = true$ **then**
15: atomically add C^{tr} in localH
16: **end if**
17: **end for**

Table 1. Cluster Configuration

Number	Processor	Accelerator	Acc. No. Cores
1	2xXeon E5-2620v2 2.1GHz six-core	Xeon Phi 3120A	57
1	Core i7-5930K 3.50GHz six-core	GTX Titan X	3072

Each thread tr of the MIC is responsible to evaluate κ candidate solutions. The first candidate solution to be evaluated by each thread is present in the vector SeeD that is sent to the MIC during offload. This candidate solution is assigned to the variable C^{tr} and the next steps is to check if C^{tr} is not disjoint with all of the clauses present in the vector SortS. If such condition holds true, C^{tr} is atomically added to a MIC local solution vector localH. Once a candidate solution is evaluated, the respective thread of the MIC generates the subsequent candidate solution to be evaluated, using the previous candidate solution as base.

One characteristic of the Xeon Phi is the 512-bit vector processors that are present in each core of the Xeon Phi co-processors, which constitutes a huge advantage [Jeffers and Reinders 2013]. In this regard, the vectorization of the algorithm can be done automatically by the Intel compiler, hence it is crucial that the most process demanding parts of the algorithm are implemented in a way that the vectorization can be performed by the compiler. As previously mentioned, the checking of the candidate solutions is performed through a disjunction check with the clauses of the HSP input. Since this disjunction check is the most computationally expensive task, we need to make sure that the implementation of this disjunction check exploits the 512-bit vector units present in the MIC. One can notice that we could avoid the execution of the entire loop of the lines 9-13 of the Algorithm 3 once a candidate solution is tagged as $false$. However, we implemented this loop in a way that we avoid unnecessary conditional deviations and early loop breaks, which would prevent vectorization. Although it would seem that executing this loop entirely for every candidate solution is a performance loss, the high vectorization capabilities of the Xeon Phi outperform the absence of the early loop break.

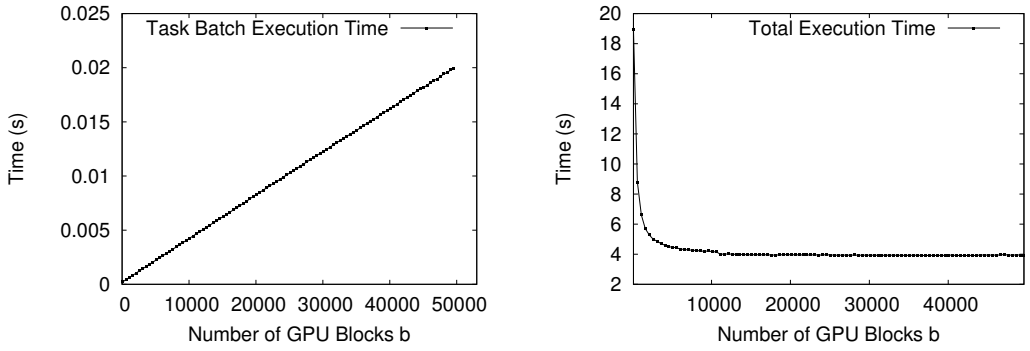
4. Experimental Results

Here we present the experimental results obtained with our hybrid HSP implementation, using a two-node heterogeneous cluster described in Table 1. The compilers used are Intel (`icc`) 16.0.1, except for the GPU parts, for which we used the NVIDIA compiler (`nvcc`) version 7.5. We also adopted Intel MPI version 5.1.2 for multi-node communication and conjunction of the processing modules. The optimization parameter `-O3` was used in all compilation steps.

For the experiments we generated HSP instances with $|X| = 8192$ (number of variables), collection of clauses $\mathcal{S}$ with $|\mathcal{S}| = 1024$ and $k = 3$. These HSP instances were randomly generated using the `rand()` function of the C standard library. For each instance, we generated random clauses with sizes in the interval $[\lceil min\_range * |X| \rceil, |X|]$, with $min\_range = 0.6$. This min_range value was empirically determined to permit the generation of instances with valid solutions. Instances with no solutions were discarded. In all experiments, we start to measure the total time at the moment the algorithm receives the HSP instance and finish when the results are returned.

4.1. Load Balance Tuning

To evaluate the algorithm for CPU-GPU-MIC platforms, the first step is to determine the minimum task batch size, that permitted the application to execute with acceptable overhead. As discussed in Section 2.2, smaller task batch sizes permit the generation of more task batches and a better load-distribution with heterogeneous platforms. Figure 2(a) shows the execution time of a single task batch in a GTX TITAN X. The execution time increases linearly with task batch size, controlled by the the parameter b (remembering that b is also the number of GPU blocks to be created in the kernel call). This indicates that moderate numbers of task batch size are sufficient to fully occupy the accelerator device. However, Figure 2(b) shows that only for b values above 20000 the total execution time becomes close to constant, indicating that below this value the overhead of MPI process communication and task batch launches becomes relevant. Hence, we set the parameter $bMin = 20000$ for the heterogeneous configuration of the experiments. We set the v parameter as 128, which is the number of cores per streaming processor of the GTX TITAN X [NVIDIA 2016]. Therefore, the number of HSP solution candidates to be evaluated per each task batch that will be processed in the PUs is $bMin{\cdot}v = 20000{*}128 = 2560000$.



(a) Task Batch Execution Time

(b) Total Execution Time

Figure 2. Task batch execution times and total execution times of our proposed algorithm in function of the number of GPU blocks parameter b.

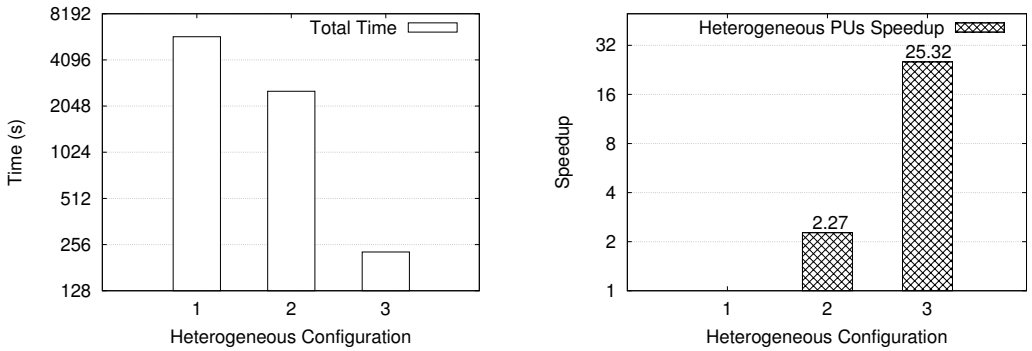
4.2. Performance Evaluation

We evaluated the scalability of our hybrid HSP implementation using a heterogeneous cluster composed of CPUs, GPUs and MICs. Figure 3(a) shows the average execution time of ten successive executions as a function of the heterogeneous configurations shown in Table 2. We see a noticeable improvement in the total execution time, from 1.6 hours to only 3.8 minutes. In this regard, we increased the number of variables $|X|$ from 8192 to 32768 and we were able to solve this HSP instance in only 4.1 hours with the four PUs (configuration 3 of Table 2. Result not shown in Figure 3(a)). With only two CPUs, the execution time would take about 25.32 times longer, or 103 hours.

Figure 3(b) shows the obtained speedups, measured as the ratios between the execution time of the HSP algorithm using a configuration shown in Table 2 and the execution time of the HSP algorithm using the dual socket CPUs (configuration 1 of Table 2). With

Table 2. Heterogeneous configuration used in the performance experiments.

Configuration	Description
1	2xXeon E5-2690 (dual socket configuration)
2	2xXeon E5-2690 and 1xXeon Phi 3120A (single node)
3	2xXeon E5-2690, 1xXeon Phi 3120A and 1xGTX TITAN X



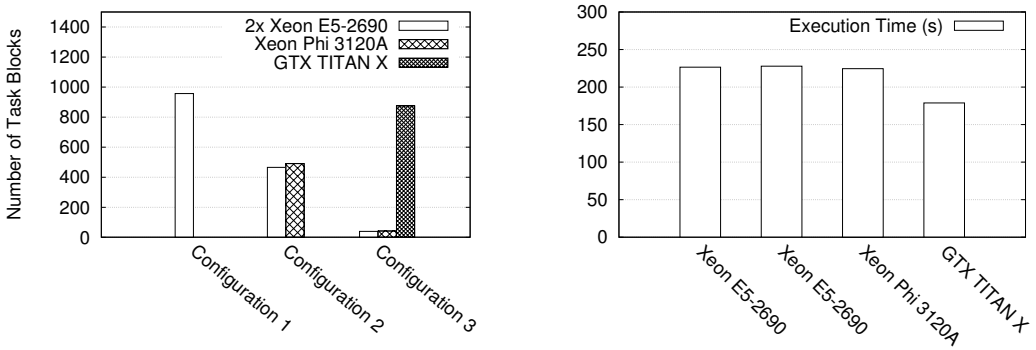
(a) Execution Time (log_2 scale on y axis) (b) Speedup (log_2 scale on y axis)

Figure 3. Execution time and consequent speedups of proposed hybrid algorithm in function of the number of processing units.

the addition of the Xeon Phi 3120A, the speedup jumped to 2.27, which shows that the Xeon Phi 3120A not only outperformed two Xeon E5-2690 CPUs, but also shows that the performance of the HSP algorithm can at least double by adding only one Xeon Phi device, therefore attesting the HPC capabilities of the Xeon Phi in our HSP algorithm. Another result that cannot be overlooked is the price and the performance delivered by the processing units. In a recent price survey (May 2017), two Xeon E5-2690 CPUs would cost around 4000 USD, while one Xeon Phi 3120A would cost around 1500 USD, showing that the Xeon Phi can be in fact an appealing alternative for high performance computing.

In its turn, with the addition of a GTX TITAN X, the speedup has further increased to 25.32. This indicates that the GTX TITAN X had better performance than the Xeon Phi 3120A when processing the task batches. Although the cores of the Xeon Phi 3120A are more complex than those of the GTX TITAN X, the massive number of cores of the GTX TITAN X outperforms the Xeon Phi 3120A in our hybrid HSP implementation. Since we assign one GPU thread per candidate solution, the GTX TITAN X can concurrently evaluate 3072 candidate solutions. With the Xeon Phi 3120A, however, with four threads per core, there are only 224 concurrent threads. Therefore, we can notice an advantage for the GPU in our hybrid HSP implementation. The price and performance delivered of the GTX TITAN X is also the best of the three configurations we evaluated. In a recent price survey (May 2017), one GTX TITAN X would cost around 2000 USD and it delivered an order of magnitude of performance increase in our HSP algorithm. However, it is important to note that the Xeon Phi 3120A is from the Knights Corner architecture, which is the first non-prototype architecture of Xeon Phi product line. Hence, the GPU *versus* MIC performance difference shown here might be lower with the current Xeon Phi architectures.

To get a better understanding of these results, we monitored the number of task batches assigned to each processing unit (Figure 4(a)). We can see that with three processing units, the Xeon Phi 3120A received 51% of the tasks, and with four processing units, the Xeon Phi 3120A and GTX TITAN X received 4.3% and 91.4% of the tasks, respectively. We also monitored the time that each PU spent with the HSP algorithm. Figure 4(b) shows the time spent for the four PUs used. As can be seen, with task batch sizes small enough, all PUs have a tendency of finishing its work at the same time, which is a good load balance measure. This shows that our hybrid algorithm can in fact handle heterogeneous CPU-GPU-MIC platforms, adequately balancing the load among the tasks in function of each processing unit's performance.



(a) Number of task batches assigned to each PU.

(b) Total time that each processing unit has spent solving HSP.

Figure 4. Load balance evaluation results of our hybrid exact HSP algorithm.

5. Conclusions

In this paper we presented an exact hybrid CPU-GPU-MIC algorithm for the Hitting Set Problem suited for large input size applications. We solved instances of HSP on the order of thousands of elements with reasonable time, achieving good performance and scalability with regard to execution time.

We performed several experiments on a heterogeneous cluster composed of CPUs Intel Xeon E5-2620v2, a MIC Intel Xeon Phi 3120A and a GTX TITAN X GPU. The results show that, in our algorithm, each of the accelerator device used has resulted in a satisfactory performance increase even for early development accelerator devices such as the Xeon Phi 3120A. Moreover, the results also show that our hybrid HSP algorithm has good performance and scalability by correctly distributing the tasks among the processing units according to their computational efficiency in processing the task batches. This enabled speedups of up to 25.32, when using all processing units instead of two six-core CPUs. Furthermore, with this cluster, we unprecedentedly solved HSP instances in order of tens of thousands of variables in less than 5 hours, which characterizes a formidable performance increase with regard to exact HSP algorithms. All of these improvements also reinforce the statement that the adoption of hybrid CPU-GPU-MIC algorithms can characterize a performance improvement if each architecture is adequately exploited, the load balance is properly set and the communication cost is minimized. However it is important to note that the execution time of our hybrid exact HSP algorithm can still be

unfeasible if the solutions present very high cardinalities. This problem can only be attenuated by adding more processing units to evaluate the solution candidates. But for applications where the cardinalities are not usually high, such as gene regulatory networks inference [Carastan-Santos et al. 2015, Carastan-Santos et al. 2017], the solution proposed here is suitable.

For future works, we plan to further optimize our algorithm to increase the efficiency of finding HSP solutions with very high cardinalities. Moreover, we also plan to provide a full software package for solving HSP, with a friendly interface and an autotuning procedure, capable to automatically discover the best tuning parameters for a specific cluster.

Acknowledgment

The authors would like to thank UFABC, FAPESP (Procs. n. 2013/26644-1, 2014/50937-1 and 2015/01587-0), CNPq (Procs. n. 559955/2010-3, 302620/2014-1 and 465446/2014-0) and CAPES for the financial support.

References

Andrade, G., Ferreira, R., Teodoro, G., Rocha, L., Saltz, J. H., and Kurc, T. (2014). Efficient execution of microscopy image analysis on CPU, GPU, and MIC equipped cluster systems. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, pages 89–96. IEEE.

Carastan-Santos, D., de Camargo, R. Y., Martins, D. C., Song, S. W., and Rozante, L. C. (2017). Finding exact hitting set solutions for systems biology applications using heterogeneous gpu clusters. *Future Generation Computer Systems*, 67:418–429.

Carastan-Santos, D., Yokoingawa De Camargo, R., Correa Martins, D., Song, S. W., Silva Rozante, L., and Ferreira Borelli, F. (2015). A multi-gpu hitting set algorithm for grns inference. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 313–322.

Cardoso, N. and Abreu, R. (2013). Mhs2: A map-reduce heuristic-driven minimal hitting set search algorithm. In *International Conference on Multicore Software Engineering, Performance, and Tools*, pages 25–36. Springer.

Cendic, B. L. (2014). A genetic algorithm for the minimum hitting set. *Scientific Publications of the State University of Novi Pazar*, 6(2):107.

Duran, A. and Klemm, M. (2012). The intel® many integrated core architecture. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 365–366. IEEE.

Gainer-Dewar, A. and Vera-Licona, P. (2017). The minimal hitting set generation problem: algorithms and computation. *SIAM Journal on Discrete Mathematics*, 31(1):63–100.

Garey, M. R. and Johnson, D. S. (1999). *Computers and Intractability - A guide to the Theory of NP-completeness*. W. H. Freeman and Company.

Hochbaum, D. S. (1997). *Aproximation Algorithms for NP-Hard Problems*. PWS Publishi ng Company.

Hvidsten, T. R., Lægreid, A., and Komorowski, J. (2003). Learning rule-based models of biological process from gene expression time profiles using gene ontology. *Bioinformatics*, 19(9):1116–1123.

Ideker, T. E., Thorsson, V., and Karp, R. M. (2000). Discovery of regulatory interactions through perturbation: inference and experimental design. *Pacific symposium on biocomputing*, 5:302–313.

Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi coprocessor high-performance programming*. Newnes.

Knuth, D. E. (2005). *The Art of Computer Programming, Fascicle 3: Generating All Combinations and Partitions*, volume 4. Addison-Wesley, Reading.

Kolman, P. and Walen, T. (2007). Reversal distance for strings with duplicates: Linear time approximation using hitting set. *The Electronic Journal of Combinatorics*, 14(1):11.

Murakami, K. and Uno, T. (2014). Efficient algorithms for dualizing large-scale hypergraphs. *Discrete Applied Mathematics*, 170:83–94.

NVIDIA (2016). Maxwell: The Most Advanced CUDA GPU Ever Made. `https://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/`. Online; last access 18 july 2016.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, 96(5):879–899.

Pearson, W. R., Robins, G., Wrege, D. E., and Zhang, T. (1996). On the primer selection problem in polymerase chain reaction experiments. *Discrete Applied Mathematics*, 71(1):231–246.

Reza, H., Aguilar, M., and Jalal, S. F. (2015). Regression testing of gpu/mic systems for hpcc. In *Proceedings of the 2015 International Workshop on Software Engineering for High Performance Computing in Science*, pages 30–37. IEEE Press.

Ruchkys, D. P. and Song, S. W. (2003). A parallel solution to infer genetic network architectures in gene expression analysis. *International Journal of High Performance Computing Applications*, 17(2):163–172.

Shi, L. and Cai, X. (2010). An exact fast algorithm for minimum hitting set. *Int. Joint Conference on Computational Science and Optimization*, 1:64–67.

Sîrbu, A. and Babaoglu, O. (2016). Power consumption modeling and prediction in a hybrid cpu-gpu-mic supercomputer. In *European Conference on Parallel Processing*, pages 117–130. Springer.

Wolfe, N., Liu, T., Carothers, C., and Xu, X. G. (2014). Heterogeneous concurrent execution of monte carlo photon transport on cpu, gpu and mic. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, pages 49–52. IEEE Press.

An Advance Resource Reservation Approach in a Cloud Database Environment

Vinicius da S. Segalin[1], **Carina F. Dorneles**[1], **Mario A. R. Dantas**[1]

[1]Departamento de Estatistica e Informatica (INE)
Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brazil

vinicius.segalin@posgrad.ufsc.br,dorneles@inf.ufsc.br,mario.dantas@ufsc.br

Abstract. *A well-known challenge with long running time queries in database environments is how much time a query will take to execute. This prediction is relevant for several reasons. For instance, by knowing that a query will take longer to execute than desired, one resource reservation mechanism can be performed, which means reserving more resources in order to execute this query in a shorter time in a future request. In this research work, it is presented a proposal in which the use of an advance reservation mechanism in a cloud database environment, considering machine learning techniques, provides resource recommendation. The proposed model is presented, in addition to some experiments that evaluate benefits and the efficiency of this enhanced proposal.*

1. Introduction

Advance reservation can be defined as the assignment of the capacity of private resources in a restricted or limited fashion in previously defined time interval [MacLaren 2003, Gomes and Dantas 2014]. Therefore, in our scenario, this can be understood as the process where users can reserve computational resources to be used for a period of time. Usually, the resources reserved are CPU, memory, space in disk and bandwidth. Resource reservaftion aims to assure a user will have the requested resource at the requested time, allowing the user to have the expected performance in a defined time interval [Sulistio and Buyya 2004].

At the same time, cloud database services (database-as-a-service or DBaaS) manage data and automate operations of distributed database infrastructures. It has been seen a trend in the increase of the data size never seen before, as a consequence the more susceptible databases are to have a large amount of data [Assunção et al. 2015], and the more data can be translated to more data management complexity and usually longer the queries will take [Singhal and Nambiar 2016]. For this reason, in some cases it is important to have an estimation of the time in which a query will take to finish, such as the examples that are followed highlighted:

- A Database as a Service (DBaaS) provider requires to monitor constantly the workload so it can decide to accept (or even postpone) query executions in order to follow a Service Level Agreement (SLA). If a query takes longer than expected, the client will not have the results as promised and both would be harmed. However, if the provider knows this issue in advance, it could reschedule this query to a more appropriated period when it will be able to finish the requested execution without consequences.

- On online games, there are often events to attract new players or to benefit those who already play. In these events, that take place at a certain period of time, database requests increase substantially, demanding more from it. By having an estimate of how much it would increase and require from the database, the provider can, momentarily, improve its hardware capabilities in order to maintain the performance.

In such scenarios, a query execution time prediction would help to avoid undesirables problems, even if the prediction is not totally accurate, for cases when it is not very strict. Predicting a query execution time is not a simple task, though, since many variables can influence the query execution, such as concurrent and dynamic workloads, locks obtained by other queries and other processes keeping the machine busy. These and other reasons result in the same query having different execution times depending on the moment they are executed. As a result, to be able to provide a reasonable estimation, some works in literature usually ignore some of these variables due to their unpredictability.

Many approaches have been proposed to deal with these difficulties, such as analytical modeling [Wu et al. 2013a, Singhal and Nambiar 2016], statistical models [Ahmad et al. 2011] and machine learning [Ganapathi et al. 2009, Matsunaga and Fortes 2010, Gupta et al. 2008]. The latter, much more discussed in literature, has been chosen for the approach adopted to the research work described in this paper, since it is easier to implement, its accuracy is proven by previous works, and as observed in some works [Ganapathi et al. 2009, Wu et al. 2013b], the cost model by itself, provided by DBMSs, is not very useful and reliable to estimate query time execution. Our proposal aims to allow a user to reserve resources to utilize in the future and execute a query in a shorter time through an estimation of how long the query will take to be executed in different configurations (e.g., one machine with twice the memory of another).

In this paper, we present an approach to provide users of DBaaS an advance resource reservation mechanism, which allows them to enhance their databases hardware temporarily in order to improve performance. In addition, we use a prediction mechanism based on machine learning to give the user a resource recommendation regarding time and cost. To verify our approach, we have tested different machine learning algorithms and Docker [Merkel 2014] to simulate different machines. Experiments have been made using two different datasets: one with synthetic data and another with real data, and the DBMS used was PostgreSQL. Our experiments have presented interesting results for query running time prediction, with an accuracy, in average, above 80%. It is possible to conclude from our effort that we were able to provide a more accurate resource reservation by helping any user to choose the most suitable resource combination.

The paper is organized as follows. Section 2 discusses some related works. We present our approach and its objectives in Section 3. In Section 4, we describe our experimental environment and result cases from the utilization of our proposal approach. We conclude the paper in Section 5.

2. Related Work

In this section, we present related work in two main subject dimensions, those the query execution time prediction issue and resource reservation.

2.1. Query Execution Time Prediction

In past years, there has been significant work in predicting query execution time. Some works are developed using commercial database systems, such as HP NeoView [Ganapathi et al. 2009], IBM DB2 [Ahmad et al. 2011] and SQL Server [Lee et al. 2016]. The proposal described in [Ganapathi et al. 2009] uses machine learning to predict query execution time and other performance metrics, such as message count and disk I/O. In [Ahmad et al. 2011], the focus is to provide query completion time prediction by fitting statistical models to query samples based on machine learning techniques. [Lee et al. 2016] presents a new feature in Microsoft SQL Server 2016 to provide query progress estimation, giving a percentage of work done. Although this can be useful for the user to know how much progress has been done so far, this kind of approach does not predict how much time the query will take to finish, nor can it be done prior to the query execution.

Some works propose to predict query execution time on PostgreSQL. In [Wu et al. 2013b], the authors propose an alternative for machine learning, stating the cost model provided by the query optimizer can be used, if well calibrated, to predict query execution time, and other techniques, such as machine learning, are not strictly necessary. In [Wu et al. 2013a] the objective is to propose an analytical model to predict query execution time for dynamic concurrent workloads, which the authors state is competitive to machine-learning based approaches. Another analytical model proposed is presented in [Singhal and Nambiar 2016], which dynamically adapts itself to the structure of the query execution plans to deal with large data. A framework is proposed in [Duggan et al. 2014] using a combination of semantic information and empirical evaluation to build a model able to cope with concurrent and dynamic workloads.

The idea described in [Matsunaga and Fortes 2010] extends a classification tree algorithm [Gupta et al. 2008] and compares some other machine learning techniques to propose a new method to predict execution time, memory and disk requirements. Outside the relational database world, [Hasan and Gandon 2014] compares two machine learning algorithms to provide SPARQL query performance prediction, and [Farias et al. 2016] uses machine learning to predict mean response time (MRT) per second in NoSQL databases.

Considering all analyzed works about query execution time prediction, [Ganapathi et al. 2009] has been selected as the baseline for the approach presented in this paper. The reason is the good results the authors have achieved through machine learning, and the fact that they have used the TPC-DS benchmark for their experiments, which has allowed us to compare our results.

2.2. Resource Reservation

Resource reservation has been explored in [Funke et al. 2012], where the authors show how beneficial resource reservation may be in a cloud computing environment for both the consumer and the provider. In a similar approach, [He 2015] states nowadays cloud providers take the decision of provisioning resources based only on workload peaks, what usually results in excess of resource provisioning and the need to scale up/down drastically, causing high service costs. The author, therefore, proposes an elastic reservation mechanism to enhance the use of resources and the user's satisfaction. [Wang et al. 2015] proposes a new mechanism in cloud environments to allow users to reserve instances de-

pending on their real need, offering more flexibility to reserve resources for any length and from any point in the future, and not only for predefined options as performed by cloud providers nowadays.

Resource reservation is not only applicable to cloud environments. For instance, [Gomes and Dantas 2014] proposes an approach for an opportunistic grid computing environment, where there are many personal computers in the grid sharing resources. In this type of environment, users share their idle resources, so in many situations there may be no available resource being shared, therefore reserving can be one way of increasing the chances of having the needed resource available for the future.

2.3. Summary

Our proposal is the first step forward to a solution of resource reservation that uses a method to predict the time the query will take to execute in order to make a resource recommendation to the user. [Funke et al. 2012, He 2015, Wang et al. 2015] assume the users know by advance the duration of their reservation, while [Gomes and Dantas 2014] calculates the average of CPU and memory usage to help the user to make the reservation choice. Moreover, no resource reservation approach has been proposed DBaaS.

To fill this gap, we have studied ways to provide query execution time prediction in order to use it in resource reservation. Section 2.1 has presented some different approaches that would allow us to perform the prediction, and as mentioned before, we have chosen machine learning to do so.

3. Proposed Approach

As illustrated in Figure 1, our approach contribution can be understood in two dimensions: the offline processing and the online execution. During the offline processing, the model that will provide the predictions is prepared to be used on the online execution. First of all, we create the containers [Bernstein 2014], which will represent to the user the machines that can be hired. Afterward, the queries are executed inside each container to obtain the query plans with their execution time. The query plan works as the feature vector, while the execution time is the variable we want to predict. The algorithm, during the training, adjusts parameters in order to map the input features to the output variable. The queries executed inside the containers should be the same, so the containers are all trained with the same data and can achieve the same prediction performance. The queries can be acquired either by analyzing the database log or by generating new ones. As all the containers execute the same queries, this process of acquiring them needs to be performed only once. Finally, with the query plans and the execution times as input, the machine learning algorithm can be trained. This step should be performed by someone who knows the queries and the database schema, so this specialist can decide the best parameters and tune the model in order to get the best prediction possible. This entire part is done offline, since it may take hours, or even days, depending on the dataset, to execute all the training queries and to extract the data needed to train the models.

The second step is the online execution, which starts with the query executed by the user. This query is sent to the database, which returns only the query plan without the execution time, since it does not execute the query. The query plan is sent to the previously trained models, which will be able to output the prediction of the execution

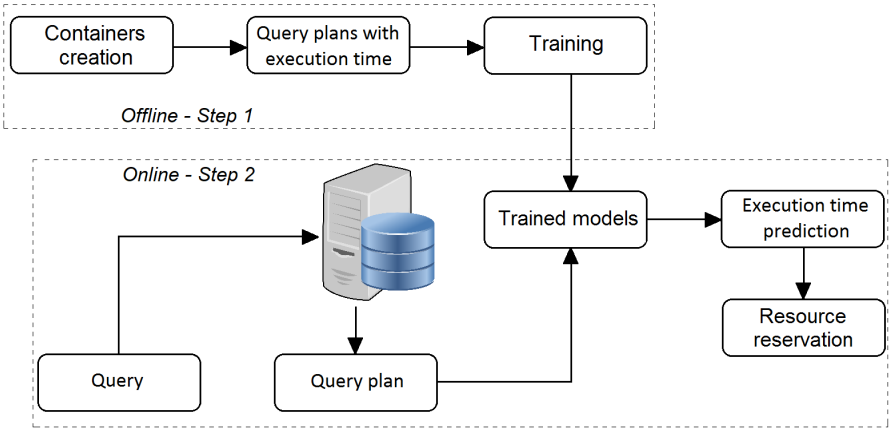


Figure 1. Resource reservation approach

time for each container. With these predictions, the user will be able to know which resource combination to choose according to the needs in terms of time and cost.

This model allows the prediction to be nearly instantaneous to the user, provided that the long processing task (i.e., to obtain the query plans and execution times and to train the machine learning models) is performed offline. The prediction, to the final user, is fast, since it will only use information available before query execution starts (i.e., the query plan produced by the query planner).

The final step of the approach - the resource reservation - depends on the choice the user has made. After receiving the resource combinations from the provider with time and cost for the entered query, the user must decide whether to stay with the same resources or migrate temporarily the database. By choosing the second option, the user selects the resource set with the best cost benefit ratio and chooses the moment the database migration should occur (for example, user reserves resources for 5 pm on the next day). The provider is then in charge of creating the container of choice, migrating the database to the new container at the moment chosen by the user, and migrating the database back to the original container after this period has ended. Figure 2 illustrates the entire online execution process.

Using this approach, the user receives a resource recommendation through the query execution time prediction. The cloud provider is then in charge of the choice of how it should be implemented, including the containers creation, machine learning process and database migration. An important consideration though, is that the machine learning training should be executed in a frequent manner as the data volume in the database grows. The reason is that query plans change according to how much data there is in the database, therefore the trained model needs to be up-to-date in order to provide a correct prediction. The training frequency varies according to the database utilization, thus it is the specialist duty to monitor the need to perform this procedure.

4. Experimental Evaluation

In this section, we describe the experiments we have performed in order to empirically validate our approach. The goal of these experiments is to demonstrate that it is possible to

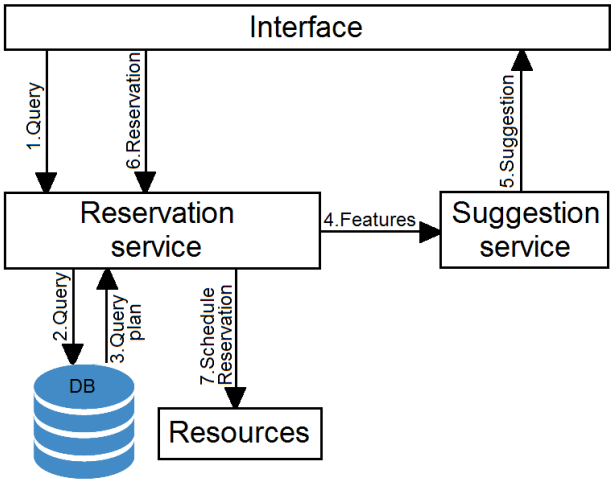


Figure 2. Resource reservation

provide suggestions for resource reservation by having a good execution time prediction, so the user could select the best resource combination, allowing the provider to reserve it. The section presents how our experiments environment has been configured, which machine learning techniques have been used, and which results we have reached on testing our approach.

4.1. Environment Settings, Datasets and Evaluation Metric

The experiments have been executed on a virtual machine provided by the cloud service at the Federal University of Santa Catarina (UFSC), which uses VMware VSphere ESXi 5.5, with an 8 core Intel i7 3.07 GHz processor and 32 GB of RAM. We have used PostgreSQL 9.5.4 on Ubuntu 16.04 to execute the queries and to obtain the query plans. We discuss and compare the results of query execution time prediction with a baseline from literature that we have considered to be most relevant, that is that presented in [Ganapathi et al. 2009].

The datasets used are mainly an open dataset from the Brazilian government[1], called *Prova Brasil DB*, and the benchmark TPC-DS [Nambiar and Poess 2006]. *Prova Brasil DB* has been used for the experiments of resource reservation, while TPC-DS has been used to compare our machine learning results to our baseline.

Since we did not have real user queries for any of the datasets, we have designed a simple algorithm to generate a workload, which aimed to provide us all kinds of representative queries that would give us distinct execution times. This script has generated different kinds of queries with single tables and using joins, so the combination variety would be large, providing us hundreds of different queries.

We have evaluated our models comparing the graphs of predicted time vs. actual time. These graphs show the precision the models have obtained by predicting several query execution times, therefore have helped us to identify which model had better fitted our data. Besides, we have predicted the execution time for random queries to confirm we could get good predictions with the model.

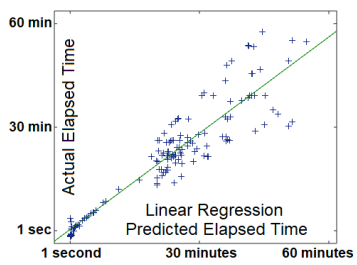
[1]http://dados.gov.br/dataset/microdados-prova-brasil

4.2. Query Execution Time Prediction

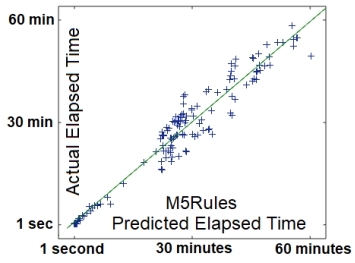
This section presents the results we have reached by comparing our query time prediction experiments with our baseline [Ganapathi et al. 2009]. In the same manner as performed by [Ganapathi et al. 2009], we have instantiated the TPC-DS benchmark at scale factor 1. We have then generated 1487 queries with the script mentioned before. Since it was difficult to have an estimate of the time the queries would take, specially because the same template could generate a query that takes only seconds and another one that would take hours (e.g., a cross join with 3 small tables and a cross join with 3 big tables), they were executed with the intention of having a wide time variation. In the end, the queries execution took from milliseconds to 63 minutes.

Figures 3a-3c show the prediction using our queries with Linear Regression [Kutner et al. 2004], M5Rules [Quinlan et al. 1992] and Multilayer Perceptron [Pal and Mitra 1992] respectively, while Figure 3d shows the prediction performed by [Ganapathi et al. 2009]. The graph in Figure 3a shows our data does not present a linear pattern, where a few short queries have had good predictions, while longer queries have presented bad results, with both very high or low predictions. Figure 3b indicates the algorithm M5Rules is better for this dataset than Linear Regression. For this algorithm, the data is closer to the perfect prediction line than for the first experiment, indicating better results, although still with many wrong predictions (the data more distant from the line).

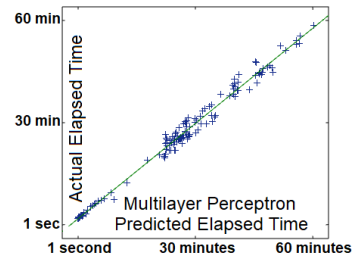
Multilayer Perceptron, represented in Figure 3c, has given the best results among the experimented algorithms. Data is closer to the diagonal line than to the other algorithms, presenting a similar pattern to the one in Figure 3d, our baseline.



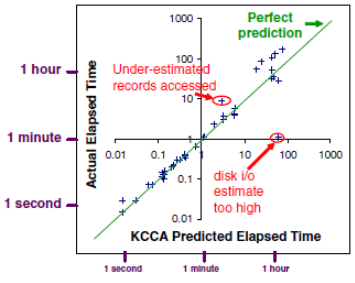
(a) Prediction using Linear Regression



(b) Prediction using M5Rules



(c) Prediction using Multilayer Perceptron



(d) Result from Ganapathi et al. 2009

Figure 3. Comparison of different algorithms and (d) the baseline

In addition to having presented a similar pattern to the approach we consider as

our baseline, we randomly generated 10 more queries in order to see whether we could make good execution time predictions. Table 1 shows the results. It can be seen good predictions have been made for the long queries (>100 seconds) and bad predictions for the short queries (<60 seconds). This indicates this model is not suitable for predicting the execution time for short queries, although it looks accurate for long queries. As our approach uses query execution time prediction to provide the users a resource recommendation to execute long queries faster, bad predictions for short queries do not affect the proposed model. Thus these results encouraged us to use this algorithm to provide a good resource suggestion for advance reservation.

Table 1. predictions made on TPC-DS

#	Exec time (s)	Prediction	Prediction error
1	1029.30	1102.25	72.95 / 7.08%
2	625.33	679.33	54.00 / 08.63%
3	104.83	114.20	9.37 / 8.93%
4	7.98	56.79	48.81 / 611.74%
5	1.86	47.704	45.83 / 2452.38%
6	50.97	82.03	31.06 / 60.94%
7	210.21	233.80	23.59 / 11.22%
8	565.23	552.14	-13.08 / -2.31%
9	996.77	1036.41	39.64 / 3.97%
10	1163.05	1117.25	-45.801 / -3.94%

To assure our model would work and provide us good predictions, more experiments have been performed on two other datasets (a synthetic DVD rental database[2] and *Prova Brasil DB*). The results obtained have presented the same accuracy as the ones of this section, but due to the limited space, we do not present them in this paper.

4.3. Containers Setup

We have used Docker to simulate new machines with different configurations. Table 2 shows the containers created and their respective resources. All the containers have been created in the same virtual machine, therefore they all share the same Operating System and resources (we have only limited some of them for the experiments). We have stipulated a fictitious price to each of them as part of the experiments, so the user would have to consider the cost benefit ratio when choosing which configuration to reserve.

Table 2. Containers configuration

Container Name	Memory (MB)	PE (Processing Element)	Price Per hour
C1	1024	1	$0.05
C2	2048	1	$0.07
C3	2048	2	$0.08
C4	4096	4	$0.11

[2]http://www.postgresqltutorial.com/postgresql-sample-database/

4.4. Resource Reservation Experiments

In this section, we present the experiments performed to test resource reservation. We analyze two scenarios. The first one does not use our model, while the second one does. We then compare the scenarios to show the benefits in using resource reservation.

To simulate a real situation, the user has executed a query that was thought to be long[3] (for our scenario, longer than 1 hour):

```
SELECT *
FROM ts_item, ts_pesos, ts_quest_professor,
ts_resposta_aluno, ts_quest_diretor;
```

4.4.1. Without Resource Reservation

To test this scenario, we have used the database in a normal fashion. The container used for this test was C1. In this case, the user has executed the query and would take, according to our model prediction, 108 minutes to complete. Since the user did not use our proposed model, no prediction has been provided, thus the query was executed with no prior knowledge of how long it would take and took 103 minutes. In a fictitious situation, the user would have to wait for almost two hours in the expectation of the work to finish.

4.4.2. With Resource Reservation

In this scenario, we have used our proposed approach. In order to do so, the cloud provider had to perform the steps presented in Section 3, which means it had created containers with different resources and trained, with the same queries and the same machine learning algorithm, one network for each container that would be able to provide query execution time prediction. For our experiment, we have created the four containers described in Section 4.4. Besides the different resources for each container, we have configured each PostgreSQL instance to be optimized according to the resources.

By using the resource reservation, we had an estimate of how long the query would take if we were to enhance our machine's capacities. To make the predictions we have used multilayer perceptron, which provided us the following values: 108 minutes for C1, 81 minutes for C2, 80 minutes for C3, and 87 minutes for C4.

As we can see, the improvement from C1 to C2 was significant, result of an extra GB of RAM added. On the other hand, adding an extra PE has not given apparent improvements to C3, probably to the fact that the query is executed by a single core, so the new one would be idle for this execution. Curiously, the prediction for C4 is about 7% worse than for C2, even with double the memory and PE. This could be only a prediction error, but we can conclude that it has not presented improvements because our queries did not demand more memory, so this extra GB was insignificant for our dataset.

The cloud provider, with the predictions made, can present to the user each prediction with the price it would take to execute the query. With this in mind, the user would

[3]This query is from the dataset *Prova Brasil DB*

see something similar to the information presented in Table 3. Note that the table does not show the configuration of each container. The only relevant information for the user is the time and cost it will take to execute the query. Any other information is not important for the user as the final client.

Table 3. Price/time to execute the query

Configuration	Estimated time (minutes)	Price
1	108	$0.10
2	81	$0.14
3	80	$0.16
4	87	$0.22

The information presented in Table 3, which is shown to the user, is considered to be a recommendation to help the user to choose a set of resources that are able to execute the input query faster than originally. The user can then select which resource combination to use or, if desired, to maintain the resources already hired. For our case, we have compared the cost we would have for each configuration. Considering it would take 108 minutes with C1, we would spend $0.10 to execute the query, according to Table 3, and $0.14 if the query takes 81 minutes in C2 (for this conclusion we have only considered full hours). Therefore we have chosen to migrate to C2 temporarily to execute the query and go back to C1 after its execution. To perform this, we have reserved C2 for 2 hours at the end of the day. The provider, after the user decides which configuration to use, should schedule the database migration from the original configuration to the configuration of choice, but only during the period agreed. After that, the database should migrate back to the original configuration.

For our experiment, thus, at the end of the day the database has been migrated to C2 during two hours, the sufficient time to execute the query. At the end, the query took 83 minutes to complete, 2 minutes later than expected. 37 minutes later, by the end of the reserved period, the database has been migrated back to the original configuration, and this transition was imperceptible, since container creation takes only a few seconds.

4.4.3. Considerations

This section has presented our approach experiments. We have presented two scenarios, in which the first the user has not used our approach and has taken 103 minutes to execute the query paying $0.10 for the two full hours. In the second scenario, the user has chosen to join our approach and executed the query 20 minutes faster paying $0.14. With our approach the user had to consider the cost benefit ratio, and has chosen to pay $0.04 more in order to execute the query 20 minutes faster.

5. Conclusions and Future Work

In this research work paper, we have presented an advance resource reservation approach for DBaaS, adopting machine learning to provide accurate resource recommendations. Through many experiments, we have obtained some very reasonable predictions with high accuracy for long-running queries. Using these predictions, our contribution proposal is

able to present to any user an estimate of how long the input query would take to execute in different resource sets, therefore enabling a selection of one that could better match requirements from the user viewpoint in terms of time and cost. The approach proposal main goal allows users of DBaaS to execute a number of queries in a shorter time than they would at the resource set contracted. As a result, users have the option to pay a little more for better resources in a short predefined period of time, assuming the new resource set would result in a considerable execution time improvement.

We have executed experiments in both synthetic and real datasets, using only generated queries. For future experiments, we would like to be able to test the effectiveness of the technique in a real life situation. To do so, we should have a real workload with hundreds of queries, and also consider concurrency to simulate the everyday use. More experiments could be performed using bigger datasets to explore how the database size can influence, specially regarding the user experience. In addition, we have simulated a cloud environment with synthetic values. For future work, a real cloud provider is considered, providing more resource combinations and choosing the one with the best cost-benefit ratio automatically. An improvement that could be applied to our work is to further explore machine learning techniques or to use another approach in order to obtain more accurate query execution time predictions. For future research it is possible to conceive an improvement for the proposal with some features such as more complex queries and I/O analysis prediction similar to those presented in [Inacio et al. 2017].

References

Ahmad, M., Duan, S., Aboulnaga, A., and Babu, S. (2011). Predicting completion times of batch query workloads using interaction-aware models and simulation. In *Proc. of the 14th Int. Conf. on Extending Database Technology*, pages 449–460. ACM.

Assunção, M. D., Calheiros, R. N., Bianchi, S., Netto, M. A., and Buyya, R. (2015). Big data computing and clouds: Trends and future directions. *Journal of Parallel and Distributed Computing*, 79:3–15.

Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84.

Duggan, J., Papaemmanouil, O., et al. (2014). Contender: A resource modeling approach for concurrent query performance prediction. In *EDBT*, pages 109–120.

Farias, V. A., Sousa, F. R., Maia, J. G., Gomes, J. P., and Machado, J. C. (2016). Machine learning approach for cloud nosql databases performance modeling. In *Cluster, Cloud and Grid Computing, 2016 16th IEEE/ACM Int. Symposium on*, pages 617–620. IEEE.

Funke, D., Brosig, F., and Faber, M. (2012). Towards truthful resource reservation in cloud computing. In *Performance Evaluation Methodologies and Tools (VALUETOOLS), 2012 6th International Conference on*, pages 253–262.

Ganapathi, A., Kuno, H., Dayal, U., Wiener, J. L., Fox, A., Jordan, M., and Patterson, D. (2009). Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Data Engineering, 2009. IEEE 25th Int. Conf. on*, pages 592–603. IEEE.

Gomes, E. and Dantas, M. A. R. (2014). An advance reservation mechanism to enhance throughput in an opportunistic high performance computing environment. In *Network Computing and Applications, IEEE 13th Int. Symposium on*, pages 221–228. IEEE.

Gupta, C., Mehta, A., and Dayal, U. (2008). Pqr: Predicting query execution times for autonomous workload management. In *Autonomic Computing, 2008. ICAC'08. International Conference on*, pages 13–22. IEEE.

Hasan, R. and Gandon, F. (2014). A machine learning approach to sparql query performance prediction. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2014 IEEE/WIC/ACM Int. Joint Conferences on*, volume 1, pages 266–273. IEEE.

He, H. (2015). Virtual resource provision based on elastic reservation in cloud computing. *Int. J. Netw. Virtual Organ.*, 15(1):30–47.

Inacio, E. C., Barbetta, P. A., and Dantas, M. A. (2017). A statistical analysis of the performance variability of read/write operations on parallel file systems. *Procedia Computer Science*, 108:2393–2397.

Kutner, M. H. et al. (2004). *Applied linear regression models*. McGraw-Hill/Irwin.

Lee, K., König, A. C., Narasayya, V., Ding, B., et al. (2016). Operator and query progress estimation in microsoft sql server live query statistics. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1753–1764. ACM.

MacLaren, J. (2003). Advance reservations: State of the art. *Global Grid Forum*.

Matsunaga, A. and Fortes, J. A. (2010). On the use of machine learning to predict the time and resources consumed by applications. In *Proc. of the 10th IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing*, pages 495–504. IEEE Computer Society.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

Nambiar, R. O. and Poess, M. (2006). The making of tpc-ds. In *Proc. of the 32nd Int. Conf. on Very large data bases*, pages 1049–1058. VLDB Endowment.

Pal, S. K. and Mitra, S. (1992). Multilayer perceptron, fuzzy sets, and classification. *IEEE Transactions on neural networks*, 3(5):683–697.

Quinlan, J. R. et al. (1992). Learning with continuous classes. In *5th Australian joint conference on artificial intelligence*, volume 92, pages 343–348. Singapore.

Singhal, R. and Nambiar, M. (2016). Predicting sql query execution time for large data volume. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, pages 378–385. ACM.

Sulistio, A. and Buyya, R. (2004). A grid simulation infrastructure supporting advance reservation. In *16th International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*, pages 9–11.

Wang, C., Ma, W., Qin, T., Chen, X., Hu, X., and Liu, T.-Y. (2015). Selling reserved instances in cloud computing. In *IJCAI*, pages 224–231.

Wu, W., Chi, Y., Hacígümüş, H., and Naughton, J. F. (2013a). Towards predicting query execution time for concurrent and dynamic database workloads. *Proceedings of the VLDB Endowment*, 6(10):925–936.

Wu, W., Chi, Y., Zhu, S., Tatemura, J., Hacigümüs, H., and Naughton, J. F. (2013b). Predicting query execution time: Are optimizer cost models really unusable? In *Data Engineering (ICDE), 2013 IEEE 29th Int. Conf. on*, pages 1081–1092. IEEE.

BinLPT: A Novel Workload-Aware Loop Scheduler for Irregular Parallel Loops

Pedro Henrique Penna[1,2,3], Márcio Castro[1], Patricia Plentz[1], Henrique C. Freitas[2], François Broquedis[3], Jean-François Méhaut[3]

[1] Universidade Federal de Santa Catarina, Florianópolis, Brazil
[2]Pontifícia Universidade Católica de Minas Gerais, Belo Horizonte, Brazil
[3]Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

pedro.penna@posgrad.ufsc.br
{marcio.castro, patricia.plentz}@ufsc.br, cota@pucminas.br
{francois.broquedis,jean-francois.mehaut}@univ-grenoble-alpes.fr

Abstract. *Workload-aware loop schedulers were introduced to deliver better performance than classical strategies, but they present limitations on workload estimation, chunk scheduling and integrability with applications. Targeting these challenges, in this work we propose a novel workload-aware loop scheduler that is called BinLPT and it is based on three features. First, it relies on some user-supplied estimation of the workload of the target parallel loop. Second, BinLPT uses a greedy bin packing heuristic to adaptively partition the iteration space in several chunks. The maximum number of chunks to be produced is a parameter that may be fine-tuned. Third, it schedules chunks of iterations using a hybrid scheme based on the LPT rule and on-demand scheduling. We integrated BinLPT in OpenMP, and we evaluated its performance in a large-scale NUMA machine using a synthetic kernel and 3D N-Body Simulations. Our results revealed that BinLPT improves performance over OpenMP's strategies by up to 45.13% and 37.15% in the synthetic and application kernels, respectively.*

1. Introduction

Evenly distributing the workload among the working threads of an irregular application is a NP-Hard minimization problem known as the Multiprocessor Scheduling Problem. This is a challenge to both academic and industry communities, and it is a recurring subject of research in High Performance Computing (HPC). In shared memory parallel applications, this problem emerges when scheduling iterations of parallel loops [Polychronopoulos and Kuck 1987]. In this scenario, the problem is referred as the Loop Scheduling Problem (LSP), and it can be reduced to the assignment of independent loop iterations such that their load is evenly distributed and the scheduling overhead is minimized.

Several loop scheduling strategies have been proposed to address the previous problem [Hurson et al. 1997], and they mainly rely on two techniques. In the first, called on-demand scheduling, iterations are scheduled to threads on-the-fly at runtime, so that both load imbalance and runtime variations may be dynamically handled. In the second technique, called chunk-size tuning, iterations are scheduled in optimally sized batches (*i.e.* chunks) so that (i) scheduling overheads are mitigated, (ii) load imbalance is further amortized and (iii) iteration affinity is exploited. When coupled together, on-demand scheduling and chunk-size tuning may indeed deliver reasonable performance to a wide

range of scenarios. Nevertheless, these techniques do not consider any knowledge about the underlying workload of the target parallel loop, and thus scheduling strategies built upon them naturally turn out to be suboptimal [Penna et al. 2016]. To address this weakness, workload-aware strategies were introduced [Banicescu and Velusamy 2001, Kejariwal et al. 2006, Wang et al. 2012]. These strategies rely on some knowledge about the workload to adaptively fine-tune chunk sizes to further amortize load imbalance, and thus deliver superior performance. Although these strategies present better performance gains than workload-unaware strategies (or blind strategies), existing workload-aware loop scheduling strategies still face some drawbacks that should be addressed.

First, these strategies rely on profiling and statistical regression techniques, and thus are inherently designed to well-behaved workloads. To tackle irregular loops, whose workload varies drastically, some alternatives for estimating the workload on-the-fly are necessary. Usually, the loop scheduling strategy itself and the workload-estimation technique should be loosely coupled. This way, HPC engineers may plug into their solutions the workload-estimator that best fits their needs. However, existing knowledge-based strategies do not provide this flexibility. Moreover, existing workload-aware loop scheduling strategies fail to apply their knowledge about the underlying workload of the target irregular loop when scheduling chunks of iterations. They only rely on-demand scheduling technique, which leads to scalability problems [Bull 1998]. Finally, no workload-aware strategy is integrated in a publicly available parallel Application Programming Interface (API) or library, since such integration is usually not trivial [Banicescu 2003].

In this context, the main goal of this work is to propose a novel workload-aware loop scheduling strategy for irregular parallel loops in which iterations are independent from one another. This new strategy overcomes the aforementioned weaknesses in workload prediction and chunk-scheduling and was integrated in a widely-used API for parallel programming. In summary, this work delivers the following main contributions to the state-of-the-art:

- *A novel workload-aware loop scheduling strategy entitled BinLPT*. To enable superior performance and flexibility, our strategy is based on three features: (i) a user-supplied estimation about the workload of the target irregular loop; (ii) a greedy bin packing heuristic to adaptively partition the iteration space into several chunks; and (iii) a hybrid scheme based on the Longest Processing Time (LPT) rule and on-demand scheduling [Graham 1969].
- *An integration of our novel workload-aware loop scheduling strategy into the OpenMP runtime system of GCC*. Our implementation is open-source and publicly available for download, thereby enabling any parallel application that relies on this programming abstraction to seamlessly use our strategy.

To evaluate BinLPT, we carried out a performance analysis using a synthetic kernel and a 3D N-Body Simulations application kernel. We ran experiments on a large-scale Non-Uniform Memory Access (NUMA) machine and we contrasted the performance of BinLPT against the default strategies that are shipped with OpenMP. The remainder of this work is organized as follows. In Section 2, we introduce the LSP and classical loop scheduling strategies. In Section 3, we detail the workload-aware loop scheduling strategy proposed in this work. In Section 4, we present our evaluation methodology. In Section 5, we discuss the experimental results. In Section 6, we contrast related work with ours. In Section 7, we draw the conclusions of our work and discuss some future works.

2. Background

In this section we introduce the background on which this work relies. First, we present the LSP, and then we discuss the classical loop schedulers.

2.1. The Loop Scheduling Problem

The LSP is an instance of the NP-Hard minimization problem for multiprocessor scheduling and it down to partitioning $\widehat{x}$ into n non-overlapping chunks and assigning disjoint sets of these n chunks to the p threads; such that the following are minimized: (i) the number of n chunks that are used to partition the loop iteration space $\widehat{x}$; and the maximum load imbalance between any pair of threads in p. This formulation shows the relation of the four core variables of the LSP: (i) the loop iteration space $\widehat{x}$; (ii) the load of iterations w_k; (iii) the number of chunks n in which $\widehat{x}$ will be partitioned; and (iv) the number of threads p that will process the n chunks in parallel.

Additional variables may also play an important role in a real-world context. For instance, the *scheduling overhead* is an important concern for strategies that assign chunks of iterations to idle threads on-the-fly. If contention in synchronization structures is costly, the irregular parallel loop may face scalability issues [Fang et al. 1990]. The performance of memory-intensive irregular loops can also be severely impacted by data locality, so *memory affinity* can be exploited by loop schedulers when through the use of large chunks [Markatos and Le Blanc 1994].

2.2. Loop Scheduling Strategies

Loop scheduling strategies boil down to one of the following two approaches: *static*, in which loop iterations are assigned to the threads of the parallel application at compile-time; and *runtime*, in which scheduling decisions are made at runtime [Kejariwal et al. 2006]. Static scheduling strategies introduce no runtime overhead, but (i) they are only possible on parallel loops which can have their bounds somehow determined at compile time, and (ii) they are suitable only for parallel loops which feature a compile-time predictable workload. In contrast, runtime strategies are employed to address parallel loops that either do not meet the aforementioned compile-time requirement, or perform computation on a workload that is known only at runtime. In this work, we address the problem of scheduling irregular parallel loops in which the workload is known only at runtime, and in the following paragraphs we discuss the classical scheduling strategies on this scenario.

Pure Dynamic Scheduling (PDS) assigns iterations to threads in unit-sized chunks and on-demand. Whenever a thread becomes idle, an iteration is assigned to it. This strategy achieves good load balancing but at the price of a possibly high runtime overhead.

Chunk Self-Scheduling (CSS) works like PDS, but instead of assigning iterations one by one, it assigns iterations in equally-sized chunks [Fang et al. 1990]. Small chunk sizes deliver good load balancing, but they likely introduce prohibitive runtime overheads. In contrast, large chunk sizes avoid this problem, but may increase load imbalance. When the chunk size is fine-tuned, near-optimum load balancing is achieved [Balasubramaniam et al. 2012], and when it equals to one, this scheduling strategy degenerates to PDS.

Guided Self-Scheduling (GSS) also assigns chunks of iterations to threads on demand, but it dynamically changes their size at runtime (the size of the next chunk is given by the number of remaining iterations divided by the number of threads) [Polychronopoulos and Kuck 1987]. The idea of having a decreasing chunk size is to offer a compromise between achieving good load balancing while reducing runtime overhead.

Factoring Self-Scheduling (FSS) works similarly to GSS, but it differs in the way that chunk sizes are determined [Hummel et al. 1992]. To address the scenarios in which GSS does not perform so well, FSS computes the next chunk size by dividing a subset of the remaining loop iterations (usually half) evenly among the threads. FSS introduces no significant runtime overhead compared to GSS, and it may deliver better performance.

Among the aforementioned loop scheduling strategies, OpenMP offers builtin support for FSS and CSS. The OpenMP community pragmatically refers to them as Guided and Dynamic, respectively. Therefore, we will refer to these strategies using the latter notation, unless otherwise stated.

3. The BinLPT Loop Scheduler

In this section, we present our novel workload-aware loop scheduling strategy. First, we discuss the internals of BinLPT, and then we detail our strategy algorithmically. We implemented BinLPT in libGOMP and we made the enhanced version of this runtime system publicly available[1] under the GPL v3 License. Therefore, any parallel application that is built on top of OpenMP may seamlessly use our scheduling strategy.

3.1. Strategy Internals

BinLPT operates in two phases to deliver load balance to irregular parallel loops, namely *chunk partitioning* and *chunk scheduling*. The heuristics used in each phase are indeed the key features that enable the superior performance of BinLPT.

In the *chunk partitioning* phase, BinLPT splits the iteration space into chunks so as to amortize load imbalance while minimizing the number of chunks that are produced. In this way, runtime scheduling overheads can be reduced and iteration affinity may be exploited efficiently. Indeed, this sub-problem could be optimally solved in pseudo-polynomial time using a dynamic programming algorithm for the Linear Partition Problem. Nevertheless, since loop ranges may grow asymptotically, the overhead incurred by this algorithm makes its use prohibitive. Therefore, BinLPT relies on a workload-aware adaptive technique that takes as input a user-supplied threshold k and works as follows. First it computes the average load ω_{avg} for a chunk based on the workload information and k. Next, it uses a greedy bin packing heuristic that bundles into a single chunk the maximum number of iterations whose overall load does not exceed ω_{avg}.

In the *chunk scheduling* phase, the goal is to come up with a chunk/thread assignment that minimizes load imbalance. Therefore, BinLPT relies on a hybrid scheduling scheme that works as follows. Initially, chunks are statically scheduled to threads using the LPT rule: which assigns the heaviest chunks to the least overloaded threads, and then iteratively assigns lighter chunks to more heavily loaded threads. Next, threads are unblocked and start computing. Then, whenever a thread finishes computing all its chunks, it steals a chunk from other thread that still has work left. This simple scheme optimally handles load imbalance created by both predictable and unpredictable phenomena. Static scheduling based on LPT ensures a $4/3$-approximation scheduling solution to load imbalance incurred by the workload. On the other hand, on-demand scheduling ensures that unpredictable phenomena, such as communication latencies, external load interference, and poor workload estimation, are optimally tackled in a 2-approximative fashion [Graham 1969].

[1] `www.github.com/lapesd/libgomp`

Algorithm 1 BinLPT loop scheduling strategy.

```
1:  function BINLPT(A, k, n)
2:      C ← COMPUTE-CHUNKS(A, k)
3:      SORT(C, descending order)
4:      for i from 0 to n do
5:          T_i ← 0
6:      for i from 0 to | C | do
7:          T_j ← min T
8:          P_{T_j} ← P_{T_j} ∪ {ĉ_i}
9:          T_j ← T_j + ω(ĉ_i)
10:     return P
11: function COMPUTE-CHUNKS(A, k)
12:     j ← 0
13:     C ← empty multiset
14:     ĉ_0 ← empty sequence
15:     ω_avg ← (Σ_{i_j ∈ A} w_j) / k
16:     for i from 0 to | A | do
17:         ĉ_j ← (ĉ_j, A_i)
18:         if ω(ĉ_j) > ω_avg then
19:             C ← C ∪ {ĉ_j}
20:             j ← j + 1
21:     return C
```

3.2. Strategy Design

The BinLPT loop scheduling strategy is outlined in Algorithm 1. In the pictured notation, means that iteration A_i is added to $\widehat{c_j}$. It takes as input three parameters: an array that gives a load estimation of each iteration in the target parallel loop (A), the maximum number of chunks to generate (k) and the number of working threads (n). Then it returns a multiset (P) that states the thread/chunk assignment (*i.e.* P_j is a set containing all chunks assigned to thread j). The algorithm starts by computing chunks according to the greedy bin packing heuristic detailed in the previous section (line 13). Then, it sorts the produced chunks according to their loads (line 14). Next, chunks are statically scheduled following the LPT rule (lines 15 to 20). Later, during the execution, whenever a thread becomes idle, it steals chunks from other threads.

4. Evaluation Methodology

To evaluate BinLPT, we employed a synthetic kernel and an application kernel. The former was used to assess the upper bound performance of our strategy, whereas the latter was employed to enable such analysis in a realistic scenario.

The synthetic kernel performs embarrassingly parallel computations on private variables as proposed in [Bull 1998], and thus it benchmarks the load balancing performance of a scheduler. The kernel takes as input four parameters: the iteration space size, the input workload w, the scheduling strategy, and the number of working threads. The application kernel performs N-Body Simulations and it was chosen for two main reasons. First, it has great importance to the scientific community as a whole, since it finds application on different domains, such as Computation Fluid Dynamics and Molecular Dynamics [Springel et al. 2005]. Second, it is a typical irregular kernel that is frequently studied within the context of loop scheduling [Banicescu 2003, Wang et al. 2012]. The N-Body Simulations kernel that we considered (code-named LavaMD) was extracted from the Rodinia Benchmarks Suite [Che et al. 2009].

Table 1 summarizes the parameters we used in each set of experiments. The workloads are frequently studied by related works, and they were generated using the tool proposed by [Penna et al. 2016]. Problem sizes were chosen so as to reflect the full processing capacity of the experimental platform. Baseline strategies were selected to be

Table 1. Parameters for experiments.

Parameters	Synthetic Kernel	Application Kernel
Workload PDF	Exponential, Gamma and Gaussian	Exponential, Gamma and Gaussian
Loop Size	{384, 768, 1536, 3072, 6144}	$11 \times 11 \times 11$
Chunk Size	Guided {1}, Dynamic {1,2}, BinLPT {288, 384}	Guided {1,2,3}, Dynamic {1,2,3}, BinLPT {384, 576, 768}

consistent with related works. Chunk sizes were selected based on earlier experiments that revealed them to be the optimal values.

For the *synthetic kernel*, we adopted a full factorial experimental design, thereby resulting in 75 different scenarios. For these experiments, we set the number of threads to 192 to reflect the full computational power of the experimental platform. For the *application kernel*, on the other hand, we adopted a fractioned experimental design, where we varied the number of threads from 24 to 192, with a constant step of 24. We considered 27 different scenarios for each experimental configuration. We carried out five replications of each configuration to account for the inherent variance of the measures. For each replicate, the actual order in which individual runs were executed was randomly determined. This approach ensures that experimental results and errors are independent and identically distributed (i.i.d) random variables. In our experiments, the maximum relative standard deviation error (σ/μ) observed was below to 1.0%.

We considered the following performance metrics[2] [Luke et al. 1998, Cariño and Banicescu 2008]: (a) *Parallel Time*, which is the overall execution time of the parallel loop; (b) *Cost*, which is the aggregate time spent to execute the parallel loop, and thus quantifies the waste of processor time; (c) *Performance*, which is the ratio of the total amount of work to the parallel time; (d) *Coefficient of Variance (C.o.V)*, which is the ratio between the standard deviation and the mean execution time of the threads; (e) *Slowdown*, which is the ratio between the execution time of the slowest thread to the fastest one.

All experiments were performed on a SGI Altix UV 2000 machine, which has 24 cache coherent NUMA nodes interconnected through SGI's NUMAlink6 (bidirectional). Each node has an Intel Xeon E5 Sandy-Bridge processor and 32 GB of DDR3 memory. Overall, this platform features 192 physical cores and 768 GB of memory. In our experiments, hyper-threading was disabled and we used a first-touch memory allocation strategy coupled with a compact thread affinity policy to mitigate runtime NUMA effects.

5. Experimental Results

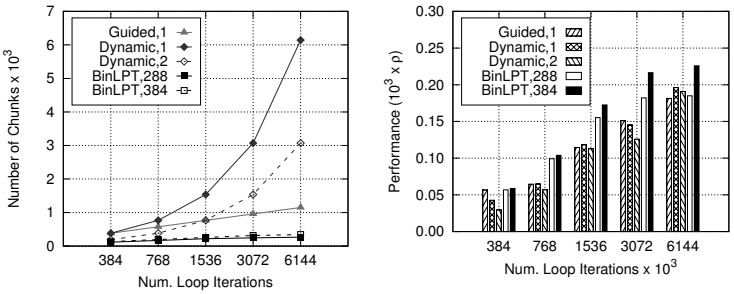
First, we show results for the synthetic kernel, and then we analyze results for the application kernel. All experimental results are publicly available for download[3].

5.1. Synthetic Kernel Results

The actual number of chunks produced by BinLPT varies according to the workload itself and it be fine-tuned by its k parameter. For instance, `BinLPT,288` produces at most $k = 288$ chunks, pragmatically. On the other hand, for the Guided and Dynamic strategies, the number of chunks that are generated depends on the iteration space $|\widehat{x}|$. For the former strategy, the number of chunks grows proportionally to $O(\log |\widehat{x}|)$, whereas for the latter

[2]We will refer hereafter to these metrics using capitalization along with their symbol.

[3]Experimental results available at: `https://doi.org/10.6084/m9.figshare.4742272`



(a) Number of chunks. (b) Performance (ρ) results.

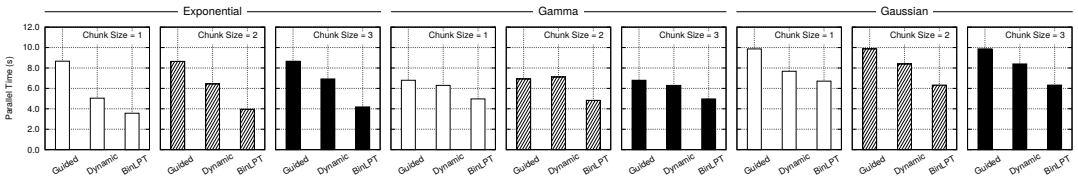
Figure 1. Synthetic kernel benchmarking results for Exponential workload.

strategy it grows with $O(|\widehat{x}|)$. In both latter strategies, the granularity of the chunk-sizes may be fine-tuned according to a parameter b. For instance, `Dynamic,1` will cause Dynamic to use unit-sized chunks ($b = 1$). On the other hand, `Guided,2` instructs Guided to generate chunks which are not smaller than 2 ($b = 2$).

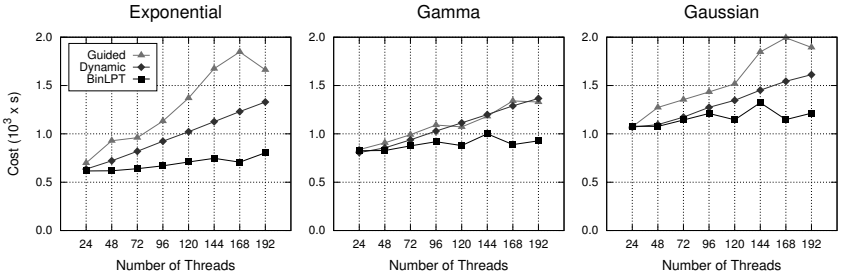
Figure 1(a) presents the number of chunks generated by each strategy for an Exponential-generated workload, when varying the size of the iteration space (*i.e.* $|\widehat{x}|$) at a constant ratio ($2\times$). We observed similar behaviors for the other workloads (*i.e.* Gamma- and Gaussian-based), and thus we omitted them due to space limitations. Overall, the results show that the number of chunks produced by BinLPT are far fewer than the ones produced by both Guided and Dynamic, regardless of the values assigned to parameters k and b. The number of chunks generated by Guided grows linearly and for Dynamic grows exponentially. Since the scheduling overhead of runtime on-demand scheduling techniques depends on the number of chunks [Cariño and Banicescu 2008], it turns out that BinLPT is the most scalable scheduling strategy for the chosen k values. Indeed, if we used larger values for k, the number of chunks produced by BinLPT could be bigger. However, in the performance analysis that follows, we reveal that the values used for k are sufficient for BinLPT to deliver superior performance.

Figure 1(b) presents Performance (ρ) results for a Exponential-generated workload, when varying the size of the iteration space (*i.e.* $|\widehat{x}|$) at the same constant ratio of $2\times$. Overall, BinLPT achieved the best results. The highest Performance (ρ) observed for `BinLPT,288` was for the scenario with 768 iterations, where it delivered 34.75% superior Performance (ρ) than the best configuration for the other two strategies (`Dynamic,1`). The highest Performance (ρ) observed for `BinLPT,384` was also in the same scenario, where it delivered 37.65% superior Performance (ρ).

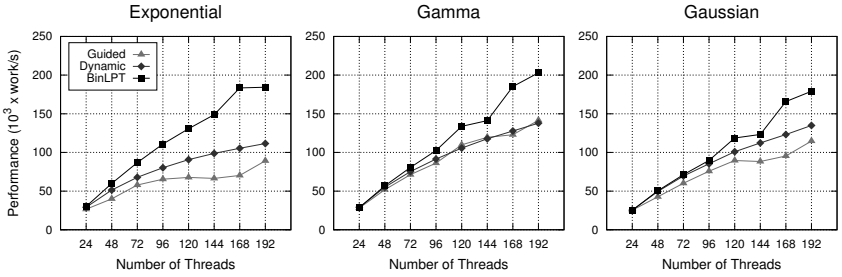
The worst Performance (ρ) observed for BinLPT was for the scenario with 6144 loop iterations and $k = 288$, where we noted 6.09% of Performance (ρ) degradation. Although `BinLPT,288` did perform slightly worse, the weight of chunks generated by BinLPT was not fine-grained enough to amortize load imbalance in this scenario. This behavior is confirmed by Figure 1(a), which shows that BinLPT generated much less chunks than the other strategies for this scenario (`BinLPT,288` generated 266 chunks, whereas `Guided,1`, `Dynamic,1` and `Dynamic,2` generated $4.33\times$, $23.09\times$ and $11.54\times$ more chunks, respectively). Thus, a fair comparison between BinLPT and the other strategies accounts for the equivalence between the maximum number of chunks k produced by our strategy and the number of chunks generated by them. Recall that `BinLPT,288` splits



(a) Parallel Time (τ) vs. chunk size when running with 192 threads.



(b) Cost (γ).



(c) Performance (ρ).

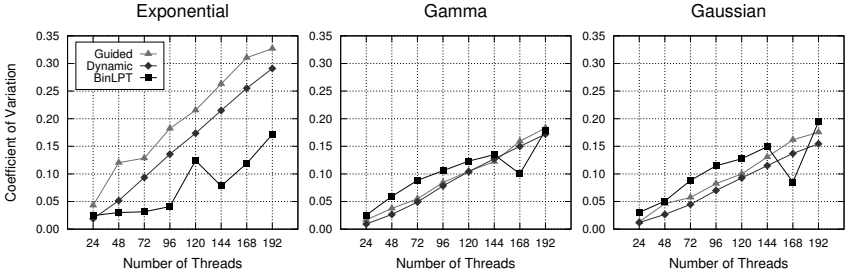
Figure 2. Experimental results for the N-Body Simulations application kernel.

the iteration space in at most 288 variable-size chunks, whereas `Dynamic,1` produces 6144 unit-sized chunks ($b = 1$). Therefore, for instance, a fair comparison would be in the scenario with 768 iterations, between `BinLPT,384` (at most 384 chunks) against `Dynamic,2` ($768/2 = 384$ chunks). In this case, BinLPT would deliver 45.13% better Performance (ρ). We observed similar results for the Gamma- and Gaussian generated workloads (they were omitted due to space limitations), in which BinLPT delivered 29.94% and 32.81% better Performance (ρ) over `dynamic,2`, respectively.

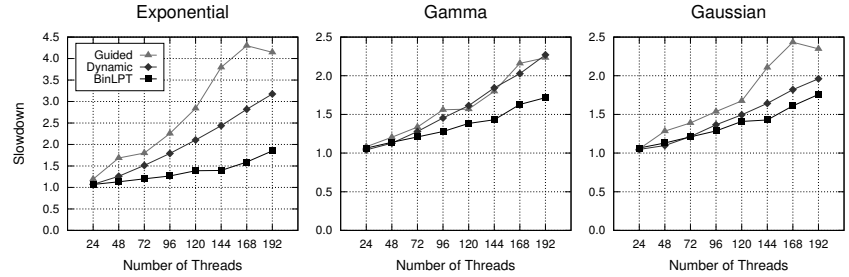
5.2. Application Kernel Results

Figure 2(a) presents Parallel Time (τ) results for the N-Body Simulations application when using 192 threads and varying both the chunk size and the workload. In this plot, the chunk size for the BinLPT means that the parameter k of our strategy was chosen so as it would lead a fair-comparison with Guided and Dynamic in each scenario. Overall, the results unveiled that BinLPT delivers better Parallel Time (τ) regardless the scenario.

Figure 2(b) presents Cost (γ) results when varying the number of threads, for a scenario with the following configurations (worst-case scenario for BinLPT): `Guided,3` `Dynamic,3` and `BinLPT,384`. When analyzing the results, we observed that BinLPT delivers near constant scalability in a large-scale NUMA machine, for all the three workloads. On the other hand, the Cost (γ) scales up quasi-linear for Guided and linear for



(a) C.o.V (γ) results.



(b) Slowdown (χ) results.

Figure 3. Experimental results for the N-Body Simulations application kernel.

Dynamic. Nevertheless, it is worthy to note that Cost (γ) results suggest that BinLPT performs better on the Exponential and Gaussian results. Indeed, we confirmed this finding with Performance (ρ) results showed in Figure 2(c). In the case of Exponential- and Gaussian-generated workloads, BinLPT delivers up to 37.15% (168 threads) and 34.45% (192 threads) better Performance (ρ), when considering Dynamic as baseline. In, contrast, for the Gamma workload, we noted 30% of improvement.

So far, we considered metrics that rely on the performance of the slowest thread executing the parallel loop. Thus, we now analyze the other two metrics that consider the performance of all threads: C.o.V (λ) and Slowdown (χ). When we analyzed C.o.V (λ), we found out that BinLPT presented lower values than Guided and Dynamic on the Exponential-generated workload; and it showed up worst C.o.V (λ) results for the other two distributions. Indeed, these results at glance suggest that BinLPT does not deliver load balancing. Nevertheless, since our strategy did present better overall Cost (γ) and Performance (ρ) results than Guided and Dynamic in all the three workloads regardless the number of threads, we drawn the following conclusion. Even though there is a great difference between the execution times of threads in BinLPT, the actual absolute difference is smaller than it is for Guided and Dynamic. We confirmed this finding by plotting histograms of execution times of threads, when running with 96 to 192 threads. However, we had to omit these plots due to space limitations. This conclusion is also pictured in Figure 3(b), which presents Slowdown (χ) results. This plot evidences that the difference between the slowest and fastest threads in BinLPT is actually smaller than in Guided and Dynamic for all thread configurations. Considering Dynamic as baseline, and when using 192 threads, gains are 45.54%, 26.32% and 12.62% for the Exponential-Gamma- and Gaussian-generated workloads, respectively.

6. Related Work

Targeting time-step applications with irregular parallel loops, Banicescu [Banicescu 2003] proposed Adaptive Weighted Factoring (AWF). In this strategy, the chunk size is dynamically adapted after each step in the application. The newly computed chunk size decreases across the iteration space, likewise in FSS. However, the performance of the threads during the last time-step and their accumulative performance during all the previous ones is additionally considered in this adjustment. To evaluate the performance of AWF, two in-house applications were studied: (i) Laplace's Equation Solver on an unstructured grid using Jacobi's method; and (ii) N-Body Simulations. Pure Static Scheduling (PSS) and FSS strategies were considered as baselines. Experiments were carried out on a synthetically-loaded homogeneous cluster, and the results unveiled that AWF may achieve up to 46% better performance than the baseline strategies. Due to the notable performance of AWF, extensions have been proposed to enable its use on non-iterative applications as well [Cariño and Banicescu 2008]. Nevertheless, the enhanced version of this strategy presented a performance that is comparable to the one achieved by FSS.

To address a broader class of applications, Kejariwal *et al.* [Kejariwal et al. 2006] proposed History-Aware Self-Scheduling (HSS). Unlike AWF, HSS relies on statistical information collected offline via profiling to carry out a smarter scheduling. Based on this extra knowledge, at every scheduling round, HSS computes chunk sizes in a decreasing fashion like FSS, but also considering the load of previously executed iterations and their corresponding actual loads. To assess the performance of HSS, irregular parallel loops extracted from the Standard Performance Evaluation Corporation (SPEC) Benchmarks were studied, and the FSS and AWF strategies were considered as baselines. Experiments were carried out on an in-house simulator, and the results unveiled that HSS may outperform baseline strategies up to 18%.

Based on a similar offline profiling-guided approach to HSS, Wang *et al.* [Wang et al. 2012] introduced Knowledge-Based Adaptive Self-Scheduling (KASS). This strategy works on two phases: a static partitioning phase, and a dynamic scheduling phase. In the first phase, a knowledge-based approach is used to partition iterations of the parallel loop into local work queues of threads, which makes the total workload to be equally distributed to the threads, approximately. In the second phase, iterations on local work queues are scheduling with decreasing sizes likewise in FSS. Each thread gets a chunk from its local queue to execute, and when it finishes the execution of all the chunks in its local queue, it steals chunks from other threads. To evaluate the performance of KASS, two scenarios were studied: (i) parallel loops extracted from the SPEC Benchmarks; and (ii) and three in-house application kernels, namely Over-Relaxation, Jacobi Iteration and Transitive Closure. The GSS, FSS, Trapezoid Self-Scheduling (TSS) and Affinity Self-Scheduling (AFS) strategies were considered as baselines. Experiments were carried out on a Symmetric Multiprocessing (SMP) machine, and the results unveiled that for the parallel loops, KASS is up to 16.9% faster than the baseline strategies. On the other hand, for application kernels, KASS achieved up to 21% better performance.

The workload-aware loop scheduling strategy that we proposed in this work differs from the related strategies discussed above in several points, thereby delivering contributions to the state-of-the-art. First, unlike all these strategies, BinLPT does not rely on a particular workload-estimation technique. Indeed, the HPC engineer is free to couple BinLPT with the one that yields to the best workload estimation for the application. Con-

sequently, the applicability of our strategy is not restricted to time-step applications such as AWF nor to applications that present well-behaved workloads like HSS and KASS, which rely on offline profiling and online regression techniques. Second, even though the aforementioned strategies do use their estimations on the workload to partition the iteration space in several chunks, they lack in using this knowledge to actually schedule chunks of iterations. Alternatively, BinLPT uses a hybrid scheduling scheme based on the LPT rule and on the on-demand scheduling technique. The former handles workload imbalance in a $4/3$-approximative fashion, while the latter deals with unpredictable phenomena in a 2-approximation optimally [Graham 1969]. Finally, existing strategies lack on integrability with applications. For instance, their source-code is not available for download, and their reported algorithmic description is not detailed enough to enable an in-house implementation of them. Our solution, on the other hand, is open-source and is built into GCC's OpenMP runtime.

7. Conclusions

In this work we proposed a novel workload-aware loop scheduling strategy called BinLPT. To enable superior performance and flexibility, our strategy is based on three features. First, it relies on some user-supplied estimation of the workload of the target irregular loop. Such estimation may be derived either from the problem structure or through online/offline profiling, thus enabling maximum flexibility. Second, BinLPT uses a greedy bin packing heuristic to adaptively partition the iteration space in several chunks. The number of chunks to be produced is a parameter of our strategy that may be fine-tuned. Third, it schedules chunks of iterations using a hybrid scheme based on the LPT rule and on-demand scheduling.

We integrated BinLPT into OpenMP, and we evaluated its performance in a large-scale NUMA machine using a synthetic kernel and a 3D N-Body Simulations application kernel. We considered several workloads and we contrasted the performance of our strategy against other strategies available in OpenMP (Guided and Dynamic). Our results unveiled that BinLPT achieves up to 45.13% and 37.15% better performance in the synthetic and application kernels, respectively. As future work, we intend to enhance BinLPT so that it also accounts for data locality when scheduling chunks; and to extend our strategy to emerging manycore platforms which feature a distributed shared memory.

Acknowledgment

This work was supported by FAPEMIG, FAPESC, CAPES, CNPq, CNRS and INRIA.

References

Balasubramaniam, M., Sukhija, N., Ciorba, F., Banicescu, I., and Srivastava, S. (2012). Towards the Scalability of Dynamic Loop Scheduling Techniques via Discrete Event Simulation. In *International Parallel and Distributed Processing Symp. Workshops*, pages 1343–1351.

Banicescu, I. (2003). On the Scalability of Dynamic Scheduling Scientific Applications with Adaptive Weighted Factoring. *Journal of Cluster Computing*, 6(3):215–226.

Banicescu, I. and Velusamy, V. (2001). Performance of scheduling scientific applications with adaptive weighted factoring. In *International Parallel and Distributed Processing Symp.*, pages 791–801.

Bull, J. M. (1998). Feedback Guided Dynamic Loop Scheduling: Algorithms and Experiments. In *International European Conf. on Parallel and Distributed Computing*, pages 377–382.

Cariño, R. and Banicescu, I. (2008). Dynamic load balancing with adaptive factoring methods in scientific applications. *Journal of Supercomputing*, 44(1):41–63.

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 *International Symp. on Workload Characterization*, pages 44–54.

Fang, Z., Tang, P., Yew, P.-C., and Zhu, C.-Q. (1990). Dynamic Processor Self-Scheduling for General Parallel Nested Loops. In *IEEE Transactions on Computers*, volume 39, pages 919–929.

Graham, R. (1969). Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429.

Hummel, S., Schonberg, E., and Flynn, L. (1992). Factoring: a method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101.

Hurson, A., Lim, J., Kavi, K., and Lee, B. (1997). Parallelization of DOALL and DOACROSS Loops - A Survey. *Advances in Computers*, 45:53–103.

Kejariwal, A., Nicolau, A., and Polychronopoulos, C. (2006). History-Aware Self-Scheduling. In *International Conf. on Parallel Processing*, pages 185–192.

Luke, E. A., Banicescu, I., and Li, J. (1998). The optimal effectiveness metric for parallel application analysis. *Information Processing Letters*, 66(5):223–229.

Markatos, E. and Le Blanc, T. (1994). Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400.

Penna, P. H., Castro, M., Freitas, H., Broquedis, F., and Méhaut, J. (2016). Design Methodology for Workload-Aware Loop Scheduling Strategies Based on Genetic Algorithm and Simulation. *Concurrency and Computation: Practice and Experience*.

Polychronopoulos, C. and Kuck, D. (1987). Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439.

Springel, V., White, S. D. M., Jenkins, A., Frenk, C. S., Yoshida, N., Gao, L., Navarro, J., Thacker, R., Croton, D., Helly, J., Peacock, J. A., Cole, S., Thomas, P., Couchman, H., Evrard, A., Colberg, J., and Pearce, F. (2005). Simulations of the formation, evolution and clustering of galaxies and quasars. *Nature*, 435(7042):629–636.

Wang, Y., Ji, W., Shi, F., Zuo, Q., and Deng, N. (2012). Knowledge-Based Adaptive Self-Scheduling. In *International Conf. on Network and Parallel Computing*, number 60973010 in Lecture Notes in Computer Science, pages 22–32.

Policies for Interference and Affinity-Aware Placement of Multi-tier Applications in Private Cloud Infrastructures

Uillian L. Ludwig, Dionatrã F. Kirchoff, Ian B. Cezar, César A. F. De Rose

[1]Faculty of Informatics – Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Avenida Ipiranga, 6681, Prédio 32 - Partenon, RS, 90619-900

{uillian.ludwig, dionatra.kirchoff, ian.cezar}@acad.pucrs.br

cesar.derose@pucrs.br

Abstract. *Multi-tier is one of the most used architectures to create applications and easily deploy them to the cloud. In the literature, there are a great number of research works focusing on the placement of these applications. While some of these works take into consideration performance, most of them are concerned about reducing infrastructure costs. Besides, none of them take into consideration the interference and network affinity characteristics. Therefore, in this work, we create placement policies that aim to improve the performance of multi-tier applications by analyzing the interference and network affinity characteristics of each tier. These characteristics work as a force pushing tiers closer or farther depending on the interference and affinity levels. Moreover, by using these placement policies, we show that multi-tier applications can better utilize the infrastructure, using the same infrastructure but with an improved performance.*

1. Introduction

With the advent of cloud computing and the constant changes in software development, multi-tier applications have become a very popular method of development of every type of application [Christoph et al. 2003]. In this approach, the application is broken down into several modules, called tiers, and each module is responsible for executing a specific task. Moreover, modules may depend on the execution of other modules; thus, they may need to communicate in order to finalize the requested task. This strategy of development brings several advantages, such as the ease of reusability and maintainability. On the other hand, as there are multiple modules to be deployed, it becomes difficult to make placement decisions. This is not a major concern when deploying the application on public clouds since the customers do not have much control over the placement of their applications. However, on a private cloud data center, administrators in general, have total power to decide how to deploy each application, and it is their goal to take the most advantage of the computing resources. Therefore, they must employ placement techniques that place the multi-tier applications maximizing Quality of Service (QoS) and reducing infrastructure costs.

The use of shared environments, such as the virtualized ones, brings the advantage of using a large pool of resources to deploy several distinct applications [Chi et al. 2014]. If only one application would be placed in a physical machine (PM), this application would most likely operate with a nearly optimal performance. However, much of the

resources available would be underutilized. In contrast, when placing multiple applications in the same PM, the resources are better utilized, but there is some performance degradation due to limited resources and performance interference. Consequently, it is important to find a good balance between applications per PM in order to maintain a good performance. Furthermore, when dealing with multi-tier applications, another concern rises, which is the fact that some tiers must communicate with each other in order to answer the requests. This communication is usually done by a physical network, and as the complexity of the applications increases, the pressure on the network also increases, causing the network to be a possible bottleneck on the applications' performance [Ferdaus et al. 2017]. Taking into consideration these two sources of performance degradation, we have the problem seen in Figure 1. In the placement a, we might have performance degradation due to interference, while in the placement b, we might have performance degradation due to network overhead.

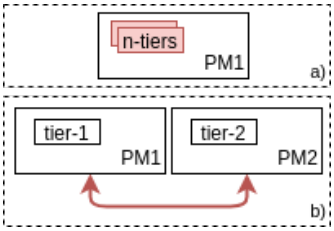


Figure 1. Conflicting placement of a multi-tier application: interference vs. affinity.

Due to this problem faced by private cloud administrators, this paper has as its main goal the creation of a set of placement policies that should lead to better application QoS and good resource utilization. The main factors that will be taken into consideration for the creation of such policies are the resource interference and network affinity of the applications. Moreover, the execution of a multi-tier benchmark with different workload and placement settings was the base the policies.

The rest of this paper is organized as follows: Section 2 goes into detail on performance interference and characteristics of multi-tier applications; Section 3 describes the placement problem and how related works aim to solve it; Section 4 shows the analysis of the multi-tier benchmark execution; Section 5 lists the placement policies; finally, Section 6 concludes the paper and describes the future work.

2. Background

2.1. Performance Interference

Shared environments, such as clouds, clusters, and data centers, are examples of multi-tenant architectures that share their resources to host several related or unrelated applications. These applications may have different purposes, such as the use for scientific research or for running the back-end of software systems. Moreover, each application uses a slice of the resource of the physical machine it is running on. The resource sharing technique varies according to each environment, but the applications may be running as processes, containers or virtual machines (VMs). Furthermore, each resource sharing technique provides different levels of resource isolation. For instance, VMs have a lower interference level between co-hosted VMs, but they have a higher overhead, while containers

and processes have a higher interference level with lower overhead [Felter et al. 2015]. Therefore, it is important to understand the characteristics of the applications when deciding which resource sharing technique is most adequate.

Even though virtualization provides a good level of resource isolation, there is still some degree of resource interference between VMs running on a same physical machine (PM). This interference is due to the fact that each VM uses a slice of the resource available, such as storage, CPU, and memory. However, there are some resources that cannot be sliced, such as disk I/O, network I/O and shared cache [Ibrahim et al. 2011]. Therefore, applications running in the same PM will contend these resources, which leads to an environment that is not fully isolated, and applications may suffer performance degradation [Lin and Chen 2012].

There are several research studies that demonstrate the performance degradation that comes from the use of these multi-tenant architectures. [Chi et al. 2014] tested several workload combinations running in the same PM. The authors showed how depending on the applications that are co-hosted, the performance varies considerably. To illustrate, the application called handbrake, which is CPU intensive, does not suffer much performance degradation when co-hosted with other CPU intensive workloads. On the other hand, the application called dd, which is disk I/O intensive, suffers considerably when co-hosted with other disk I/O workloads. In addition, [Jersak and Ferreto 2016] showed that when running multiple applications that use the same resource intensively on the same host, the performance degrades considerably. They have also found a similar behavior as the previous research, where CPU had a maximum performance degradation of 14%, while for memory and disk I/O this number was as high as 90%. Finally, [Xavier et al. 2015] studied the impact of performance interference on disk intensive applications running on container-based clouds. Their experiments showed that while some combination of workloads running on the same host led to a performance degradation of 38%, they could combine workloads with no performance degradation. Therefore, these studies show how the placement of the applications can impact in the performance of the applications.

2.2. Multi-tier Application Characterization

In these multi-tenant environments, several types of workloads are executed. These workloads include high-performance systems, big data processing, multi-tier applications and so on. The performance requirements of these applications vary, but it is always desirable to ensure that the maximum performance is achieved from the resources available. In this paper, we focus specifically on multi-tier applications, which are characterized by having multiple modules that may communicate in order to execute tasks. The multi-tier architecture is widely used nowadays, but it is not a new concept. The first use of this architecture dates back to the 90s, where client/server applications were commonly used [Data Abstract 2017]. However, this architecture has suffered many changes to adequate to technology evolution and security concerns.

Web and mobile applications heavily rely on the multi-tier architecture as the main method of development [Christoph et al. 2003, Huang et al. 2004]. An important reason is the fact that the multi-tier architecture brings better maintainability and reusability [Huang et al. 2004]. Therefore, tiers may be used to provide services to both web and

mobile systems due to its good reusability, and it is easier to develop and maintain since each tier will be responsible for a specific task. Moreover, a good example of a multi-tier application is the online latex editor ShareLaTeX [ShareLaTeX 2017]. This application contains several tiers that are responsible to execute specific tasks. For example, the web tier is responsible for building the web interface, the real-time tier is responsible for syncing the document changes in real time, the latex compiler tier generates the PDF files, and the spelling checking tier verifies for writing mistakes. Each tier in this application has different resource requirements, and some need higher performance guarantees than others. Thus, it is important to the service provider to ensure that the performance needed for each tier is achieved.

As the applications grow more complex, the number of tiers grow linearly; therefore, it becomes vital to have a good management of the infrastructure [Huang et al. 2004]. This management becomes harder because there are many factors that impact the workload of different applications tiers. The periodicity of the workload is one of these factors since some applications have different number of users during different periods of time such as hours and days of the week. Moreover, the number of users may impact the tiers differently. While for one tier, having a varied number of requests per second will have no impact on the performance, for another tier it might have a high impact. Furthermore, tiers need to communicate through the network in order to finish the user's requests. Thus, it is important to have a good understanding of the characteristics of each tier, making a placement that takes these characteristics into consideration.

3. Problem Statement and State-of-the-Art

Improving the performance of applications by making better placement decisions is widely studied in the literature. Most of these existing strategies use factors such as resource utilization, number of active PMs, and data-center traffic in order to decide the best placement configuration [Masdari et al. 2016]. Moreover, some works are concerned about the interference that is generated by co-hosted applications, while others are concerned about the network affinity of them. However, there is no strategy that takes into consideration both interference and affinity characteristics. Furthermore, in the following paragraphs, we describe the most relevant research studies related to both interference and affinity placement strategies.

A performance-aware placement of virtual machines was created, taking into consideration the interference levels of VMs running in the same PM [Jersak and Ferreto 2016]. The authors have built an interference model that extracts the interference levels of CPU, RAM, and disk I/O intensive applications. Based on this model, they make placement decisions, trying to minimize the interference while keeping the quantity of PMs necessary low. In addition, with the same purpose, VUPIC, which is a placement scheme that takes into consideration the performance isolation and resource utilization in order to decide the best placement, was developed [Somani et al. 2012]. It classifies the VMs in groups of interference, and those VMs which interferes the same resources intensively are placed in different PMs.

Focusing specifically on network interference, a placement scheme was implemented to reduce the performance degradation [Lin and Chen 2012]. Since network I/O is a resource that cannot be sliced, whenever multiple VMs are using it, they will con-

tend for the same resource, leading to performance loss. Thus, they favor placing VMs that stress the network in different PMs. Moreover, using a different approach, a network affinity aware placement was built [Su et al. 2015]. In this approach, the authors group VMs that have network affinity, and try to place them in the same PM; thus, reducing the network overhead across physical networks. Besides, [Ferdaus et al. 2017] considered the network affinity, and created a placement of multi-tier applications in the cloud. They could reduce network costs, as well as keeping the server well utilized.

It is clear the importance of both resource interference and network affinity strategies. However, none of the placement strategies consider both characteristics at the same time. The main reason for this is due to the fact that these are opposed characteristics, and it is not simple to find the best trade-off. For this reason, this paper focuses on reducing the resource interference while considering the network affinity. Moreover, it should keep the quality of service (QoS) high, while maintaining the number of necessary PMs low.

4. Performance Analysis

As a first step to perform the performance analysis, we had to decide which multi-tier application we would use as our target. We analyzed the characteristics of the benchmarks that are used in other research projects in the literature, and we found that none of them were suitable for our purposes. We needed a multi-tier benchmark, but most of the benchmarks used to characterize performance interference are not implemented with this architecture. Most of them execute a large task, which usually takes up to several minutes to finish and stress a specific resource. However, the tasks in multi-tier applications take a short time to finish, usually less than a second. The high load comes from the high number of requests arriving at the same time, rather than the execution of a big task. Thus, this type of benchmarks would not be useful for characterizing the multi-tier applications. Furthermore, there is a widely used multi-tier benchmark called RUBiS, which simulates an e-commerce application [OW2 Consortium 2001]. RUBiS, however, is not very flexible when it comes to customization of resource utilization, and its use is more adequate for use cases rather than to perform deep analysis.

Given the lack of an appropriate benchmark, we decided to build a multi-tier benchmark that would allow us to do fine grained customization of both resource and network utilization. We built a multi-tier benchmark that allows to use up to n tiers, where each tier might be CPU or disk intensive or non-intensive. Moreover, it is flexible and more types of tiers, such as memory or cache intensive, may be added. Besides the ability to modify the characteristic of each tier, the benchmark allows modifying the request size for each arriving request, which will directly affect the network performance. Thus, this benchmark allows us to easily test different combinations of workloads. The architecture of the benchmark is shown in Figure 2, and the source code is found in a Github repository[1].

In order to generate load in the application, we used a tool called Artillery [Shoreditch Ops Ltd 2017]. It allows us to create virtual users, and each user will execute HTTP requests in the server. Since each request by itself does not take a long time to finish, the best way to impact the resource utilization and the performance is by varying

[1] `https://github.com/uillianluiz/node-tiers`

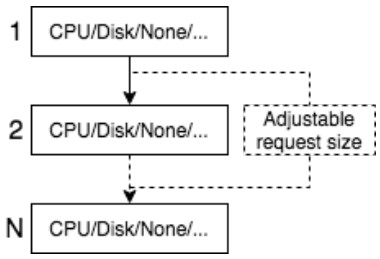


Figure 2. Multi-tier architecture of the implemented benchmark with N tiers.

the quantity of arriving requests. Thus, we configured Artillery to vary the request demand from 1 to 300 requests per second. Moreover, at the end of the processing, Artillery provides a detailed report, listing response times, requests per second processed, and so on.

The execution environment consisted of two physical machines with: 2x Intel(R) Xeon(R) CPU X6550, 64GB RAM, 128GB of storage and a gigabit ethernet connection between both of them. Each tier was set to use the maximum of 4 CPU and unlimited RAM. Moreover, we set two request size configurations for the communication of each tier: 1KB and 512KB.

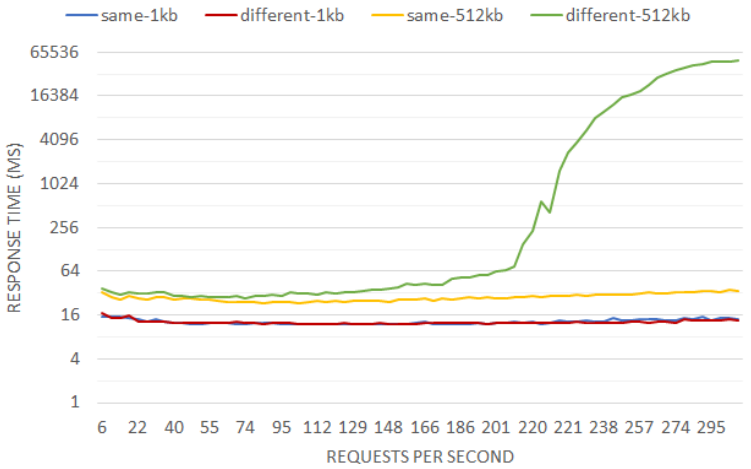


Figure 3. Response time of application CPU-CPU while varying the workload. The lines are in log scale.

Figure 3 shows the performance of an application consisted of two CPU intensive tiers. The response time axis is shown in logarithmic scale for better visualization. It can be noticed that the execution with higher request size (512KB) had a worse performance as compared with the lower request size (1KB). This is a natural behavior since the higher the request size is, the more pressure it puts on both operating system and network. Additionally, while the request rate was low, the performance for all executions remained stable. However, as the request rate increased, the execution with high request size running in different hosts suffers performance degradation. In this case, the network becomes flooded with many requests, and as the network bottleneck is reached, the response time increases exponentially. On the other hand, while running with same request size, but in

same hosts, there is no impact on the performance. Therefore, we can conclude that the network affinity is the most impacting factor when dealing with CPU intensive tiers.

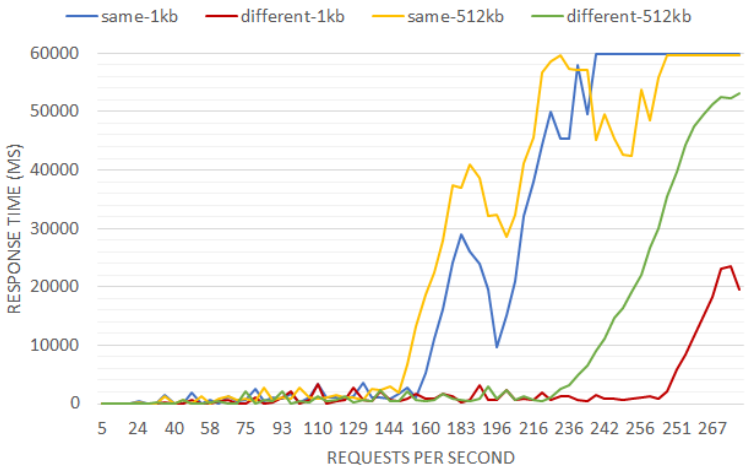


Figure 4. Response time of application Disk-Disk while varying the workload.

Figure 4 presents the response time of the application that had two disk I/O intensive tiers. The response time kept acceptable while the workload was low. However, as the workload increased, the application presents a different behavior from the one seen in the CPU intensive application. All four executions of this application have performance degradation, but this degradation comes earlier in the executions that run the tiers on the same host. Furthermore, the network affinity also has an impact on the performance, and the higher the request size is, the higher the impact is. As a conclusion of this execution, disk I/O intensive applications tend to suffer more from the interference of co-hosted tiers, but there is still impact generated from the network affinity levels.

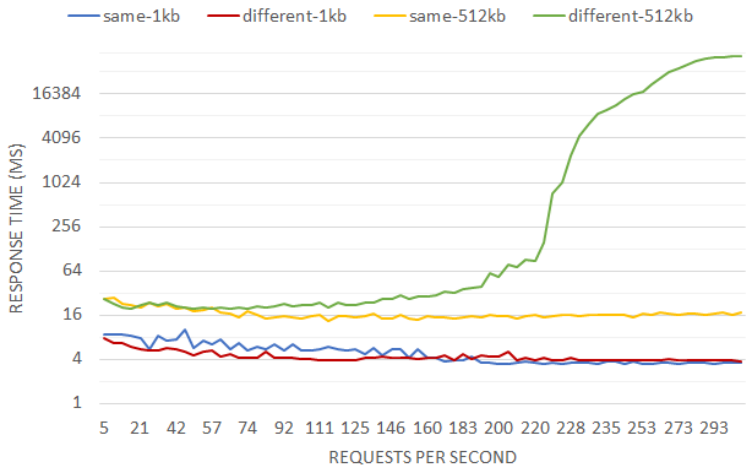


Figure 5. Response time of application Non-Intensive-Non-Intensive while varying the workload. The lines are in log scale.

Figure 5, which contains the execution of an application with two non-intensive tiers, shows a similar behavior to the one seen in the CPU intensive application. This

is due to the fact that both applications are not highly affected by interference, and the characteristic that generates the most impact is the network affinity.

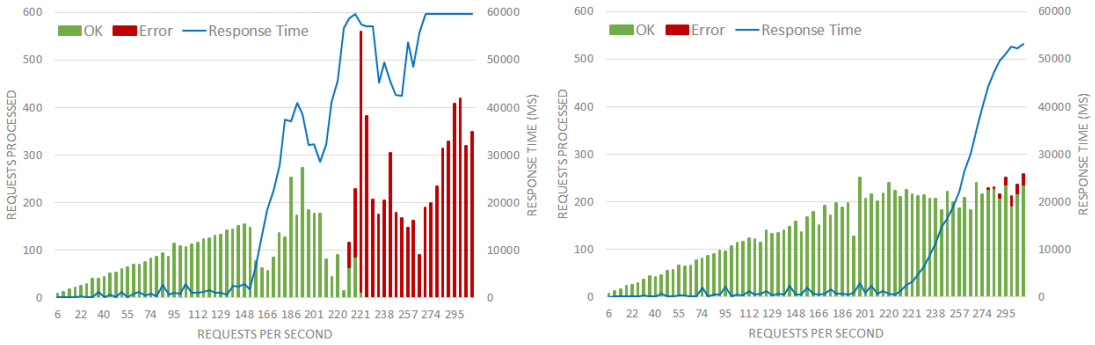


Figure 6. Impact of increasing the request rate in the response time and request status of the execution of the DISK-DISK application with 512KB of request size.

Going further into the analysis of these executions, Figure 6 shows how the requests' status was affected by the increase in the request rate. It shows the impact of the execution of the disk intensive application with 512kb of bandwidth. The chart on the left shows the results of the execution in the same host. It can be noticed that while the request rate was low, the response time was constant and the number of requests with OK status was growing linearly. However, as the request rate surpassed 150, the response time started to grow rapidly, the requests started taking longer to finish, and, finally, all of them were failing. In contrast, the chart on the right shows the execution of the same benchmark configuration but running in different hosts. As we saw in the previous analysis, the execution on different hosts had a better performance. Here, we can see that the effect on the request status matched the response time, and requests only failed when the request rate was above 270.

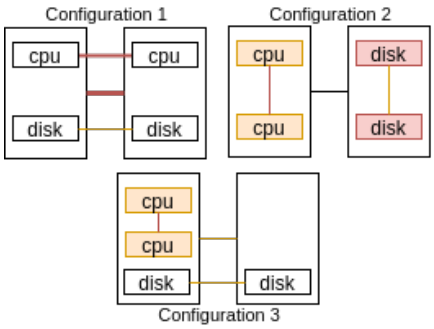


Figure 7. Different configurations of execution of two multi-tier applications.

In addition to analyzing the placement of one multi-tier application, Figure 7 shows three different placement configurations of two multi-tier applications. One application has two CPU intensive tiers, using 512KB of request size. The other one has two disk I/O intensive tiers, using 1KB of request size. For this experiment, we setup three configurations, where each one favors different characteristics. The configuration 1 favors the reduction of the interference by placing the tiers that stress the same resource

in different PMs. The configuration 2 favors the network affinity by placing the tiers that communicate in the same machine. Finally, the configuration 3 tries to find the best compromise between interference and network affinity by placing the application with high affinity in the same PM and the application with high interference in different ones.

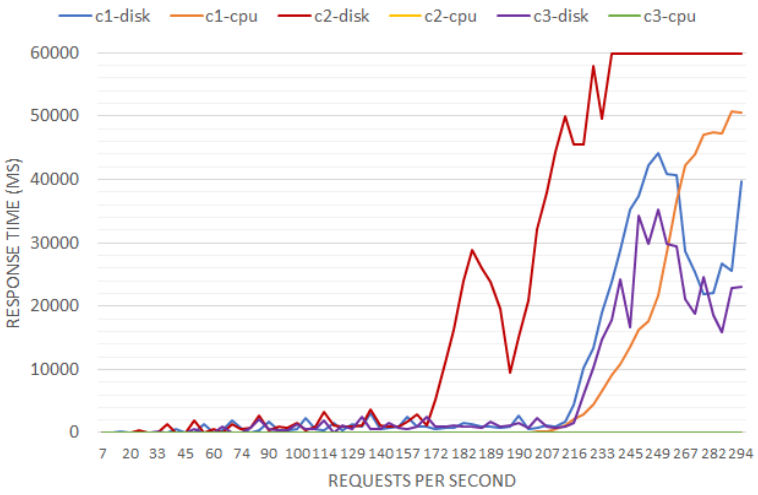


Figure 8. Response time of three combinations of placement of two multi-tier applications in two physical hosts.

The response time of the execution of the three configurations is shown in Figure 8. As seen in the previous experiments, the response time for lower request rate did not vary in the different configurations of placement. The configuration 1, which favors the interference, had a poor performance for both applications. In this case, since the CPU intensive application has a high network affinity, and the tiers were placed in different hosts, the performance is degraded. Moreover, this placement, in theory, should contribute to the performance of the disk I/O intensive application. However, it can be noticed that it also degrades. The reason behind this fact is a network interference generated from the CPU intensive application. Therefore, since the tiers of the CPU intensive application were placed in different PMs generate performance degradation on both applications placed in the infrastructure.

In addition, in the configuration 2, which favors the network affinity, the CPU intensive had no performance degradation since its tiers were placed in the same PM and there was no network overhead. On the other hand, the disk I/O intensive application had a high performance degradation due to interference, and it presented the worst performance of all executions. Finally, in the configuration 3, which tries to find the best trade-off between affinity and interference, the best performance was achieved. Even though in this configuration the quantity of tiers per host was not well-balanced, it achieved the best overall performance. The disk I/O intensive application still had performance degradation, but it was the lowest among all configurations. Therefore, these experiments showed how a better performance can be achieved by taking into consideration both interference and network affinity characteristics.

5. Placement Policies

Taking the results of the experiments into consideration, we may proceed and create a set of placement rules. As seen in the experiments, the workload has a great impact on the performance of the application. For this reason, the following rules focus on maximizing the workload that the application is able to handle without having performance degradation.

R1. Tiers that interfere the same resource should be placed in different hosts. When there are few servers available, PMs might have to host tiers that interfere the same resource. However, it should be given preference for separating the tiers that generate the most performance degradation. In this case, disk I/O tiers have a strong repulsion, while CPU intensive tiers have a weak repulsion.

R2. Tiers that have network affinity should be placed in the same PM. When all the resources of a given PM are being used, it should be given preference to place in the same PM the tiers that have higher affinity, i.e. higher request size. Moreover, the higher the affinity is, the more attraction force it generates.

R3. Tiers that do not interfere nor have network affinity may be placed anywhere in the infrastructure. There is no force pushing the tiers.

R4. Tiers that interfere and have network affinity should follow a sub-set of rules. These sub-rules extract the best trade-off, which should generate a placement that leads to the best QoS possible.

- R4a. CPU intensive tiers should be placed in the same PM. The attraction force generated by affinity is stronger than the repulsion force generated by the interference.
- R4b. Disk I/O intensive tiers should be placed in separate PMs. The performance degradation that comes from interference leads to a stronger repulsion than the attraction generated by the network affinity.

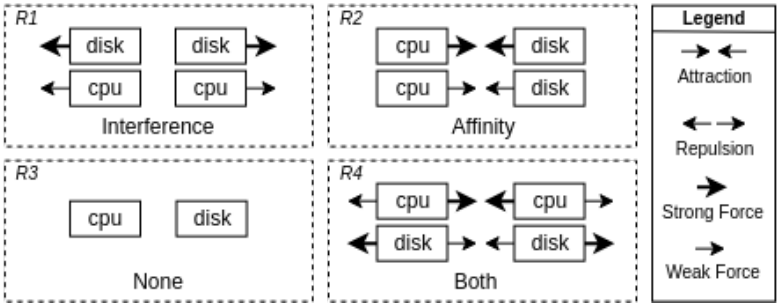


Figure 9. Illustration of the placement policies based on the forces pushing them. While interference repel tiers to different PMs, affinity attracts them to the same one.

Figure 9 shows an illustration of each rule. It demonstrate the forces that affinity and interference put in the tiers, attracting them to the same PM (affinity) or repelling them to different PMs (interference). The line width represents the strength of the force pushing them. Moreover, these rules would be enough if the infrastructure was able to scale out and up without any restrictions. However, such infrastructure does not exist, so it is important to think of how to deal whenever the resources bottleneck is reached. In

this case, a way to deal with this problem is to use a business rule that gives the priority of each tier. For example, a slowdown in a tier that handles logs is less detrimental than a slowdown in a tier that handles users authentication. Therefore, tiers with less priority may be placed in the same PM even with interference, or in different PMs even with network affinity since this placement will not directly affect the business. However, tiers with high priority must follow the placement rules for achieving better QoS.

6. Conclusion and Future Work

The multi-tier architecture is undeniable one of most important methods of development of applications from all computing segments. Its wide utilization brings some important concerns related to how to achieve better performance. Besides optimizing the applications' code itself, the best way to achieve such performance is by better utilizing the infrastructure resources, which may be achieved by using smart placement techniques. Through the experiments, it was demonstrated that different placements have a great impact on the applications' performance, and that the characteristics of resource interference and network affinity should be carefully analyzed in order to decide the best placement.

By observing the difference in performance while executing the multi-tier benchmark in different placement settings, we were able to create a set of placement policies. These policies are important on determining the best placement configuration of multi-tier applications, leading to a better performance while keeping the same infrastructure. Besides, the multi-tier benchmark has a good potential, and it can be used with many purposes, such as to characterize the environment and to test placement settings.

During the experiments, we only considered CPU and disk I/O, but as future work, we plan to expand the multi-tier benchmark adding memory and cache intensive tiers. Therefore, we may be able to make decisions based on a broad type of resource intensive applications. Moreover, in addition to the placement policies, we plan to create a placement model that would receive the interference and affinity levels, and return the best placement setting for the tiers. This model will have as its base the results of this work and the experiments with the other resource intensive tiers that we will implement. After that, the model will be used to build a placement algorithm, which can be used to automate the placement process.

7. Acknowledgments

This work is sponsored by Dell Technologies Inc. for research efforts under Dell Monitoring Project. The views expressed in this paper are those of the authors and does not reflect the official policy or position of our sponsor.

References

Chi, R., Qian, Z., and Lu, S. (2014). Be a Good Neighbour: Characterizing Performance Interference of Virtual Machines Under Xen Virtualization Environments.

Christoph, H., Heinz, S., and Ralf-Detlef, K. (2003). The Distributed Architecture of Multi- Tiered Enterprise Applications. In *A Middleware Architecture for Transactional, Object-Oriented Applications*, chapter 3, pages 15–40. FREIEN UNIVERSITÄT BERLIN, Berlin.

Data Abstract (2017). Why multi-tier? `https://docs.dataabstract.com/Introduction/WhyMultiTier/`. Accessed: 2017-05-24.

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 171–172. IEEE.

Ferdaus, M. H., Murshed, M., Calheiros, R. N., and Buyya, R. (2017). An Algorithm for Network and Data-aware Placement of Multi-Tier Applications in Cloud Data Centers. *Journal of Network and Computer Applications (JNCA)*.

Huang, D., He, B., and Miao, C. (2004). A Survey of Adaptive Resource Management in Multi-Tier Web Applications. *IEEE Communications Surveys & Tutorials*, 16(3):1574–1590.

Ibrahim, S., He, B., and Jin, H. (2011). Towards pay-as-you-consume cloud computing. *Proceedings - 2011 IEEE International Conference on Services Computing, SCC 2011*, pages 370–377.

Jersak, L. C. and Ferreto, T. (2016). Performance-aware server consolidation with adjustable interference levels. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 420–425. ACM.

Lin, J.-W. and Chen, C.-H. (2012). Interference-aware virtual machine placement in cloud computing systems. In *Computer & Information Science (ICCIS), 2012 International Conference on*, volume 2, pages 598–603. IEEE.

Masdari, M., Nabavi, S. S., and Ahmadi, V. (2016). An overview of virtual machine placement schemes in cloud computing. *Journal of Network and Computer Applications*, 66:106–127.

OW2 Consortium (2001). RUBiS. `http://rubis.ow2.org/`. Accessed: 2017-02-24.

ShareLaTeX (2017). Sharelatex. `https://github.com/sharelatex/sharelatex`. Accessed: 2017-07-24.

Shoreditch Ops Ltd (2017). Artillery. `https://artillery.io/`. Accessed: 2017-06-05.

Somani, G., Khandelwal, P., and Phatnani, K. (2012). Vupic: Virtual machine usage based placement in iaas cloud. *arXiv preprint arXiv:1212.0085*.

Su, K., Xu, L., Chen, C., Chen, W., and Wang, Z. (2015). Affinity and conflict-aware placement of virtual machines in heterogeneous data centers. In *Autonomous Decentralized Systems (ISADS), 2015 IEEE Twelfth International Symposium on*, pages 289–294. IEEE.

Xavier, M. G., De Oliveira, I. C., Rossi, F. D., Dos Passos, R. D., Matteussi, K. J., and De Rose, C. A. F. (2015). A performance isolation analysis of disk-intensive workloads on container-based clouds. *Proceedings - 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015*, (February):253–260.

Intrinsics-HMC: An Automatic Trace Generator for Simulations of Processing-In-Memory Instructions

Aline Santana Cordeiro, Tiago Rodrigo Kepe, Diego Gomes Tomé, Eduardo Cunha de Almeida, Marco Antonio Zanata Alves

[1]Department of Informatics – Federal University of Paraná

{asc12, trkepe, dgtome, eduardo, mazalves}@inf.ufpr.br

Abstract. *Processor-in-Memory (PIM) architectures, such as the Hybrid Memory Cube (HMC), are emerging nowadays as a solution for processing large amount of data directly inside the memory. In this area, several researchers are proposing and evaluating new instructions and new PIM architectures. For such evaluations, trace-driven simulators, as the Simulator of Non-Uniform Cache Architectures (SiNUCA), are commonly used in order to model these new proposed systems. Such simulators provide fast prototyping of new architectures, while it requires the researcher to write simulation traces manually when evaluating new Instruction Set Architecture (ISA) proposals, which is an time consuming and error prone task. In this work, we propose a methodology for fast generation of simulation traces focused on HMC architecture, which consists on a high-level Intrinsics-HMC library and a modification inside the trace-generator tool from SiNUCA. Our proposal enables the researchers to write high level code in C/C++ languages using our library, which mimics the behavior of HMC instructions. These codes can be compiled and executed in traditional x86 architectures for verification. After ensure the code is correct and working, the user can use our modified version of SiNUCA-Tracer to translate HMC functions into HMC instructions know by the simulator, providing a convenient solution to generate traces and fast simulations of new PIM architectures. Results using the proposed technique applied on database application kernels show the correct translation and simulation of new HMC instructions using SiNUCA.*

1. Introduction

Processor-in-Memory (PIM) architectures [Patterson et al. 1997], [Elliott et al. 1999], [Balasubramonian et al. 2014], as the new Hybrid Memory Cube (HMC), are emerging in the last few years after the release of 3D-stacking technologies [Olmen et al. 2008]. The HMC presents high parallelism between the Dynamic Random Access Memory (DRAM) banks, ensuring low average latency during high pressure in memory (memory bursts). The HMC also supports PIM in order to mitigate data movement between memory and processor, where processing occurs in the same chip from the memory. Thus, many researchers are evaluating performance and energy consumption of existing HMC [Jeddeloh and Keeth 2012], [Khalifa et al. 2013], [Hadidi et al. 2017] or proposed new PIM architectures [Pugsley et al. 2014], [Alves et al. 2016].

In general, the processor designers and researchers relies on full-system or trace-driven simulators to evaluate performance of new Instruction Set Architecture (ISA) and architectural components. Full-system simulators require executable binaries compiled

for the specific ISA to be simulated, requiring thus a compiler ready for such new systems. On the other hand, trace-driven simulators are more flexible and deterministic, requiring only the simulation trace, which contains the instructions recognized by the simulator and the dynamic execution order of such instructions. The generation of traces is usually performed automatically using binary instrumentation tools for the existing ISA. However, for new PIM architectures, the instruction traces must be manually generated, using many times, the help of scripts to generate the dynamic traces. This manual task is error-prone and can demand a considerable amount time, depending on the complexity of the program to be simulated.

In this context, the main objective of this paper is to propose a method that allows automatic generation of simulation traces that uses HMC instructions directly from a high-level compiled program. We developed a library called Intrinsics-HMC that provides a series of functions that emulate the HMC behavior, based on the HMC specification version 2.1 [HMC Consortium 2017]. This library was written in C/C++ and uses x86 instructions only, so it can be normally linked, compiled and executed just to assure its correct operation. After the developer validates the code, the trace generator can be used to identify the HMC functions in the Intrinsics-HMC library and convert them to HMC instructions recognizable by the simulator. In order to demonstrate our translations of HMC functions and simulate the traces generated automatically, we used the Simulator of Non-Uniform Cache Architectures (SiNUCA) [Alves et al. 2015]. We also used the SiNUCA-Tracer and the dynamic binary instrumentation tool *Pin* from Intel to extract the simulation traces from the full execution of two database kernels developed in C++ language making use of our Intrinsics-HMC library.

The rest of this paper is organized, as follows: Section 2 details the basic concepts about HMC and SiNUCA. Section 3 explains how execution traces are generated, introducing the Intrinsics-HMC and the SiNUCA-Tracer. Section 4 analyzes the simulation results and Section 5 presents some related work. Finally, Section 6 concludes this paper and proposes future directions.

2. General Concepts

In this section, we present basic concepts about Hybrid Memory Cube (HMC) and an overview about the Simulator of Non-Uniform Cache Architectures (SiNUCA).

2.1. Hybrid Memory Cube – HMC

HMC is a memory device formed by up to 8 stacked layers of Dynamic Random Access Memory (DRAM) and the base is a logic layer, as illustrated in Figure 1. The HMC is logically partitioned in 32 vaults. Each vault has a dedicated memory controller located in the logic layer and up to 16 DRAM banks (distributed among the layers of DRAM) connected together using Through-Silicon Vias (TSVs) [Olmen et al. 2008].

The HMC presents high parallelism for accessing data [HMC Consortium 2017], [Jeddeloh and Keeth 2012], [Pawlowski 2011]. Theoretically, the HMC can fetch data from the 32 different vaults at the same cycle, reaching a maximum theoretical bandwidth of 320 GB/s [Jeddeloh and Keeth 2012], which is 25% higher than High Bandwidth Memory (HBM) version 2 (256 GB/s) [Kim and Kim 2014]. Different from Double Data Rate (DDR) 3 memories, which transmit 64 bits per channel, the HMC uses

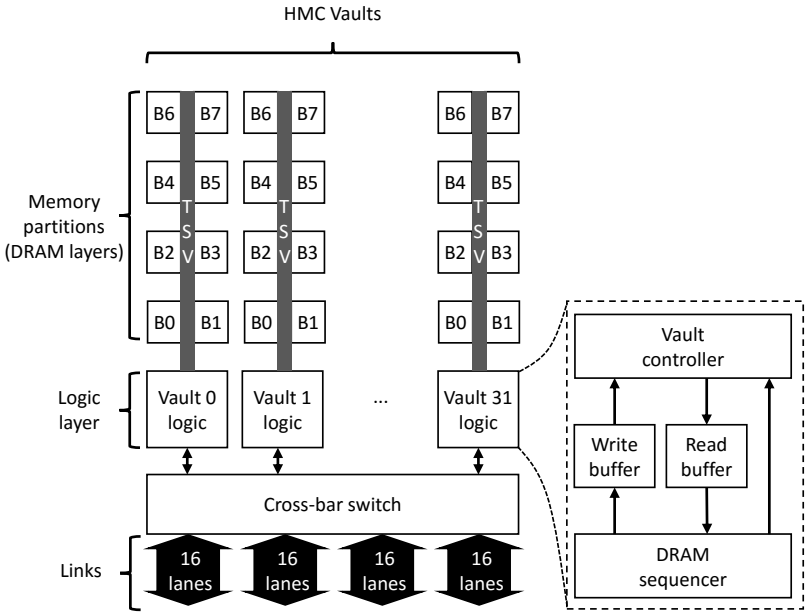


Figure 1. HMC block diagram with 32 vaults each one with 8 banks.

4 serial links formed by 16 full-duplex lanes each. These serial links can reach high frequencies with low interference during data transmissions [HMC Consortium 2017], [Thanh-Hoang et al. 2014]. Internally, there is a crossbar switch that allows these links to transfer data to/from any vault.

HMC supports read and write requests from 16 bytes up to 256 bytes. It also supports arithmetic and binary instructions that operates over 8 or 16 bytes. During the fetch and execution of HMC instructions, the processor treats these instructions the same way as the ordinary memory read or write requests. Thus, the processor fetches, decodes, computes the address and sends the instruction to the HMC. The instructions may also have an immediate operand to be used in the operation. This immediate is sent together to the HMC. When an instruction arrives in the HMC, it is forwarded to the vault responsible for that indicated memory address. The logic layer then interprets each instruction, fetching data and operating over it. Depending on the HMC instruction, the result is updated in the same memory address or sent back to the processor.

HMC can save up to 70% of energy compared to DDR3-1333 memory, presenting a theoretical speedup of 15× [HMC Consortium 2017, Jeddeloh and Keeth 2012, Pawlowski 2011]. However, is not clear if all kind of applications can benefit by current HMC instructions. In this way, studies about Instruction Set Architecture (ISA) extensions are important to evaluate new architectural components.

2.2. Simulator of Non-Uniform Cache Architectures – SiNUCA

During the evaluation of new processor architectures, only simulation represents a viable solution for designers, as the system to be evaluated is too complex to be handled by analytical models, and highly expensive to be prototyped [Jain 1990]. Thus, most computer architects use simulation tools. In contrast to full-system simulation, the trace-driven simulators do not require to actually executing the application instructions during

Table 1. Format of the SiNUCA input traces.

Static Trace

```
#main
@1
MOV 1 0x95727 4 1 14 1 34 14 0 1 0 0 0 0 0 0
#main
@2
MOV 8 0x95717 3 1 14 1 65 14 0 1 0 0 0 0 0 0
ADD 1 0x95720 3 2 14 65 1 34 14 0 1 0 1 0 0 0
ADD 1 0x95723 4 1 14 1 34 14 0 1 0 1 0 0 0
CMP 1 0x95727 4 1 14 1 34 14 0 1 0 0 0 0 0 0
JBE 7 0x95731 2 2 35 34 1 35 0 0 0 0 0 1 0 0
```

Dynamic Trace

```
1
2
2
```

Memory Trace

```
R 4 0x1701448 1
#
R 4 0x1701448 2
R 4 0x1701452 2
W 4 0x1701452 2
R 4 0x1701448 2
W 4 0x1701448 2
R 4 0x1701448 2
#
R 4 0x1701448 2
R 4 0x1701452 2
W 4 0x1701452 2
R 4 0x1701448 2
W 4 0x1701448 2
R 4 0x1701448 2
```

the simulation. In fact, they just need to consider the behavioral details (algorithmic) and microarchitectural latencies for the given traced application. These simulators use execution traces of real applications. These traces are formed by one or multiple files that contains the flow of instructions observed during the program execution. The traces can be generated manually by researchers or automatically by binary instrumentation tools.

In this work, we use SiNUCA, which is a validated cycle-accurate, trace-driven simulator [Alves et al. 2015]. SiNUCA is based on x86 architecture and simulates the execution of mono and multi-threaded applications. It provides an adjustable number of out-of-order processor cores, modeling technologies such as Non-Uniform Cache Architecture (NUCA), Non-Uniform Memory Access (NUMA), Network-on-Chip (NoC) and DDR. SiNUCA also simulates components used in current state-of-the-art architectures, such as data prefetchers, non-blocking cache memories, detailed DRAM memory controller and branch predictors.

SiNUCA splits the simulation traces in three types: static, dynamic and memory, as illustrated in Table 1. The static trace consists of instructions formed by asm code, SiNUCA opcode number, size, read and write registers and other flags. The instructions are grouped in basic blocks, indicated by "@". The dynamic traces contain the sequence calls of basic blocks (from the static trace) performed by the application during normal execution. The memory traces contain the memory address and the instruction size for each memory access performed by the application.

3. The Trace Generator

Emulators and binary instrumentation tools are commonly used to generate trace for trace-driven simulators. Binary instrumentation tools such as Pin, has fast execution and low overhead as advantage. However, these tools depend on complete execution of an application in a real machine for generating an execution trace. Therefore, it becomes very difficult to generate traces of nonexistent instructions in current architectures. Emulators could also be used to generate such traces, but it would require modifications in it to recognize the new ISA as well a new compiler for such architecture. It is clear then, that a new methodology for fast and accurate trace generation is required. In this section, we describe the Intrinsics-HMC library and the trace generation with the SiNUCA-Tracer and Pin tools.

3.1. Proposal Overview

Figure 2 presents the steps required to generate simulation traces referred to HMC ISA. The Intrinsics-HMC library is composed of functions that mimics the behavior of HMC instructions [HMC Consortium 2017]. These functions are called in C/C++ programs and can be successfully compiled to the binary files using the x86 ISA. This binary is later executed with SiNUCA-Tracer (part of our proposal) and the *Pin* instrumentation to generate execution traces. During this trace generation phase, all occurrences of Intrinsics-HMC functions are converted in simulation HMC instructions.

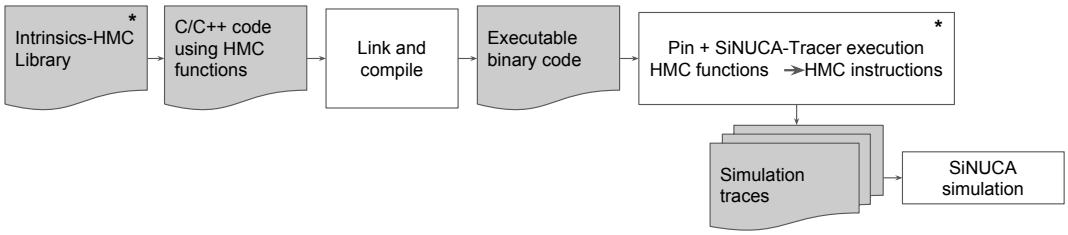


Figure 2. Sequence of steps to generate the simulator input traces (* indicates our main contributions).

3.2. Intrinsics-HMC Library

The logic layer of HMC devices can execute arithmetic instructions and bitwise operations. Therefore, in this section we present functions for creating the Intrinsics-HMC library in order to reproduce the behavior of HMC statements. The types of implemented functions, together with the description of their functionalities are listed below, and in this case, the read and write instructions are not implemented in the library because their traces are generated from the traditional instructions (i.e., load or store requests) normally decoded and executed by the processor being simulated. We split the HMC instructions into four classes of operations, as follows:

Arithmetic: the *sum (add)* and the *increment (inc)* operations. The *add* operation is performed over the operand coming from the DRAM and the immediate, both informed in the HMC instruction. This *add* operation works either with $2\times$ 8-bytes or with $1\times$ 16-bytes operands. The *inc* operation uses only one operand to read a value from the DRAM, increment it by one, and then store it back.

Bitwise: the *bit write (bwr)* and the *swap* operations. The *bwr* operation sends inside the immediate two fields, the bit mask and the write data. For a given address, two operands of 8-bytes each will be loaded and using the bit mask, only specific bits will be updated. In the *swap* operation, the 16-bytes immediate is written to the memory, and the old value is returned to the processor.

Boolean: *and*, *nand*, *or*, *nor* and *xor* operands are used between memory operations and immediate addresses.

Comparison: There are four operations in this class, the *compare and swap if greater than (casgt)*, *compare and swap if less than (caslt)*, *compare and swap if zero (caszero)* and *equal to (eq)*. These operations perform over $1\times$ 8-bytes or $1\times$ 16-bytes operands.

Notice that each operation class supports different operand formats and sizes. There are also variations in the return type sent back to the processor by the HMC. For

further details regarding these operations please refer to the HMC specification version 2.1 [HMC Consortium 2017]. The Intrinsics-HMC library, written in C++ language, covers all the functions described in specification and uses a data type standard to reproduce the HMC operations, described in table 2.

Table 2. Intrinsics-HMC library data types standard.

Data type	Description
__h16l1	Data type equivalent to a short unsigned
__h64l1	Data type equivalent to a long unsigned
__h64l2	Data type equivalent to a vector of 2 log unsigned
__h128ll1	Data type equivalent to a long long unsigned

We define the new data types starting with *'__h'* to denote that this data type refers to the HMC ISA. The number in sequence indicates the data length in bits (16, 64 or 128), together with the letter *'l'* used to refer to *long unsigned*, or *'ll'* to *long long unsigned*. Finally, the last number indicates how much variables that data type allocates (one or two) for each instruction.

```
1 #include "../hmc.hpp"
2
3 int main(int argc, char *argv[]){
4   uint128_t mem_ret;
5   mem_ret =  _hmc128_nor_s
6              (&mem_op1, imm_op2);
7 }
```

Code 1. Intrinsics-HMC function call example.

```
1 __h128ll1 _hmc128_nor_s
2  (__h128ll1 *mem_op, __h128ll1 imm_op){
3     __h128ll1 r = *mem_op;
4     *mem_op = ~(*mem_op | imm_op);
5     return r;
6 }
```

Code 2. Intrinsics-HMC source code for *nor* HMC operation.

Codes 1 and 2 present a *nor* operation between a memory operand and an immediate performed by the function call to *_hmc128_nor_s()*. Notice that, during the function call, the programmer is free to use the usual data types defined by the compiler instead of the ones defined in our library because they are equivalent. However, these new data types are declared to maintain the data type standard of the HMC. Once the program is compiled, the programmer can perform tests and code debug to ensure correctness before the trace generation step.

3.3. Modified SiNUCA-Tracer

With the binary code in hand, the Intel Pin instruments it with the Pin tool called SiNUCA-Tracer. Pin is developed by Intel and integrated with SiNUCA to instrument and analyze code, allowing program developing with routines provided by its own called Pin tools. These Pin tools can be integrated to a program to determine which code parts will be analyzed and inspected and in sequence, what kind of analysis/operation should be done in these parts (memory, execution, cost and performance).

The Pin tools should be applied in binary code when source code does not need more changes. The analysis tools should be developed in C or C++ and analysis occurs at the same time as execution, this is why Pin is known as a just-in-time compiler. In our work, we are using Pin in version 3.2 (revision number 81205).

SiNUCA-Tracer starts opening the program binary image and traverses it. For each routine identified it records into the static trace output file all the instructions present in that routine. The instructions are split into Basic Blocks (BBL). These BBLs contains

all the instructions starting from the instruction after some branch/jump (target instruction) and ends at the first branch/jump found in the execution flow. This way, the static trace may contain the same instruction present in more than one BBL.

During the identification of instructions and BBLs, all the instructions that can perform load or store requests are instrumented in order that, whenever it is executed, the memory address accessed will be written into the memory trace file. Notice that we are dealing with x86 ISA, which may contain a single instruction that perform up to 3 memory accesses (two loads and one store). The head of each BBL identified will receive an identification and will be instrumented in such way that, whenever such BBL is executed, it will write into the dynamic trace file its identifier. This way, a loop over a BBL present in the execution will only store the BBL identification in the dynamic trace to denote each repetition.

After understanding the SiNUCA-Tracer mechanism, we modified it first to identify all the Intrinsics-HMC functions. Each function call must be translated to the correct HMC operation. In this case, the goal is to simulate traces as they have executed by a computer that has an architecture with support to HMC ISA, therefore, the SiNUCA-Tracer suppresses temporarily the generation of x86 traces.

Notice that compiler may have created real dependencies between the registers from outside and inside each Intrinsics-HMC function. After the translation of these functions to HMC instructions, such dependencies must be kept. During the binary instrumentation, we analyze all the input and output dependencies. We call input dependency, all registers that are read inside the function, but the register was written before the function call. The output dependency corresponds to all registers written inside the function that can be read after the function return. Therefore, for each HMC instruction, it contains as read registers a list of all input dependencies, and for the write registers a list of all the output dependencies.

Figure 3 illustrates the analysis performed to keep the right dependencies during each HMC function translation. In this example, we can observe the static trace code for the *add* function (lines 1∼27). The lines 28∼30 contains the translated HMC instruction to perform the *add* operation. This figure brings read (bold with shade), write (bold), base and index (underlined) registers for each instruction. We can see that registers 5 and 6 (line 3) are first read before any previous write, it means that such registers are real dependencies, and thus must be present on the final HMC instruction. The same does not occur for read register 10 (line 10), it was previous written by instruction on line 9, so it does not represent a real dependency to the HMC instruction. This way, when performing the trace generation for HMC, our mechanism analyzes all the input and output registers (inside the circles in the figure) from our function and we use these as read and write registers respectively in the simulable HMC instruction.

In order to replace the functions call by simulable HMC instructions, we added new basic blocks in the static trace, each containing just one HMC instruction according to the function provide by the library. When generating the dynamic trace, we replaced each call to HMC function by the call to one of these basic block containing the respective HMC instruction. Nevertheless, the memory trace will have the specific memory address used by the HMC function. Such address is extracted from routine parameters by Pin tool.

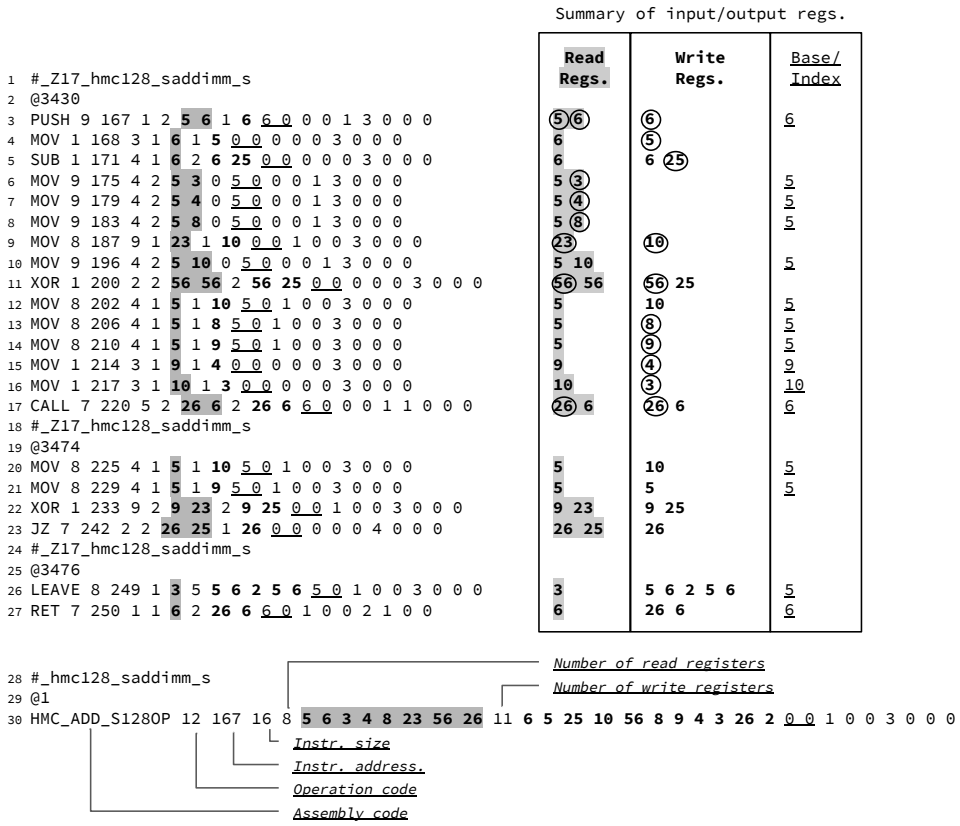


Figure 3. Example illustrating the x86 function translated to HMC instruction, also presenting the read and write register dependencies.

At the end of the execution, the generated traces can be used to feed the SiNUCA in order to simulate the program execution considering an ISA containing HMC instructions.

Currently, only the instructions provided in the HMC specification, are available in the Intrinsics-HMC library. However, such library is easy to extend, thus enabling the creation of new functions that still do not exist. For such extensions, few modifications are required inside the Intrinsics-HMC library and the SiNUCA-Tracer, so that the correct translation of the function is made to a new instruction supported by the simulator.

4. Methodology and Experimental Results

In this section we present the features of the micro benchmark used describing the process to prepare the traces. We also present the results generated by the simulations describing the benefits of using Intrinsics-HMC.

4.1. Benchmark applications

In order to validate our approach, we evaluated the simulation with a micro benchmark composed of a database *join* algorithm and *select scan* from a real query from TPC-H[1]. Those programs were chosen due to their behavior of data streaming which is suitable to exploit in-memory processing coupled with HMC data processing capabilities.

[1]A standard benchmark for decision support in database systems: `http://www.tpc.org/tpch`.

```
#include "../hmc.hpp"

void nljoin(vector<__h64ll> &outer, vector<__h64ll> &inner,
vector<__h64ll> &join_index) {
  for(size_t i=0; i < outer.size(); ++i) {
    for(size_t j=0; j < inner.size(); ++j) {
      if( _hmc64_equalto_s(outer[i], inner[j]) == 1 ) {
        join_index[i] = j;
        break;
      }
    }
  }
}
```

Code 3. Nested Loop Join using Intrinsics-HMC.

The Join algorithms are the kernel of the join operator in Database Management System (DBMS) which combines two relations (tables) by comparing the join attributes and generating a set of tuples (records or rows in a table) that matching these attributes. Commonly, the join attributes are primary and foreign keys. One of the forerunner join algorithm is the nested loop join (NLJoin), depicted in code 3. That algorithm traverse two vectors, each one representing a relation, the outer loop interact in the largest relation and the inner in the smallest one. Inside inner loop is performed a comparison between the join attributes, in which the HMC intrinsic function *_hmc64_equalto_s* is used, case these attributes match a join index[2] is performed.

From TPC-H was selected the query 6 because it scans some columns and their values are accessed just once, i.e., the values, after used, are not touched anymore in the query plan. The query 6 is presented in code 4, it performs a select scan by applying predicates in three columns (WHERE clause) and projecting a sum of two columns just for the tuples that passed in the predicates evaluation.

Code 5 is an implementation of query 6, using Intrinsics-HMC. It encompasses a loop to traverse four columns stored in arrays, the columns involved in the WHERE clause are evaluated by the intrinsic functions: *_hmc64_cmpswapgt_s* (compare and swap if greater than) and *_hmc64_cmpswaplt_s* (compare and swap if less than). Just the tuples that passed by the predicates evaluation are added to the resulting variable.

```
SELECT
  sum(l_price * l_disc) as revenue
FROM
  lineitem
WHERE
  l_date >= date '1994-01-01'
  AND l_date < date '1994-01-01' +
      interval '1' year
  AND l_disc between 0.05 AND 0.07
  AND l_quant < 24;
```

Code 4. Query 6 from TPC-H in SQL code.

```
void query6() {
  for(size_t i=0; i < l_date.size(); ++i) {
    _hmc64_cmpswapgt_s(&l_date[i], 19940101);
    _hmc64_cmpswaplt_s(&l_date[i], 19950101);
    _hmc64_cmpswapgt_s(&l_disct[i], 5);
    _hmc64_cmpswaplt_s(&l_disc[i], 7);
    _hmc64_cmpswaplt_s(&l_quant[i], 24);
    if( l_date[i] != 19940101 &&
      l_date[i] != 19950101 &&
      l_disc[i] != 5 &&
      l_disc[i] != 7 &&
      l_quant[i] != 24)
    {
      res += l_price[i] * l_disc[i];
    }
  }
}
```

Code 5. Query 6 using Intrinsics-HMC in C code.

[2]More explanations about join index can be found in `http://cs-www.cs.yale.edu/homes/dna/papers/vldb.pdf`.

4.2. Generated Traces and Simulation Results

The results of the simulations with the generated trace from Intrinsics-HMC are illustrated in figure 4. The simulations illustrates the execution of the *select scan* and *join* operations from typical DBMS.

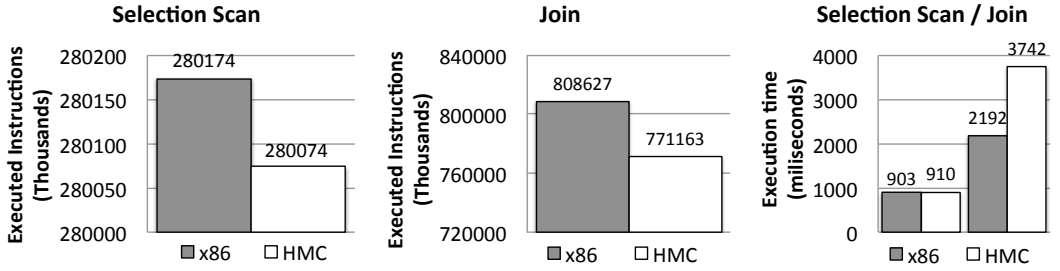


Figure 4. Simulation results for two database operations: select scan and join.

Figure 4 presents the total number of processed instructions and the execution time for each experiment, representing both selection and join operations with HMC instructions compared with x86 instructions only. When the operations are executed interleaving x86 and HMC instructions the total number of instructions reduces since some x86 instructions (from HMC-intrinsics) are replaced by HMC instructions. Comparing the results from the number of executed instructions, it is clear the complete translation of the code through our proposal, but the result also demonstrates an increase in the execution time when using HMC instructions such as the results presented in previous work [Alves et al. 2016], [Hadidi et al. 2017].

5. Related Work

The arrival of the HMC motivated several work over the past decade[Alves et al. 2016], [Hadidi et al. 2017], [Oliveira et al. 2017]. Together with HMC, the High Bandwidth memory (HBM) emerged with a 2.5D stacked memory architecture. In the case of HBM, the vendor can supply a logical die with a memory controller and a set of specific instructions. Thus, we could easily apply our technique to HBM. However, in this paper we choose HMC because it uses a simple and well documented ISA. Moreover, HMC offers a higher bandwidth of 320 GB/s compared to HBM, which achieves only 128 GB/s in the first release, and 256 GB/s on the second version [Kim and Kim 2014].

Most of Processor-in-Memory (PIM) researchers proposed to improve the architecture efficiency, using simulators to perform its analysis. On [Alves et al. 2016], the HMC receives a mechanism called HMC Instruction Vector Extensions (HIVE) to execute vectorized instructions in the logic layer of the HMC. However, the researchers had to generate simulation traces manually to evaluate the mechanism (HIVE). Due to lack of emulators and compilers capable of generating vector codes running in memory, this work refrained from using only simple workloads. On a different direction, the work proposed in [Hadidi et al. 2017] uses a FPGA that generates and sends customized requests to a coupled physical HMC. Although they make use of a physical hardware, any evaluation of new architectural components or different ISA should demand modifications to the compiler also requiring hardware prototypes, which is a costly task.

The work proposed in [Oliveira et al. 2017] presented the Precise Cycle Parallel PIM Simulator (CLAPPS) that allows the modeling of custom PIM architectures for simulation. Compared to gem5 simulator with SMC Simulation Environment (SMC-Sim) [Azarkhish et al. 2016], CLAPPS was developed to provide a more precise model for PIM architectures. In this way, the authors have created an architecture similar to the HMC, based on the specification 2.0 [HMC Consortium 2017]. However, on CLAPPS there is still a necessity of an efficient way to generate input workloads that uses the HMC instructions. Moreover, CLAPPS only simulates HMC memory and it depends on integration to some processor simulator in order to obtain realistic results. The CasHMC [Jeon and Chung 2017] is a cycle-accurate simulator for HMC. It was developed in C++ and covers most specific architecture details of HMC specification. Different from SiNUCA, CasHMC models the HMC as a simple memory, without any PIM capability. Our proposal in this paper improves the SiNUCA features by allowing the automatic generation of traces with HMC instructions from binary code, while also allow some customization in the operation size. Furthermore, our methodology is adaptable to other simulators, and it can be extended to support a different ISA.

6. Conclusions and Future Work

In this paper, we present a methodology for aiding the simulation of emergent HMC architectures and new instructions. Our Intrinsics-HMC library allows writing full codes in high-level languages by utilizing functions that emulate new instructions for in-memory HMC processing. This saves time and reduces errors when writing assembly / simulated code language, which is a major benefit for PIM architecture designers and researchers.

Our proposal also allows the generation of simulated traits through the SiNUCA-Tracer that translates the HMC behavioral functions to HMC simulated instructions. We focused our proposal on the SiNUCA simulator as use case, but other simulators can also benefit from the presented outcomes. Besides, the trace generator SiNUCA-Tracer can also be extended allowing to simulate new PIM architectures. Our Intrinsics-HMC library is freely available and can be accessed in the repository `https://github.com/AlineS/intrinsics-hmc`.

As future work, we consider to modify the SPEC-CPU 2006 [Henning 2006] benchmark suite in order to use the Intrinsics-HMC library. Moreover, we also consider to re-validate previous work that performed the evaluation of new PIM architectures.

References

Alves, M. A. Z., Diener, M., Moreira, F. B., et al. (2015). SiNUCA: a validated micro-architecture simulator. In *High Performance Computation Conf.*

Alves, M. A. Z., Diener, M., Santos, P. C., and Carro, L. (2016). Large vector extensions inside the HMC. In *Conf. on Design, Automation & Test in Europe*.

Azarkhish, E., Rossi, D., Loi, I., and Benini, L. (2016). A case for near memory computation inside the smart memory cube. In *Workshop on Emerging Memory Solutions*.

Balasubramonian, R., Chang, J., Manning, T., et al. (2014). Near-data processing: insights from a MICRO-46 workshop. *IEEE Micro*, 34(4).

Elliott, D. G., Stumm, M., Snelgrove, W. M., et al. (1999). Computational RAM: Implementing Processors in Memory. *Design and Test of Computers*, 16(1).

Hadidi, R., Asgari, B., Mudassar, B. A., Mukhopadhyay, S., Yalamanchili, S., and Kim, H. (2017). Demystifying the characteristics of 3d-stacked memories: a case study for hybrid memory cube. *arXiv preprint arXiv:1706.02725*.

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4).

HMC Consortium (2017). *HMC specification 2.1*.

Jain, R. (1990). *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons.

Jeddeloh, J. and Keeth, B. (2012). Hybrid memory cube new DRAM architecture increases density and performance. In *Symp. on VLSI Technology*.

Jeon, D.-I. and Chung, K.-S. (2017). Cashmc: A cycle-accurate simulator for hybrid memory cube. *IEEE Computer Architecture Letters*, 16(1).

Khalifa, K., Fawzy, H., El-Ashry, S., and Salah, K. (2013). Memory controller architectures: A comparative study. In *Int. Design and Test Symp.*

Kim, J. and Kim, Y. (2014). Hbm: Memory solution for bandwidth-hungry processors. In *Hot Chips Symposium*.

Oliveira, G. F., Santos, P. C., Alves, M. A. Z., and Carro, L. (2017). A generic processing in memory cycle accurate simulator under hybrid memory cube architecture. In *Int. Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation*.

Olmen, J. V., Mercha, A., Katti, G., et al. (2008). 3D stacked IC demonstration using a through silicon via first approach. In *Int. Electron Devices Meeting*.

Patterson, D., Anderson, T., Cardwell, N., et al. (1997). A case for intelligent RAM. *IEEE Micro*, 17(2).

Pawlowski, J. (2011). Hybrid memory cube (hmc). *Hot Chips*, 23.

Pugsley, S., Jestes, J., Balasubramonian, R., et al. (2014). Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads. *IEEE Micro*, 34(4).

Thanh-Hoang, T., Shambayati, A., Deutschbein, C., Hoffmann, H., and Chien, A. (2014). Performance and energy limits of a processor-integrated fft accelerator. In *High Performance Extreme Computing Conf.*

Projeto e Avaliação de uma Arquitetura do Algoritmo de Clusterização *K-means* em VHDL e FPGA

Lucas Andrade Maciel, Matheus Alcântara Souza, Henrique Cota de 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

{lamaciel,matheus.alcantara}@sga.pucminas.br, cota@pucminas.br

Resumo. *O crescimento constante no volume de bases de dados em variadas áreas de pesquisa tem demandado arquiteturas de computadores mais eficazes para a utilização de algoritmos de mineração de dados, de maneira a realizar análises eficientes dessas bases. Mais desempenho é necessário, e arquiteturas mais poderosas tendem a consumir mais energia, acrescentando desafios para os projetos de hardware de processadores. Dessa forma, o projeto de novas arquiteturas com eficiência energética se faz necessário. Este trabalho propõe o projeto e avaliação de uma arquitetura em VHDL e FPGA para o algoritmo de clusterização K-means, visando alto desempenho em arquiteturas heterogêneas. Os resultados mostram que a implementação proposta apresenta uma redução de 91% em relação número de ciclos executados por um processador Intel Xeon E5-2620, consumindo até 95% menos energia.*

1. Introdução

O uso de técnicas para o tratamento de dados produzidos por diversas áreas, tais como processamento de imagens, bioinformática, previsão do tempo e redes sociais, tem produzido resultados com informações complexas, volumosas e heterogêneas. O conceito de *Big Data* surgiu, com o objetivo de tratar esses dados, que possuem estruturas variadas, além de serem obtidos a todo momento. O tratamento desses dados deve ser realizado em tempo hábil, para, por exemplo, gerar resultados para a tomada de decisões. Os processos de descoberta de dados em *Big Data* buscam encontrar padrões e similaridades entre os dados, a partir de análises computacionais, de modo a organizá-los de uma maneira lógica de acordo com suas características.

Realizar o processo de tratamento em tempo hábil demanda alto desempenho computacional, motivo pelo qual os algoritmos pertinentes à área são executados em poderosos processadores, como Unidades de Processamento Gráfico, do inglês *Graphics Processing Units* (GPUs) e o Intel Xeon Phi [Lee et al. 2016], além de arquiteturas heterogêneas, e.g., processadores convencionais + *Field Programmable Gate Array* (FPGA) [Neshatpour et al. 2016], bastante utilizados em computação de alto desempenho. Uma outra abordagem é a utilização de *clusters* de computadores, algumas vezes produzidos de maneira não convencional, e.g., um *cluster* de placas *Raspberry Pi*, buscando redução de custos e economia de energia. [Saffran et al. 2017].

O *K-means* é um algoritmo de clusterização muito utilizado para problemas de *Big Data*, devido sua eficiência. Porém, o tratamento de um grande volume de dados com

alta dimensionalidade não é uma tarefa trivial. Os algoritmos utilizados no contexto de *Big Data* geralmente necessitam de alto desempenho para atingirem seus objetivos em tempo hábil. Dessa forma, FPGAs tem sido alvo da academia e indústria, como alternativa para atender à demanda por desempenho e por soluções em tempo real [Dollas 2014]. Desenvolvedores e arquitetos esbarram em algumas limitações de *hardware*, como tipo, quantidade de bits e número de dimensões dos dados suportados, quando realizam a implementação do algoritmo *K-means* e outros relacionados a mineração de dados em dispositivos não convencionais.

Desta forma, o objetivo deste trabalho é o projeto da arquitetura do algoritmo *K-means* em um *hardware* programável, usando a linguagem de descrição *VHSIC Hardware Description Language* (VHDL). As contribuições dessa pesquisa compreendem uma arquitetura com: (i) suporte a dados de entrada de 64 bits; (ii) suporte a ponto fixo ou ponto flutuante; e (iii) flexibilidade na alteração dos principais parâmetros do algoritmo (número de pontos, centroides, dimensões e iterações) em tempo de operação do *hardware*, sem a necessidade de uma nova compilação para uma nova carga de execuções. Além disso, é apresentado um comparativo do *hardware* implementado com o algoritmo em *software*, em termos de eficiência energética, e uma análise de escalabilidade da solução.

Este artigo está organizado da seguinte maneira. A Seção 2 apresenta uma revisão da literatura, apresentando também alguns trabalhos correlatos. A Seção 3 demonstra a arquitetura proposta e o funcionamento do *hardware* desenvolvido. Na Seção 4 a metodologia adotada no desenvolvimento do projeto é apresentada. A Seção 5 contém os resultados encontrados e, na Seção 6 são apresentadas as considerações finais do trabalho.

2. *Background*

Na busca por alto desempenho para aplicações de *Big Data* há o emprego de arquiteturas não convencionais. Como exemplo, um *cluster* de *Raspberry Pi* para algoritmos de mineração de dados pode ser usado como alternativa para alto desempenho com eficiência energética, quando comparada com a utilização de arqutieturas mais robustas como o Intel Xeon Phi [Saffran et al. 2017].

Da mesma maneira, FPGAs podem ser utilizados para solucionar os mesmos problemas, dada a sua flexibilidade e desempenho [Dollas 2014]. Um FPGA é um circuito integrado que permite ao desenvolvedor reconfigurar o *hardware* após sua fabricação, modificando o funcionamento do circuito. Para reconfigurar, ou reprogramar o FPGA, utilizam-se linguagens de descrição de *hardware* (HDL). O código HDL é compilado, para que a o *hardware* descrito seja implementado em um FPGA.

O uso de linguagens de descrição em *hardware* reconfigurável permite que algoritmos passíveis de implementação em linguagens de programação também possam ser implementados em *hardware*. Embora existam alguns desafios, essa característica é uma vantagem, dado que o processamento em *hardware* tende a ser mais rápido do que em *software*. Partindo desse princípio, o algoritmo *K-means* também pode ser desenvolvido para um FPGA, que é a proposta do presente artigo.

O *K-means* é um algoritmo usado em mineração de dados para particionamento de dados em grupos, ou *clusters*, de acordo com a similaridade entre eles. O algoritmo

trabalha com n pontos com d dimensões espaciais, que são agrupados em k *clusters*, levando-se em consideração a menor distância entre os pontos e os centros de cada *cluster*, chamados de centroides [Lloyd 1982].

Dado o conjunto de n pontos (objetos) e as d dimensões (atributos), define-se a quantidade k de *clusters* que serão gerados. Em seguida, é feita a inicialização dos centroides, a partir de pontos estratégicos (normalmente aleatórios). Na etapa seguinte é calculada a distância euclidiana entre cada ponto com cada centroide, vinculando o ponto ao centroide mais próximo. Por fim, os valores de cada centroide são atualizados, calculando-se a média de todos os pontos mapeados para o centroide verificado. Os dois últimos passos se repetem até que os valores dos centroides se mantenham constantes ou até que se atinja um número pré-determinado de iterações. Após a finalização das iterações, o mapeamento final dos dados em cada *cluster* é produzido. A Figura 1 exibe um exemplo obtido com a execução do algoritmo com $k = 4$.

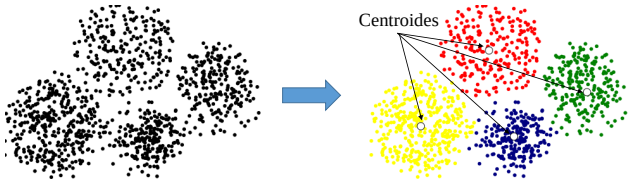


Figura 1. Resultado do algoritmo *K-means* com $k = 4$

A seção seguinte apresenta alguns trabalhos correlatos à este, que realizaram avaliações do algoritmo *K-means* em dispositivos que visam alto desempenho.

2.1. Trabalhos correlatos

No trabalho desenvolvido em [Baydoun et al. 2016] são apresentadas implementações do *K-means* com a biblioteca *OpenMP* utilizando CPU, e também com a biblioteca *CUDA* utilizando GPU. A avaliação dos autores utilizou entradas e parâmetros distintos, demonstrando bons resultados ao sugerir otimizações nas implementações do algoritmo, com a implementação em CUDA sendo a mais eficiente ao se aumentar o número de *clusters*.

De forma similar, em [Lee et al. 2016] o algoritmo *K-means* em paralelo foi avaliado no processador Intel Xeon Phi. Os autores exploraram as técnicas específicas para este processador para obter mais desempenho. As técnicas baseiam-se na organização dos conjuntos de dados em memória de maneira estratégica, para otimizar o processamento paralelo em nível de dados e *threads*.

Uma arquitetura do algoritmo *K-means* para FPGA é proposta em [Kutty et al. 2013]. O objetivo principal é a operação em alta velocidade com frequências de até 400 MHz. Para atingir o objetivo, a arquitetura armazena os dados de entrada em uma *Block RAM* inserida no FPGA; utiliza a distância de Manhattan para o cálculo das distâncias entre os pontos e os centroides; e os dados são de 8 bits. Contudo, a implementação limita o número de *clusters* do algoritmo entre 7 e 9.

Os autores em [Lin et al. 2012] também propõem a implementação em FPGA, visando dados com alta dimensionalidade. São aplicados conceitos de desigualdade triangular, além da eliminação do cálculo de raiz quadrada da distância euclidiana. A partir de dados com 8 bits, 6 *Lookup Tables* (LUTs) e uma memória DDR3 de 512MB externa,

foi possível processar dados de até 1024 dimensões, com desempenho até 17,5% melhor que o *benchmark* MNIST.

Os trabalhos citados que utilizaram FPGA, embora apresentem boas estratégias para a implementação do algoritmo *K-means*, deixam brechas para pesquisas, que são endereçadas pelo presente trabalho. Como exemplo, não foram encontrados experimentos que utilizem o *K-means* para FPGA com dados de 64 bits, sendo de ponto flutuante ou ponto fixo, e flexíveis em termos de alteração dos principais parâmetros (o número de *clusters*, pontos, dimensões e iterações) em tempo de operação. Além disso, nenhum dos trabalhos correlatos encontrados realizou uma análise da eficiência energética da solução.

3. Projeto da arquitetura do algoritmo *K-means*

A arquitetura do *hardware K-means* foi projetada e descrita em VHDL, com foco em FPGA. Uma visão geral da arquitetura é mostrada na Figura 2.

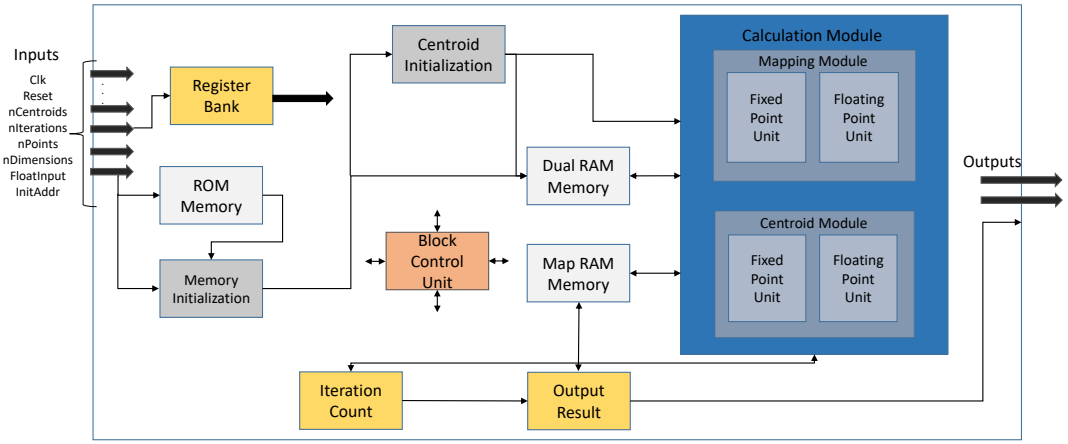


Figura 2. Arquitetura do *hardware K-means*

A arquitetura é baseada em blocos interligados e com funções específicas, gerenciados por uma unidade de controle central. Os blocos de operação desta arquitetura seguem um fluxo de execução determinado pelo diagrama de estados da Figura 3, sendo que a cada pulso de *clock*, o controlador determina qual será o próximo bloco ativo.

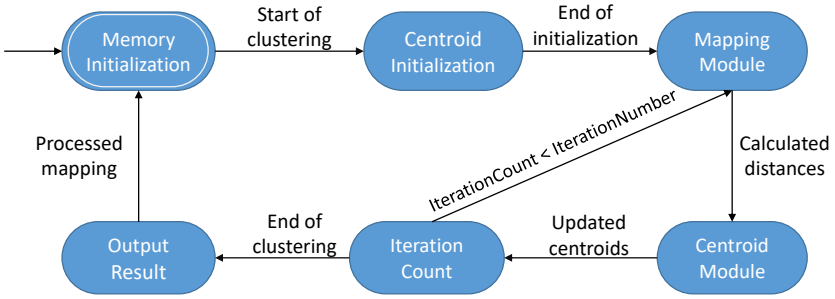


Figura 3. Diagrama de estados do *hardware*

A arquitetura depende de algumas entradas, como *clock* e *reset*; número de centroides, pontos, iterações e dimensões; se os dados serão de ponto fixo ou flutuante (Flo-

atInput); e o endereço na memória ROM do primeiro ponto (InitAddr). As suas saídas (Saida e EnSaida) representam o mapeamento dos pontos em cada grupo.

Para este protótipo, a base de dados é fornecida pelo usuário e armazenada na memória ROM, em tempo de compilação. Os dados dessa base devem ser de 64 bits, sendo que cada 32 bits representa o valor de uma dimensão de cada ponto. Ou seja, caso um ponto tenha 2 dimensões, ele será representado por 1 dado, caso tenha 4 dimensões por 2 dados. Para pontos do tipo ponto fixo, cada dimensão tem o seu valor correspondente em memória. Já para pontos do tipo ponto flutuante, cada dimensão deve ser codificada no padrão IEEE 754 (precisão simples) [Hennessy and Patterson 2014].

3.1. Módulo de Controle

Para controlar o fluxo de operação dos componentes garantindo a correta execução do algoritmo, foi elaborado um bloco de controle, com várias sub-unidades. Cada componente da arquitetura possui uma entrada de habilitação gerenciada pela unidade de controle do *hardware*. As unidades do bloco de controle são descritas a seguir:

Block Control Unit: Unidade de controle global de operação de todos os blocos da arquitetura, que gerencia, habilita e define, a cada pulso de *clock*, qual bloco irá executar sua operação. Recebe como entrada o valor correspondente ao próximo bloco a ser habilitado, e informa a todos os componentes quais as ações que devem realizar.

Iteration Count: Responsável por verificar a quantidade de iterações já realizadas no algoritmo, determinando a sua condição de parada. Possui um contador que compara seu valor com a quantidade definida na entrada, e é incrementado a cada iteração, até o final do processamento, quando o resultado é enviado para a saída.

Output Result: Este componente é acionado após a execução de todas as iterações do *K-means*. Possui um contador que é incrementado a cada pulso de *clock* até atingir o número total de pontos, lendo os dados da *Map RAM Memory*, e os enviando para a saída. Durante sua execução, o pino EnSaida é habilitado informando que o mapeamento final de cada um dos pontos está sendo exibido.

Register Bank: Responsável por receber os dados de entrada (número de centroides, iterações, pontos e dimensões) e armazená-los em registradores internos, que estão disponíveis para as operações que necessitarem. Também atribui a um registrador o endereço de memória inicial para o armazenamento e leitura dos centroides na *Dual RAM Memory*, demarcando a divisão entre pontos e centroides armazenados de forma contígua.

3.2. Módulo de Memórias

Memórias embutidas no *design* da arquitetura, que permitem o acesso mais rápido aos dados, em comparação com a utilização de memórias externas. Assim, utilizando a ferramenta *IP Catalog* do *software* Quartus da Intel/Altera, projetou-se uma memória ROM e duas memórias RAMs.

ROM Memory: Memória ROM que armazenará a base de dados a ser avaliada pelo algoritmo, permitindo o armazenamento de até 16384 dados de 64 bits. Esta memória é utilizada somente para facilitar as avaliações do *hardware* no FPGA de prototipação, pois o objetivo é utilizar memórias externas para armazenar a base de dados.

Dual RAM Memory: Memória RAM com capacidade de até 16384 dados de 64 bits, utilizada para armazenar os pontos da base de dados que serão processados e os valores dos centroides que serão calculados. Esta memória possui dois canais de entrada e dois de saída (A e B), que permitem a leitura e escrita de duas informações em um mesmo ciclo de *clock*, desde que não sejam escritas em um mesmo endereço. Além disso, é organizada de modo a armazenar dinamicamente os pontos, seguidos dos centroides.

Map RAM Memory: Memória RAM utilizada para armazenar o mapeamento final dos pontos de entrada. Possui capacidade de até 16384 dados de 10 bits, podendo receber valores entre 0 e 1023, correspondentes aos centroides mapeados.

3.3. Módulo de Inicialização

No algoritmo *K-means*, é necessário definir inicialmente os centroides de cada *cluster*. Pensando na necessidade de alto desempenho do *hardware*, foi elaborado um bloco de inicialização que realiza a preparação para a execução das etapas do algoritmo. Este bloco é subdivido da seguinte forma:

Memory Initialization: Etapa inicial, na qual, por meio de um contador, o endereço de um ponto a ser lido é enviado para a memória ROM, a partir de um endereço inicial (InitAddr). Em seguida, a memória ROM lê o dado solicitado e o envia para a *Dual RAM Memory*, que armazena o valor recebido em seu primeiro endereço. A cada pulso de *clock*, o contador é incrementado para buscar um novo dado na ROM, que será salvo no próximo endereço da RAM. Este processo ocorre até que todos os pontos sejam armazenados na *Dual RAM Memory*.

Centroid Initialization: No algoritmo *K-means* convencional, a escolha dos pontos que serão centroides, inicialmente, é feita aleatoriamente. Porém, a estratégia utilizada neste projeto consiste em atribuir para os centroides os valores dos k pontos iniciais da base dados, reduzindo a complexidade e latência do *hardware*, sem prejuízos no resultado. Este componente possui um funcionamento baseado em estados, com um contador de centroides, um controlador e um bloco de inicialização. O controlador envia o endereço de um ponto para a *Dual RAM Memory*, que, em seguida, envia o valor lido para o bloco de inicialização. O bloco define o endereço da *Dual RAM Memory* em que o centroide será armazenado. O contador é incrementado e o centroide é salvo na memória. O processo é repetido até que todos os k centroides sejam iniciados.

3.4. Módulo de Cálculo do Mapeamento

A Figura 4 mostra a etapa principal do algoritmo *K-means*, que engloba o cálculo da distância entre pontos e centroides. Este bloco busca encontrar, para cada ponto, o centroide mais próximo dele, atualizando este mapeamento na *Map RAM Memory*. Este processo é repetido até que todos os pontos tenham sido mapeados.

O *hardware* projetado processa em paralelo duas dimensões de cada ponto de entrada, acelerando o cálculo das distâncias. O módulo de cálculo do mapeamento possui duas unidades de processamento com as mesmas funcionalidades e lógica de execução, sendo que uma unidade é responsável por operações com dados do tipo ponto fixo e outra que trabalha com dados do tipo ponto flutuante.

A distância euclidiana foi escolhida, pois produz uma melhor acurácia no resultado para o algoritmo *K-means* [Estlick et al. 2001], porém, uma adaptação foi realizada.

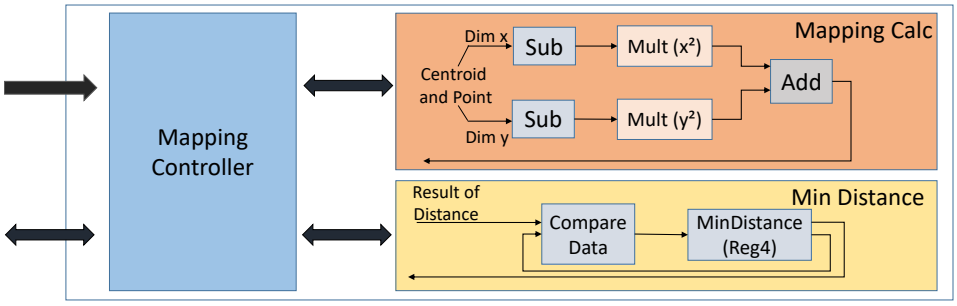


Figura 4. Módulo de Cálculo do Mapeamento

Essa distância é dada pela fórmula $d(p,c) = \sqrt{\sum_{k=1}^{n} |p_k - c_k|^2}$, em que n representa o número de dimensões e p_k e c_k representam o k-ésimo atributo do ponto p e do centroide c que são comparados. Contudo, a implementação do cálculo de raiz quadrada adiciona complexidade extra ao *hardware*, tal como identificado em [Lin et al. 2012].

Como é necessário encontrar somente a menor distância entre um ponto e um centroide, foi removido o cálculo da raiz quadrada, produzindo uma solução aproximada que não prejudica o mapeamento final. Foi usado então o valor da distância sem a raiz quadrada, obtido pela equação adaptada: $d(p,c)^2 = \sum_{k=1}^{n} |p_k - c_k|^2$.

Cada bloco de cálculo de distância tem dois multiplicadores e um somador que verificam duas dimensões em paralelo, armazenando a soma dos quadrados em um acumulador (*distanceCalc*). Enquanto houver dimensões para serem verificadas, o *hardware* calcula a distância destas próximas dimensões e soma seus resultados com os valores armazenados no acumulador. Ao final do cálculo das distâncias, o valor armazenado no *distanceCalc* é comparado com a distância mínima quadrática encontrada até o momento, que está armazenada em um registrador (Reg4). Se a nova distância for menor do que o valor de Reg4, este é atualizado e o centroide correspondente é mapeado para o ponto, de modo que seu valor é armazenado em outro registrador (Reg7). Este processo é repetido até que todos os centroides tenham sido comparados com o ponto selecionado. Ao terminar estas operações, o valor armazenado em Reg7 é mapeado para o ponto e gravado no endereço correspondente na *Map RAM Memory*. Paralelamente, o próximo ponto é carregado no bloco de cálculo de distâncias e um novo ciclo de verificações é executado, repetindo até que todos os pontos tenham sido mapeados nos centroides.

3.5. Módulo de Cálculo dos Centroides

O módulo de cálculo de centroides, exibido na Figura 5, possui três vetores para armazenar a soma dos atributos. O processamento ocorre de acordo com uma máquina de estados que coordena as unidades de soma e de cálculo de média. O módulo entra em operação após o término do cálculo das distâncias, sendo responsável por processar e atualizar os valores dos centroides, armazenados na região inferior da *Dual RAM Memory*.

Este módulo é constituído de dois componentes, um para dados de ponto fixo e outro para dados de ponto-flutuante, da mesma forma que no módulo de cálculo do mapeamento. A seguir, descreve-se a sua máquina de estados, em sua ordem de execução.

IDLE: Estado de espera do módulo, até receber um sinal para iniciar.

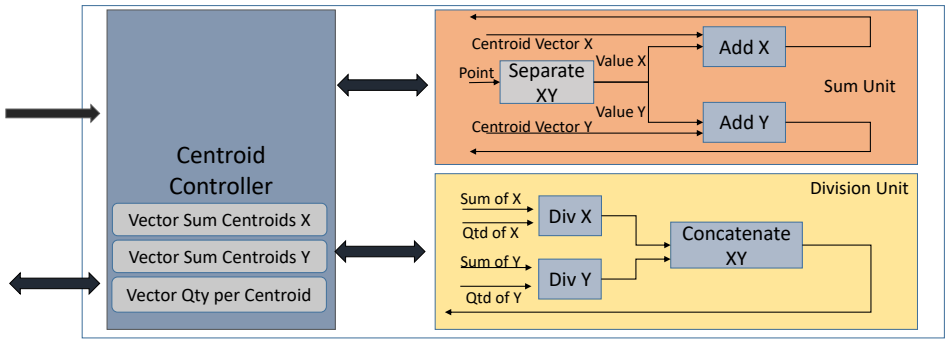


Figura 5. Módulo de Cálculo dos Centroides

LOADDATA: Neste, é verificado se todos os dados já foram lidos. Se sim, o estado VERIFYDIV é acionado. Se não, é solicitada a leitura das dimensões dos pontos na *Dual RAM Memory* e do mapeamento na *Map RAM Memory*.

WAITDATA: Aguarda até que a leitura dos dados solicitados seja realizada.

SUM: Aciona a unidade de soma, em que os valores de duas dimensões são somados e armazenados em vetores. Cada posição do vetor corresponde a um centroide. O processo é executado até todos os pontos serem verificados, em seguida vai para o estado LOADDATA.

VERIFYDIV: Neste estado, é verificado se todos os centroides já foram validados. Se sim, aciona-se o estado VERIFYDIMENSION. Se não, o estado DIV.

DIV: Responsável por controlar a unidade que calcula a média dos pontos mapeados para cada centroide. Esta unidade possui dois componentes de divisão, projetados na ferramenta *IP Catalog*, que calculam em paralelo, a média de atributos de duas dimensões, que ao final são concatenados em um dado de 64 bits. Caso os valores sejam de ponto flutuante, as médias são calculadas mantendo-se os valores decimais. Se forem do tipo ponto fixo, os valores são arredondados, considerando apenas a parte inteira do resultado da divisão.

STOREDATA: Neste estado, os valores dos centroides são atualizados com os novos resultados, em seus endereços correspondentes na *Dual RAM Memory* e em seguida retorna para o estado VERIFYDIV.

VERIFYDIMENSION: Estado em que é verificado se todas as dimensões dos pontos foram calculadas. Se sim, aciona o estado ENDSTATE, caso contrário, as próximas dimensões são lidas, retornando ao estado LOADDATA.

ENDSTATE: Encerramento do módulo, após atualização de todos os centroides.

4. Metodologia de avaliação

A partir da apresentação da arquitetura proposta do algoritmo *K-means*, a descrição do *hardware* respectivo foi realizada em VHDL. A sintetização e compilação foram feitas no *software* Quartus Prime Lite 16.1. O *hardware* foi implementado no FPGA *Intel/Altera Cyclone IV-E EP4CE115F29C7*, contido na placa DE2-115. A estratégia para avaliação do *hardware* compreendeu 4 etapas.

Na 1ª etapa foram geradas bases de dados sintéticas, com valores pseudoaleatórios de uma distribuição normal para cada dimensão, entre 0 e 65536.

A 2ª etapa compreendeu a comparação do número de ciclos de *clock* gastos na execução do *K-means* com dados de ponto fixo para FPGA e com dados de ponto flutuante para FPGA e para o *software* do *CAP Bench* [Souza et al. 2017] executado com 12 *threads*, com diferentes tamanhos da base de dados de entrada. A quantidade de centroides (k) variou entre 2, 4 e 8. Já a quantidade de pontos, variou entre 256, 1024 e 4096. Nesta etapa, o número de dimensões de cada ponto (seus atributos) foi mantido em 4, da mesma forma que o número de iterações do algoritmo (também 4). Utilizou-se um computador com 2 processadores Intel Xeon E5-2620 de 2.10GHz, com 6 núcleos cada, 32 GB de memória RAM, Linux CentOS, e GCC versão 4.9.2.

Na 3ª etapa, fixou-se o número de pontos, iterações e centroides, variando-se a quantidade de dimensões, comparando os ciclos gastos pelo FPGA e pelo *software*.

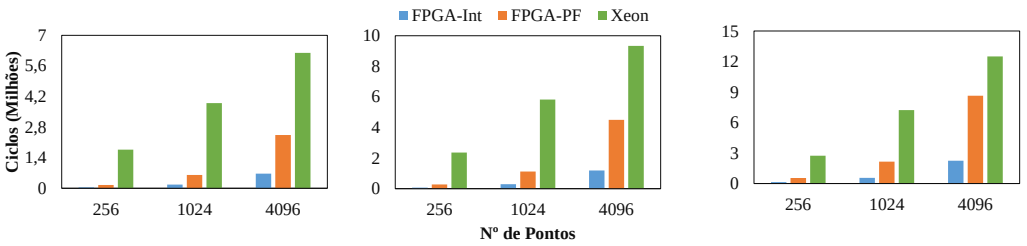
A comparação do tempo de execução e do consumo de energia das duas implementações foi feita na 4ª etapa, com as mesmas bases de dados da 2ª etapa. Também fixou-se a base de dados, porém variando a frequência de processamento do FPGA. Foram utilizadas as ferramentas *Intel PowerPlay EPE* e PAPI versão 5.5.0 para verificar o consumo.

5. Resultados

Nesta seção, são discutidos os resultados obtidos no trabalho. A síntese do *hardware* apresenta os valores mostrados na Tabela 1, referentes a total de multiplicadores, registradores, elementos lógicos e bits de memória disponíveis e utilizados pelo *hardware* no FPGA usado.

Tabela 1. Elementos do *Hardware*

Elementos	Qtde. do FPGA	Qtde. Arq. *K-means*	Uso FPGA (Arq. *K-means*)
Multiplicadores	532	62	12%
Registradores	114480	6866	6%
Lógicos	114480	15000	13%
Bits Mem.	3981312	2270602	57%



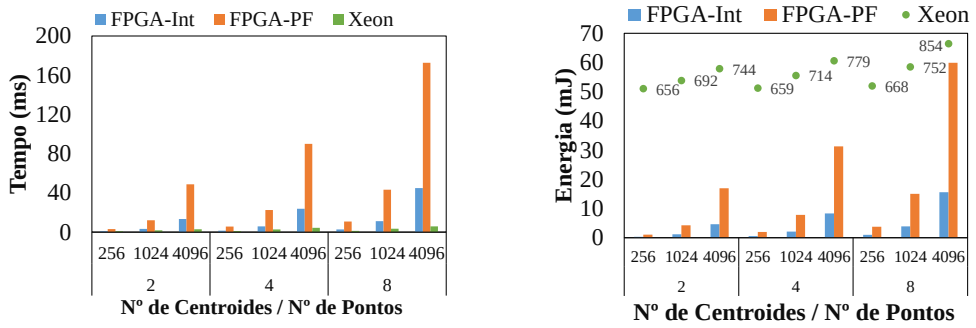
(a) 2 centroides (b) 4 centroides (c) 8 centroides

Figura 6. Quantidade de ciclos por Número de Pontos

O desempenho do *hardware* comparado com a implementação em *software*, em número de ciclos de *clock*, é apresentado na Figura 6, que mostra comportamentos semelhantes ao variar a carga de dados em cada um dos gráficos apresentados. O objetivo

é verificar o desempenho ao utilizar dados de ponto flutuante (FPGA-PF e Xeon) e de ponto fixo (FPGA-Int). Assim, o FPGA consegue um desempenho melhor em relação ao Xeon ao ter uma redução, no pior caso, de 82% e 31%, e no melhor caso de 98% e 91% no número de ciclos, respectivamente com dados de ponto fixo e flutuante. Após a clusterização é obtido um resultado final com 98% de semelhança na distribuição dos dados ao comparar o FPGA-PF com o Xeon, e 70% de semelhança ao comparar FPGA-Int com FPGA-PF, mostrando que mesmo ao se utilizar truncamento são obtidos resultados satisfatórios que justificam a utilização de dados de ponto fixo, já que os cálculos de ponto flutuante são mais complexos. Constata-se também que o *hardware* suporta o aumento da carga de trabalho adequadamente.

De modo a verificar o desempenho da arquitetura ao se variar o número de dimensões dos dados de entrada em 2, 4, 6 ou 8, foram realizadas execuções no FPGA e no Xeon, fixando o número de pontos em 1024, o de centroides e o de iterações em 4. Com dados de ponto fixo gasta-se 6% e com dados de ponto flutuante 20% do número de ciclos executados pelo processador, mostrando que a quantidade de dimensões não interfere na melhoria adquirida com o uso do FPGA.



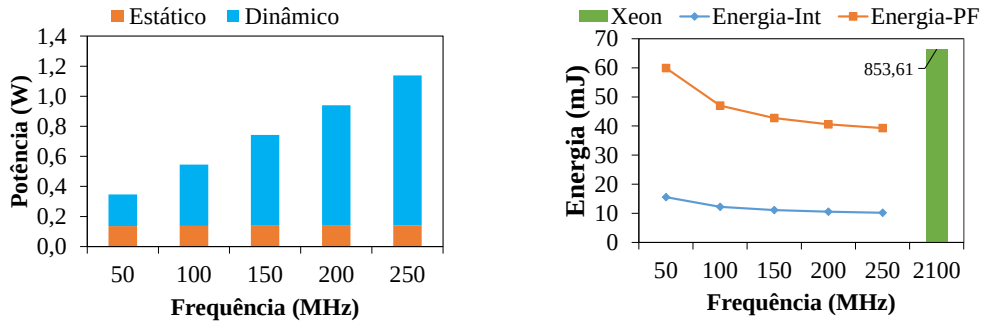
(a) Tempo de execução

(b) Energia consumida

Figura 7. Tempo e energia com variação da carga de dados

Foram comparados os tempos de execução do *K-means* em FPGA (50MHz) e no processador Xeon (2100MHz), com o intuito de mostrar, que embora a arquitetura proposta execute o algoritmo em uma quantidade de ciclos menor, a frequência do FPGA utilizado pode interferir consideravelmente no tempo de resposta da aplicação, como pode ser verificado na Figura 7(a), onde o tempo de execução do Xeon é até 80% menor que o do FPGA-Int e até 90% menor que o do FPGA-PF. Além dessa avaliação, foi medido o consumo de energia, em milijoules, em dois tipos de análises. O primeiro tipo baseou-se em se variar a carga de dados de entrada, utilizando o FPGA com frequência de 50MHz, mostrando que o FPGA consome em seu pior caso, 2% com dados de ponto fixo e 8% com ponto flutuante, e em seu melhor caso 1% em ambos os tipos de dados, da energia consumida pelo Xeon, mostrado na Figura 7(b).

O outro tipo mostra uma projeção do consumo de energia do FPGA ao variar sua frequência de operação mantendo uma carga fixa de dados de ponto flutuante com 4096 pontos, 8 centroides, 4 dimensões e 4 iterações. O FPGA possui um gasto de potência estático, *hardware* ligado sem executar operações, e outro dinâmico, *hardware* ligado processando operações, que cresce à medida que frequência aumenta, mostrado na Fi-



(a) Potência por frequência (FPGA-PF)

(b) Energia por frequência

Figura 8. Consumo por frequência

gura 8(a). O resultado de energia consumida do Xeon e do FPGA com ponto fixo e flutuante, é comparado na Figura 8(b), mostrando que o FPGA tende a consumir menos energia à medida que a frequência aumenta, pois mesmo com o aumento de potência dinâmica, o tempo de processamento reduz, diminuindo consequentemente a energia consumida. Além disso, é possível verificar que o FPGA consome em seu melhor e pior caso, respectivamente 95% e 93% com dados de ponto flutuante, e 98% com dados de ponto fixo, menos energia do que o processador.

6. Conclusão e trabalhos futuros

Neste artigo é apresentado o projeto de uma arquitetura de *hardware* do algoritmo *K-means* e realizada sua implementação em FPGA. O projeto desenvolvido possui uma entrada de dados de 64 bits, com atributos de 32 bits, podendo ser de ponto fixo ou flutuante, que permitem que o usuário altere o número de pontos, *clusters*, iterações e dimensões em tempo de operação do *hardware*. As estratégias do processamento de dois atributos em paralelo e no cálculo de distância euclidiana tornam o *hardware* proposto mais robusto.

A avaliação de desempenho do projeto em comparação ao *benchmark* CAP Bench, executado em uma máquina com processador Xeon, mostra que o FPGA apresenta um ganho em relação ao Xeon, no melhor caso, de 98% e 91% do número de ciclos, respectivamente com dados de ponto fixo e flutuante. Ao avaliar o tempo de execução do algoritmo em um FPGA com 50MHz e no Xeon com 2100MHz, nota-se que a frequência do dispositivo utilizado interfere diretamente no tempo de resposta da aplicação, onde o Xeon apresenta um tempo de até 90% menor, em comparação com o FPGA-PF. Já em relação a avaliação de consumo de energia, o FPGA-PF consome, em seu melhor caso, 95% menos energia do que o processador. A contribuição deste trabalho é a elaboração de uma arquitetura de *hardware* e sua implementação em VHDL para FPGA, do algoritmo *K-means*, com suporte a dados de análise com valores de ponto fixo ou ponto flutuante de 64 bits, não encontrado em trabalhos correlatos, demonstrando sua avaliação de desempenho e energia.

Os resultados apresentados são promissores e para trabalhos futuros o protótipo será avaliado em outros FPGAs, e.g. Arria 10. O projeto visa um funcionamento em frequências mais altas com variação da carga de trabalho, baixo tempo de execução, es-

calabilidade e eficiência em desempenho e energia. Além disso, há também a retirada da memória ROM da arquitetura, a melhoria das etapas de cálculo de mapeamento e cálculo de centroides para execução em paralelo de um número superior a 2 atributos, substituição do número de iterações como ponto de parada da clusterização na arquitetura, pela comparação da distância mínima de modificação dos valores dos centroides e verificação da possibilidade de se utilizar a distância de Manhattan no cálculo de mapeamento.

Agradecimentos

Os autores agradecem ao CNPq, FAPEMIG, CAPES e a PUC Minas pelo suporte no desenvolvimento do trabalho.

Referências

Baydoun, M., Dawi, M., and Ghaziri, H. (2016). Enhanced parallel implementation of the k-means clustering algorithm. In *3rd International Conference on Advances in Computational Tools for Engineering Applications (ACTEA)*, pages 7–11.

Dollas, A. (2014). Big data processing with fpga supercomputers: Opportunities and challenges. In *IEEE Computer Society Annual Symposium on VLSI*, pages 474–479.

Estlick, M. et al. (2001). Algorithmic transformations in the implementation of k-means clustering on reconfigurable hardware. In *9th International Symposium on FPGA*, pages 103–110. ACM.

Hennessy, J. L. and Patterson, D. A. (2014). *Organização e Projeto de Computadores: a interface hardware/software*, volume 4. Elsevier Brasil.

Kutty, J. S. S., Boussaid, F., and Amira, A. (2013). A high speed configurable fpga architecture for k-mean clustering. In *IEEE International Symposium on Circuits and Systems (ISCAS2013)*, pages 1801–1804.

Lee, S. et al. (2016). Evaluation of k-means data clustering algorithm on intel xeon phi. In *IEEE International Conference on Big Data (Big Data)*, pages 2251–2260.

Lin, Z., Lo, C., and Chow, P. (2012). K-means implementation on fpga for high-dimensional data using triangle inequality. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 437–442.

Lloyd, S. (1982). Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137.

Neshatpour, K., Sasan, A., and Homayoun, H. (2016). Big data analytics on heterogeneous accelerator architectures. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–3.

Saffran, J. et al. (2017). A Low-Cost Energy-Efficient Raspberry Pi Cluster for Data Mining Algorithms. In *Desprez F. et al. Euro-Par 2016: Parallel Processing Workshops. Euro-Par 2016*, Lecture Notes in Comp. Science, vol 10104, Springer, Cham.

Souza, M. A. et al. (2017). Cap bench: a benchmark suite for performance and energy evaluation of low-power many-core processors. *Concurrency and Computat.: Pract. Exper.*, 29:e3892. doi: 10.1002/cpe.3892.

Análise Arquitetural Comparativa do Desempenho de Redes-em-Chip baseada em Simulação

Eduardo Alves da Silva, Cesar Albenes Zeferino

Laboratório de Sistemas Embarcados e Distribuídos (LEDS)
Centro de Ciências Tecnológicas da Terra e do Mar (CTTMar)
Universidade do Vale do Itajaí (Univali) – Itajaí, SC – Brasil

{eas,zeferino}@univali.br

Resumo. *As Redes-em-Chip são a infraestrutura de comunicação adotada nos sistemas many-core atuais e diversos estudos têm comparado o desempenho de arquiteturas de rede. Porém, esses trabalhos carecem de análises quantitativas ou de abrangência arquitetural. Para superar essas limitações, este artigo apresenta uma biblioteca de componentes de simulação que possibilita a modelagem de diferentes alternativas arquiteturais e a coleta de dados para uma comparação quantitativa. Um amplo conjunto de experimentos foi realizado para demonstrar a efetividade da biblioteca implementada. Os resultados obtidos permitem identificar as melhores alternativas de rede para diferentes cenários de tráfego.*

1. Introdução

As Redes-em-Chip (NoCs – Networks-on-Chip) se tornaram a arquitetura de comunicação preferencialmente adotada nos sistemas computacionais multiprocessados integrados em uma única pastilha de silício, os SoCs (Systems-on-Chip). Isso ocorreu devido às vantagens apresentadas por essas redes em relação às arquiteturas de comunicação tradicionalmente utilizadas nesses sistemas, como o barramento. Como principais vantagens destacam-se a escalabilidade de desempenho, a reusabilidade e o paralelismo de comunicação [Andriahantenaina et al. 2000].

Desde o surgimento das primeiras NoCs, várias alternativas arquiteturais foram propostas para atender a demanda de desempenho dos novos projetos. Usualmente, essas propostas são avaliadas por meio de uma descrição arquitetural ou por comparativos baseados em experimentos em relação a uma arquitetura de referência. Nessa segunda linha, os experimentos geralmente são realizados com o uso de simuladores [Ogras and Marculescu 2013].

De um lado, o problema é que descrições arquiteturais não dão subsídios para a tomada de decisão, pois carecem de dados quantitativos de desempenho das arquiteturas analisadas. De outro, cada grupo de pesquisadores desenvolve seu próprio simulador e realiza uma exploração arquitetural limitada, comparando a sua proposta com uma ou poucas arquiteturas de referência.

Inserido neste contexto, este trabalho busca auxiliar projetistas de sistemas integrados na tomada de decisão por meio de uma análise quantitativa do desempenho de arquiteturas de NoCs. Para isso, foi desenvolvida uma biblioteca de componentes simuláveis que foi integrada a uma ferramenta de simulação de arquiteturas de

interconexão intrachip. A biblioteca e a ferramenta foram implementadas de forma a flexibilizar a adição de novos componentes, o que possibilita que propostas futuras possam ser facilmente avaliadas frente às arquiteturas existentes e disponíveis no ambiente. Para demonstrar a diversidade dessa biblioteca e a sua flexibilidade, foram realizados experimentos que permitiram obter informações sobre as arquiteturas com melhor desempenho para diferentes cenários de tráfego.

As principais contribuições do trabalho residem na infraestrutura para avaliação unificada do desempenho de NoCs e uma análise comparativa de um conjunto de arquiteturas de comunicação intrachip com base em dados quantitativos.

O restante deste artigo está organizado em quatro seções. A Seção 2 discute os trabalhos relacionados que serviram para evidenciar o problema alvo deste trabalho. A Seção 3 apresenta aspectos do desenvolvimento da biblioteca proposta. Já a Seção 4 descreve os experimentos realizados e discute os resultados obtidos. Por fim, a Seção 5 apresenta as conclusões do trabalho.

2. Trabalhos Relacionados

Para evidenciar a lacuna na avaliação de desempenho de arquiteturas de Redes-em-Chip em que trabalhos descritivos não subsidiam escolhas e trabalhos experimentais não possuem abrangência arquitetural, uma Revisão Sistemática da Literatura foi aplicada. Essa revisão teve o objetivo de identificar os trabalhos mais recentes que realizam análises comparativas de NoCs. A pesquisa bibliográfica foi feita nas principais bibliotecas digitais da área de Ciência da Computação (IEEE Xplore, ACM Digital Library e ScienceDirect) e também com uso do Scopus e do Google Scholar. A expressão de busca foi adaptada para cada uma das bases de forma que correspondesse ao mesmo padrão de pesquisa. A forma genérica da expressão é a seguinte:

```
(noc OR "network on chip") AND
(comparison OR compare OR comparing OR comparative)
```

Somente trabalhos escritos na língua inglesa e que contivessem os termos da expressão acima nos seus títulos foram buscados. Como critério de seleção, foi definido que apenas trabalhos publicados a partir do ano 2011 seriam analisados. Foram excluídos trabalhos cujos títulos e/ou resumos não abordassem a avaliação de desempenho de NoCs baseada em simulação ou que tratassem de tecnologias alternativas (*e.g.* redes ópticas e sem fio). Com a aplicação desses critérios, foram selecionados 17 trabalhos, os quais foram analisados e caracterizados, como ilustram as tabelas a seguir.

A Tabela 1 apresenta uma visão geral que ilustra o método de comparação utilizado, o atributo arquitetural avaliado e as alternativas arquiteturais exploradas em cada trabalho. O principal método de avaliação utilizado é a simulação. Alguns trabalhos realizam comparações qualitativas (método descritivo) que não dão subsídios suficientes às suas análises, enquanto outros realizam o mapeamento em silício (síntese). Os níveis de implementação dos simuladores variam na acurácia e no tempo de execução. Porém, para avaliar novas propostas, são utilizados modelos de alta acurácia (em nível de ciclos). Em alguns casos, são utilizados simuladores já existentes, como nos trabalhos de Yin et. al. (2012), Hao et al. (2011) e Pandey e Gupta (2012), em que foram adotados os simuladores ORION 2.0, Nirgam e NS-2, respectivamente. A

maior parte dos trabalhos analisados tem por objetivo comparar entre duas a três topologias ou algoritmos de roteamento (alguns comparam técnicas de chaveamento), porém, nenhum aborda mais de um atributo arquitetural. Em geral, os trabalhos comparam uma proposta de solução em relação à uma abordagem convencional.

Tabela 1. Visão geral dos trabalhos analisados na revisão sistemática

Trabalhos	Método	Atributo	Alternativas
[Sadawarte, Gaikwad e Patrikar 2011]	Descritivo	Chaveamento	Circuito, mensagem, pacote, Wormhole e Virtual Cut-through
[Hao et al. 2011]	Simulação	Roteamento	XY e Odd-Even
[Ghidini et al. 2012]	Simulação	Topologia	Malha 2D e Malha 3D
[Yin et al. 2012]	Simulação	Topologia	Malha 2D e Honeycomb
[Kalimuthu e Karthikeyan 2012]	Descritivo	Trabalhos da literatura	*n.i.*
[Ju e Yang 2012]	Simulação	Topologia	Malha 2D, Toróide 2D e Malha Hierárquica
[Romanov e Lysenko 2012]	Simulação	Topologia	Malha 2D, Toróide, Butterfly, Árvore gorda e 6 topologias irregulares
[Manna, Chattopadhaya e Sengupta 2012]	Síntese e simulação	Topologia	Mesh of Tree original e modificada
[Parandkar, Dalal e Katiyal 2012]	Simulação	Roteamento	XY, Odd Even e DyAD (Dynamic Deterministic Adaptative)
[Pandey e Gupta 2012]	Simulação	Topologia	Malha 2D e topologias irregulares
[Du et al. 2013]	Simulação	Roteamento	XY, Turn-model e Retrograde-turn-model
[Jetly 2013]	Síntese e simulação	Chaveamento	Wormhole e Store-and-Forward
[Sllame e Abdelkader 2014]	Simulação	Topologia	Malha 2D e Árvore gorda
[Wang et al. 2014]	Síntese	Chaveamento	Wormhole e Virtual Cut-Through
[Radfar, Zabihi e Sarvari 2014]	Simulação	Topologia	Malha 2 e 3D, Toróide 2 e 3D e Hipercubo 2 e 3D
[Jaina, Kumarb e Sharmac 2015]	Síntese	Topologia	Malha 2D, Toróide 2D e Anel
[Harbin e Indrusiak 2016]	Simulação	Níveis de abstração	Em nível de ciclos e transação

n.i.: não informado

Apenas seis dos 17 trabalhos analisados foram identificados como fortemente relacionados ao tema alvo deste estudo. Os demais foram desconsiderados por não utilizarem simulação e pelo pouco detalhamento sobre a arquitetura das NoCs e os cenários de tráfego adotados, o que impossibilitaria a sua reprodução.

A Tabela 2 apresenta a caracterização dos trabalhos quanto às topologias e demais atributos arquiteturais analisados. A Tabela 3, por sua vez, identifica os métodos de injeção de tráfego e as métricas para avaliação de desempenho.

É possível verificar que a maior parte dos trabalhos é focada na avaliação de topologias e que a Malha 2D é a mais utilizada. Isso ocorre porque a topologia é um dos atributos que mais impacta nos limites de desempenho da infraestrutura de comunicação e a Malha 2D serve de referência para novas propostas. Poucos trabalhos avaliaram outros atributos, tais como a memorização e a arbitragem, sendo que nenhum dos que foram analisados abordou técnicas de controle de fluxo.

Um aspecto a ser destacado é que estabelecer cenários de avaliação justos para os comparativos não é uma tarefa trivial. Em alguns trabalhos, é nítido que a experimentação realizada é tendenciosa para "beneficiar" uma arquitetura em relação à outra, gerando vieses de avaliação. Os cenários de comunicação comumente são

implementados por meio de geradores de tráfego sintético para focar na avaliação do desempenho da rede. Porém, em um dos trabalhos, foi utilizado um modelo de sistema real justificando que tráfego sintético não representa o comportamento de aplicações.

Tabela 2. Atributos arquiteturais abordados

Trabalho	Topologia	Chaveamento	Controle de fluxo	Arbitragem	Memorização
Ghidini et al. (2012)	Malha 3D Malha 2D	Wormhole	Baseado em créditos	RR	Nas entradas, dimensionáveis e 1 VC
Yin et al. (2012)	Malha 2D Malha Honeycomb	*n.i.*	*n.i.*	*n.i.*	Nas entradas, dimensionáveis em profundidade e VCs
Ju e Yang (2012)	Malha 2D Toróide 2D Malha Hierárquica	*n.i.*	Handshake	RR	Nas entradas com 64 posições de 8 bits (64 bytes)
Manna, Chattopadhaya e Sengupta (2012)	Mesh-of-Tree	Wormhole	*n.i.*	RR	Nas entradas com 6 flits de profundidade
Jetly (2013)	Malha 2D	Wormhole SAF	*n.i,*	*n.i.*	*n.i*
Sllame e Abdelkader (2014)	Malha 2D Árvore Gorda	Wormhole	*n.i,*	RR	Nas entradas dimensionáveis em profundidade e VCs
Harbin e Indrusiak (2016)	Malha 2D	Wormhole	Baseado em créditos	Preemptivo	Com VC
				Não-preemptivo	Sem VC

Onde: VC: Virtual Channel; SAF: Store-and-Forward; RR: Round-robin; *n.i.*: não informado

Tabela 3. Modelos de tráfego e métricas de avaliação

Trabalho	Modelo de tráfego	Métricas de desempenho
Ghidini et al. (2012)	Sintético; Uniforme e complemento; Injeção constante a 800Mbps	Latência, vazão e utilização de recursos
Yin et al. (2012)	Gerador sintético com taxas de injeção de 1%, 0.5% e 0.1%	Latência
Ju e Yang (2012)	Sintético; Uniforme; Injeção de pacotes sem pontos de contenção	Latência, vazão e utilização recursos
Manna, Chattopadhaya e Sengupta (2012)	Sintético; Uniforme; Tráfego em rajadas	Latência e vazão
Jetly (2013)	Sistema real; requisições mestre-escravo	Vazão e frequência de operação
Sllame e Abdelkader (2014)	Sintético; Uniforme; Pacotes de 200, 300 e 400 bytes	Latência e Vazão
Harbin e Indrusiak (2016) Ghidini et al. (2012)	Aplicação veicular, decodificador H.264 e sintético	Tempo de execução das aplicações

Por fim, observou-se que há muita variação na precisão dos resultados apresentados. Alguns trabalhos apresentam resultados precisos e realizam uma discussão suficientemente detalhada. Outros, entretanto, apresentam resultados superficiais sem informar o quanto uma arquitetura é melhor em relação a outra, ou, ainda, carecem de uma discussão fundamentada para justificar o motivo de uma configuração ser superior à outra.

3. Desenvolvimento

O presente trabalho foi desenvolvido com a ferramenta de avaliação de desempenho de NoCs denominada RedScarf [Silva 2014]. Essa ferramenta é composta de uma interface gráfica (*front-end*), desenvolvida com o *framework* Qt [The Qt Company 2017], e um simulador (*back-end*) implementado utilizando SystemC [Accellera 2017] com acurácia em nível de ciclos. O simulador é baseado em uma biblioteca de componentes para a construção de roteadores e NoCs. O *front-end* é responsável pela interface com o usuário e oferece diversos recursos para configuração dos modelos, execução dos experimentos e análise dos resultados. Já o *back-end* realiza as simulações e gera os relatórios sobre os pacotes transferidos pela rede. Os dados desses relatórios são apresentados no *front-end* por meio de gráficos e tabelas, o que facilita a análise das métricas. A geração de tráfego é baseada nos métodos propostos por Tedesco (2005) e os modelos de latência e vazão baseiam-se nas definições de Dally e Towles (2004).

Neste trabalho, os componentes da biblioteca foram reimplementados e estendidos na forma de *plug-ins* para facilitar a adição de novos componentes. Os atributos arquiteturais suportados pela versão atual da biblioteca estão listados na Tabela 4, a qual evidencia a diversidade de alternativas em relação aos trabalhos supracitados. Destaca-se que o algoritmo de roteamento é específico a cada topologia.

Tabela 4. Atributos arquiteturais da biblioteca de componentes

<table>
<tr><th>Topologia</th><th>Roteamento</th><th>Chaveamento</th><th>Controle de fluxo</th><th>Árbitro</th><th>Memorização</th></tr>
<tr><td>Anel</td><td>Volta Mínima e Restrição 0</td><td rowspan="7">Circuito e wormhole</td><td rowspan="7">Baseado em créditos e handshake</td><td rowspan="7">Distribuído com as políticas de arbitragem: estática, rotativa, randômica e Round-Robin</td><td rowspan="7">Entradas e/ou saídas sem ou com até 8 canais virtuais</td></tr>
<tr><td>Anel Cordal</td><td>Crossfirst</td></tr>
<tr><td>Barramento</td><td rowspan="2">Endereçamento direto</td></tr>
<tr><td>Crossbar</td></tr>
<tr><td>Malha 2D</td><td>XY, WF, NF, NL e Odd-Even</td></tr>
<tr><td>Torus 2D</td><td>DOR</td></tr>
<tr><td>Malha 3D</td><td>XYZXY[1]</td></tr>
</table>

Onde: WF: West-first; NF: Negative-first; NL: North-last; DOR: Dimension Order, XY para o Torus 2D

[1] Roteamento que já considera o uso de TSV (Through Silicon Via)

Para as simulações, um cenário de tráfego pode ser definido para avaliação entre diferentes topologias e cada topologia possui pelo menos um algoritmo de roteamento de caminho mínimo. Essa escolha foi feita na tentativa de estabelecer um critério comum aos roteamentos de diferentes topologias, já que cada topologia possui suas particularidades, bem como para evitar disparidades na avaliação e criar vieses. Os mecanismos de chaveamento, controle de fluxo, arbitragem e memorização podem ser combinados para qualquer configuração. Uma exceção é o barramento, no qual não é possível estabelecer canais virtuais.

As diferentes topologias foram agrupadas em três classes: (*i*) Não-ortogonais (ou de uma dimensão), Barramento, Anel, Anel Cordal e Crossbar; (*ii*) Ortogonais 2D, Malha 2D e Torus 2D; e (*iii*) Ortogonais 3D, a Malha 3D. O formato do cabeçalho do pacote na rede foi ajustado para cada uma das classes de topologia. Os campos de

endereço do cabeçalho permitem endereçar até 256 nodos. Nas redes não-ortogonais, os endereços são compostos de uma palavra de 8 bits. Nas redes ortogonais 2D, os endereços são divididos em coordenadas X e Y, com 4 bits para cada dimensão, o que permite um arranjo máximo de 16x16 nodos. Nas redes 3D, os endereços X, Y e Z são compostos de 3, 3 e 2 bits, respectivamente. O maior arranjo de uma rede 3D é 8x8x4. Nas redes 3D, foram reservados campos para endereçamento individual de TSVs (Xtsv e Ytsv), pois assume-se a possibilidade de que nem todos os nodos possuam links verticais. Porém, por simplificação, considera que um enlace vertical interliga todas as camadas da topologia.

O protocolo de comunicação utiliza dois bits de enquadramento em todos os *flits* (*flow control units* – unidades de controle de fluxo) para indicar se o *flit* é cabeçalho, carga útil ou terminador e pode ter no mínimo 32 bits e no máximo 510 bits. Além da identificação dos endereços de origem e de destino do pacote, o *flit* de cabeçalho contém campos para diferenciação de fluxo nos canais virtuais, identificação de *threads* e estabelecimento/cancelamento de circuito.

4. Experimentos

Os experimentos visaram avaliar cada atributo arquitetural das infraestruturas de comunicação. Os requisitos temporais de tráfego foram definidos e todos os experimentos feitos utilizaram a mesma configuração. A largura de banda requerida pelos fluxos foi fixada em 320 Mbps e a frequência de operação foi variada. Ou seja, é a capacidade da rede que varia, de 320 Mbps/canal (10 MHz) a 3200 Mbps/canal (a 100 MHz), como mostra a Tabela 5. A largura do canal de comunicação é de 32 bits e todos os pacotes possuem 128 bits, quatro *flits* com o cabeçalho.

Tabela 5. Relação entre frequência de operação e o tráfego oferecido

Largura do canal (bits)	Frequência de operação (MHz)	Largura de banda do canal (Mbps)	Largura de Banda Requerida (Mbps)	Tráfego Oferecido (%)
32	100	3200	320	10,0
	90	2880		11,1
	80	2560		12,5
	70	2240		14,3
	60	1920		16,7
	50	1600		20,0
	40	1280		25,0
	30	960		33,3
	20	640		50,0
	10	320		100,0

As distribuições espaciais utilizadas para a definição dos destinos foram: Bit-reversal, Perfect Shuffle, Butterfly, Complemento, Transposto e Uniforme. Geradores de tráfego sintético são usados para se ter total controle sobre o processo de injeção. Como critério de parada, 100.000 pacotes devem ser entregues. Desses, os primeiros 40.000 (40%) foram descartados para reduzir/evitar os vieses sistemáticos de inicialização dos simuladores relacionados ao período de aquecimento da rede. No total, cerca de 1.500 simulações foram executadas (parte delas ilustradas a seguir).

Para as topologias, foram realizados experimentos com 16 e 64 nodos nos arranjos 4x4 e 8x8, para as redes 2D, respectivamente, e 4x2x2 e 4x4x4 nas redes 3D. Para estabelecer um cenário de comparação justo, todas as topologias foram avaliadas com algoritmos de roteamento determinísticos e de caminho mínimo. Os demais atributos são: chaveamento Wormhole; controle de fluxo baseado em créditos; árbitro Round-robin; e *buffers* de entrada com quatro *flits* de capacidade (sem canais virtuais).

Dos experimentos, verificou-se que nem mesmo o Crossbar possui melhor desempenho em todos os cenários. Por exemplo, no tráfego Uniforme nas redes de 16 nodos, o Torus 2D e a Malha 3D foram superiores (Figura 1). Isso porque o tráfego aceito (Figura 1.a) é maior e a latência média (Figura 1.b) é menor. No *jitter* (Figura 1.c), destaca-se a maior concentração de pacotes entregues com latências pequenas e menor dispersão na distribuição, o que também evidencia a superioridade dessas NoCs.

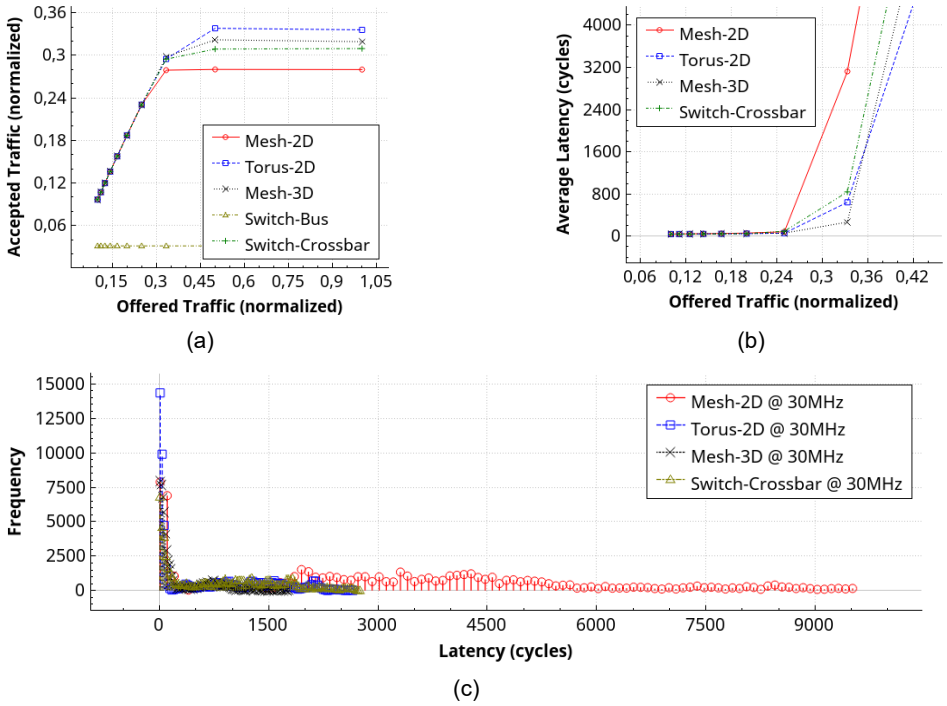


Figura 1. Tráfego Uniforme – topologias de 16 nodos: (a) Tráfego aceito x Tráfego Oferecido; (b) Latência média x Tráfego Oferecido; (c) *Jitter* próximo à saturação

Também foi possível perceber, que a escalabilidade da Malha 3D foi superior às demais configurações de NoC ao aumentar o tamanho do sistema para 64 nodos, sendo inferior apenas ao Crossbar no contexto de todos os cenários de tráfego. Na Figura 2, é possível visualizar essa situação no tráfego Perfect Shuffle. Nas redes com 16 nodos, tanto a latência (Figura 2.a) como a vazão (Figura 2.c) são semelhantes para as topologias Crossbar, Malha 3D e Torus 2D. Já nas redes com 64 nodos, nota-se que a Malha 3D é a NoC com menor latência e que suporta uma maior carga de tráfego até a rede saturar (Figura 2.b). O mesmo vale para o tráfego aceito (Figura 2.d).

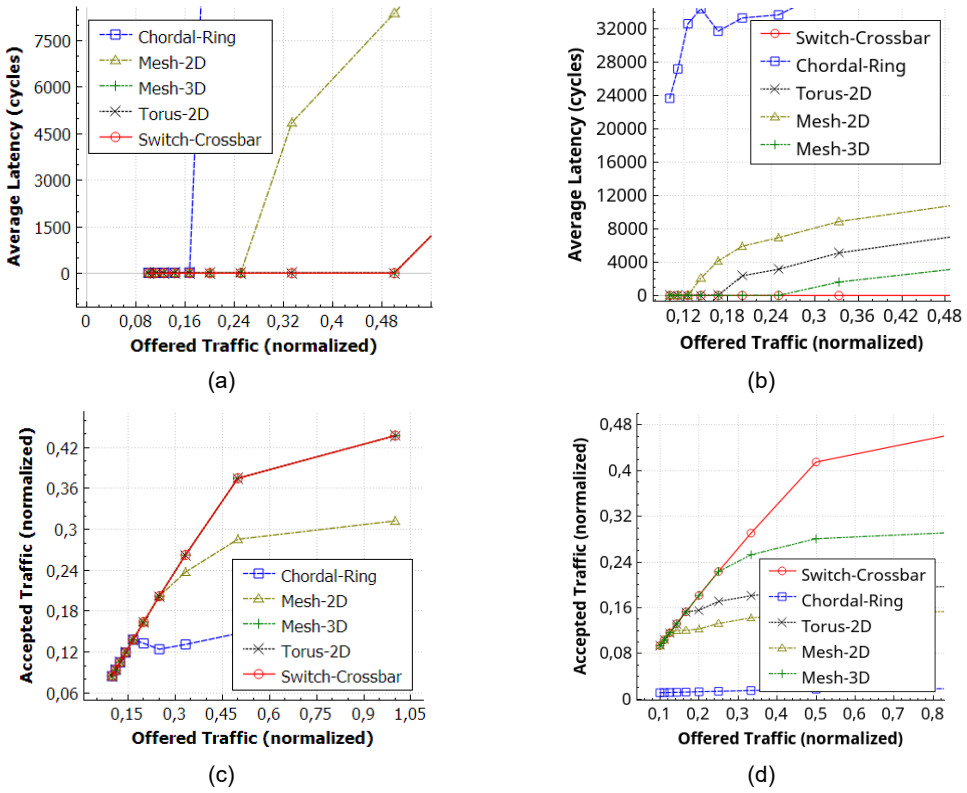


Figura 2. Tráfego Perfect Shuffle – escalabilidade: (a) Latência Média com 16 nodos; (b) Latência Média com 64 nodos; (c) Vazão com 16 nodos; e (d) Vazão com 64 nodos

Os roteamentos foram avaliados na Malha 2D, pois é a topologia com mais algoritmos implementados. Os roteamentos experimentados foram: o XY, o West-first, o Negative-first, o North-last e o Odd-even. O arranjo adotado foi o 4x4, sendo que os demais atributos da rede foram mantidos os mesmos dos experimentos das topologias.

Destaca-se que o algoritmo determinístico XY é o que melhor distribui os fluxos em cenários de alta distribuição de carga pela rede e suas bissecções, tais como o Uniforme e o Complemento. Enquanto os algoritmos adaptativos, melhor distribuem a carga em cenários de tráfego com maior localidade temporal (Bit-reversal, Perfect Shuffle, Butterfly e Transposto). Em cenários globalmente distribuídos, as rotas alternativas podem fazer com que um pacote encontre um caminho mais congestionado nos saltos posteriores. Já em cenários localmente distribuídos, as rotas alternativas tendem a estar disponíveis, o que faz com que mais pacotes enfrentem latências menores.

Ao analisar os algoritmos XY e Odd-even nos cenários de tráfego experimentados, torna-se perceptível o que foi mencionado, tanto na latência média (Figura 3) quanto no tráfego aceito (Figura 4).

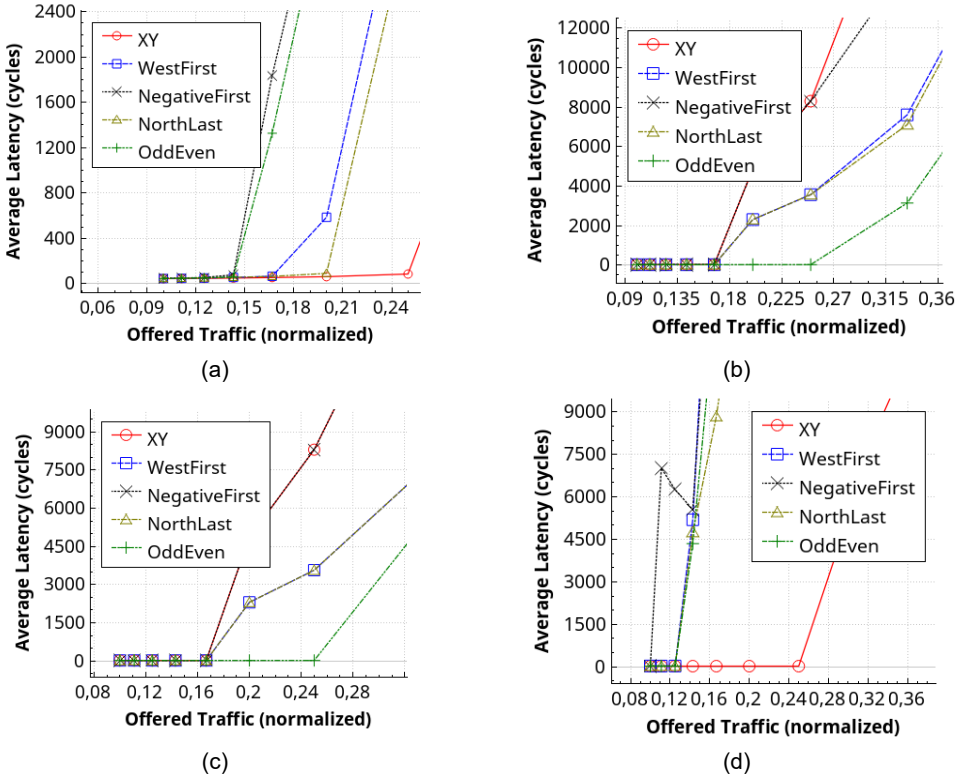


Figura 3. Latência média dos roteamentos na Malha 2D 4x4 sob diferentes cenários de tráfego: (a) Uniforme; (b) Bit-reversal; (c) Transposto; e (d) Complemento

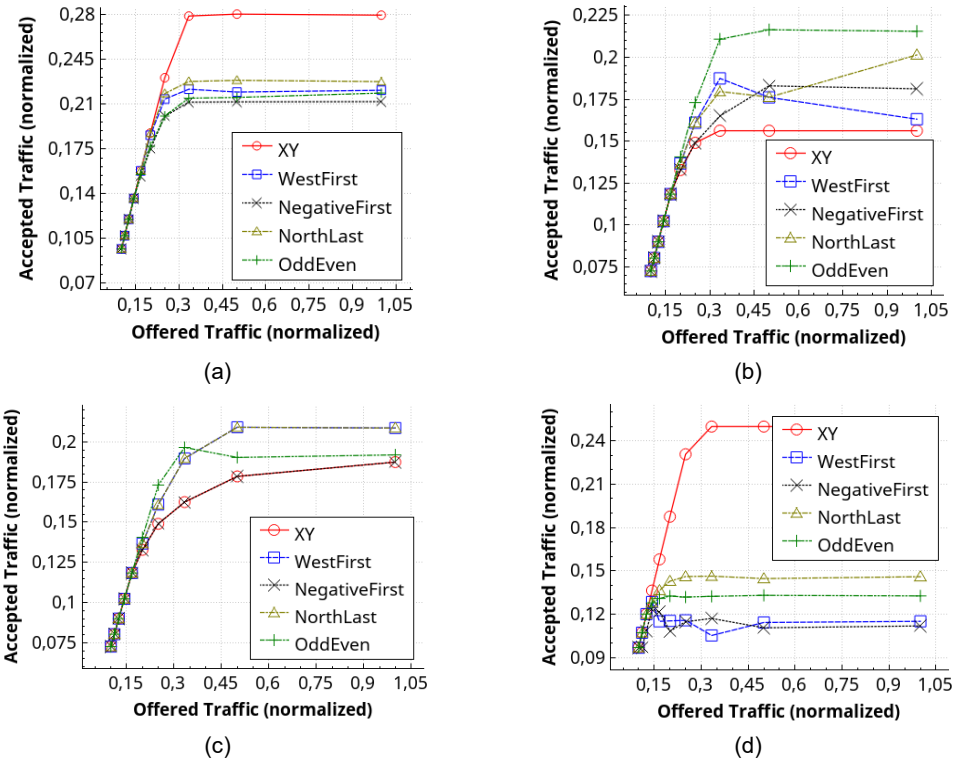


Figura 4. Tráfego aceito dos roteamentos na Malha 2D 4x4 sob diferentes cenários de tráfego: (a) Uniforme; (b) Bit-reversal; (c) Transposto; e (d) Complemento

Dos árbitros, as políticas de priorização Estática, Rotativa, Randômica e Round-robin foram avaliadas. A partir dos resultados obtidos, é possível confirmar que uma política sem equidade, caso do estático, degrada o desempenho das comunicações (Figura 5), conforme já discutido na literatura. Além disso, a entrega dos pacotes pode ser comprometida dependendo dos requisitos de comunicação. Os árbitros dinâmicos apresentaram similaridades nos seus desempenhos e o árbitro estático apresentou instabilidade na rede ao degradar o desempenho após a saturação.

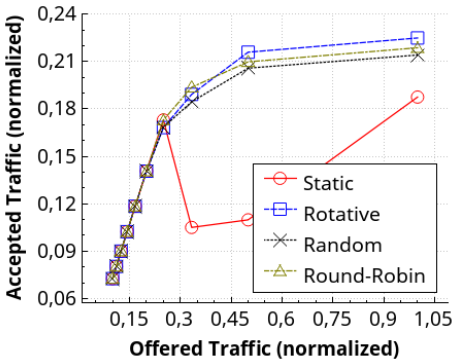


Figura 5. Tráfego aceito para diferentes tipos de árbitro

Por fim, um experimento com diferentes profundidades de *buffer* é discutido. Memórias maiores nos roteadores acarretam em melhor desempenho. Porém, ressalva-se que o ganho é significativo até uma determinada profundidade, a partir da qual o ganho obtido não justifica o custo adicional. A Tabela 6 exibe uma forma alternativa de análise dos dados de desempenho que mostra a relação de ganho de uma configuração sobre outra. As linhas em destaque identificam o melhor resultado obtido em cada métrica. Para a latência média e o *jitter*, apresenta-se quantas vezes a configuração da linha é melhor em relação à da coluna. Para o tráfego aceito, é mostrada a capacidade adicional de tráfego da configuração da linha em relação à da coluna.

Tabela 6. Desempenho para diferentes profundidades de *buffer* sob tráfego Uniforme

Profundidade dos Buffers	IN4	IN8	IN16	IN32	IN16 OUT16	Métrica
IN4		-x-	-x-	-x-	-x-	Latência Média*
IN8	5,83		-x-	-x-	-x-	
IN16	7,29	1,25		-x-	-x-	
IN32	7,75	1,32	1,06		1,03	
IN16 OUT16	7,45	1,27	1,02	-x-		
IN4		-x-	-x-	-x-	-x-	Tráfego Aceito
IN8	4,74		-x-	-x-	-x-	
IN16	6,53	1,79		-x-	-x-	
IN32	6,72	1,98	0,19		0,05	
IN16 OUT16	6,67	1,93	0,14	-x-		
IN4		-x-	-x-	-x-	-x-	Jitter*
IN8	12,67		-x-	-x-	-x-	
IN16	16,51	1,3027		-x-	-x-	
IN32	17,46	1,3777	1,0576		1,0036	
IN16 OUT16	17,39	1,3728	1,0538	-x-		

OBS: IN: Buffer de entrada; OUT: Buffer de saída; INX/OUTX: Buffer com X posições; * Operando a 30 MHz

5. Conclusões

Este artigo apresentou um estudo realizado com o objetivo de comparar diferentes arquiteturas de comunicação intrachip com base em dados quantitativos extraídos por simulação. A partir dos experimentos realizados, é possível afirmar que não há uma única arquitetura de comunicação que possua melhor desempenho em latência e vazão para todos os cenários de comunicação, nem mesmo o Crossbar. Por outro lado, foi possível identificar quais configurações apresentam melhor desempenho em cada cenário de tráfego experimentado. Além disso, alternativas de implementação dos atributos arquiteturais foram avaliadas e seus desempenhos foram relacionados para indicar o quão uma implementação é superior/inferior a outra.

Como trabalhos futuros, pretende-se implementar ferramentas que automatizem a exploração do espaço de projeto e identifiquem a melhor configuração de rede para um determinado modelo de tráfego, considerando suas métricas de desempenho e custo. Também pretende-se disponibilizar as ferramentas e bibliotecas já desenvolvidas para uso em ensino e pesquisa em outras instituições.

Agradecimentos

Os autores gostariam de agradecer o apoio recebido da Capes – Coordenação de Aperfeiçoamento de Pessoal de Nível Superior.

Referências

Accelera (2017). “SystemC”, http://www.accellera.org/downloads/standards/systemc.

Andriahantenaina, A., Charlery, H., Greiner, A., Mortiez, L., Zeferino, C. A. (2003) “SPIN: a scalable, packet switched on-chip micro-network”, In *Design Automation and Test on Europe*, IEEE CS, Los Alamitos, p. 70-73.

Dally, W., Towles, B. (2004). Principles and Practices of Interconnection Networks. San Francisco: Morgan Kaufmann.

Du, G., He, J., Song, Y., Zhang, D., Wu, H. (2013). “Comparison of NoC routing algorithms based on packet-circuit switching”. In *3rd Int. Conf. on Information Science and Technology*, IEEE, Yangzhou, p. 707-710.

Ghidini, Y., Weber, T., Moreno, E., Quadros, I., Fagundes, R., Marcon, C. (2012). “Topological impact on latency and throughput: 2D versus 3D NoC comparison”. In *25th Symp. on Integrated Circuits and Systems Design*, IEEE, Brasília, p 1-6.

Hao, P., Qii, H., Jiaqin, D., Pan, P. (2011). “Comparison of 2D MESH routing algorithm in NOC”. In 9th *Int. Conf. on ASIC*, IEEE, Xiamen, p. 791-795.

Harbin, J., Indrusiak, L. S. (2016). “Comparative performance evaluation of latency and link dynamic power consumption modelling algorithms in wormhole switching networks on chip”. *Journal of Systems Architecture*, New York, v. 63, p. 33-47.

Jaina, A., Kumarb, A., Sharmac, S. (2015). “Comparative Design and Analysis of Mesh, Torus and Ring NoC”. In *Int. Conf. on Computer, Communication and Convergence,* Procedia Computer Science, v. 48, p. 330-337.

Jetly, Krunal (2013). Experimental Comparison of Store-and-Forward and Wormhole NoC Routers for FPGA's. MSc Thesis, Dept. Electrical and Computer Enginnering University of Windsor, Windsor.

Ju, X., Yang, L. (2012). "Performance analysis and comparison of 2 × 4 Network on Chip topology". *Journal of Microprocessors and Microsystems*, New York, v. 36, n. 6, p. 505-509.

Kalimuthu, A., Karthikeyan, M. (2012). "Comparative performance evaluation of power and area Network on Chip (NoC) architectures". In *Int. Conf. on Computing Intelligence and Computing Research*, IEEE, Coimbatore, p. 1-4.

Manna. K., Chattopadhaya, S., Sengupta, I (2012). "An efficient routing technique for mesh-of-tree-based NoC and its performance comparison". *Int. Journal of High Performance Systems Architecture*, Geneva, v. 4, n. 1, p. 25-37.

Ogras, U. Y., Marculescu, R. (2013). *Modeling, Analysis and Optimization of Network-on-Chip Communication Architectures*, Springer, v. 184.

Pandey, D., Gupta, K. (2012). "Comparison between Mesh and Custom Topologies of Network-on-Chip Architectures". *Int. Journal of Scientific Engineering and Technology*, Gulmohar, v. 1, n. 5, p. 243-247.

Parandkar, P., Dalal, J. K., Katiyal, S. (2012). "Performance Comparison of XY, OE and DY Ad Routing Algorithm by Load Variation Analysis of 2-Dimensional Mesh Topology Based Network-on-Chip". *BVICAM Int. Journal of Information Technology*, New Delhi, v. 4, n. 1, p. 391-396.

Radfar, F., Zabihi, M., Sarvari, R. (2014). "Comparison between optimal interconnection network in different 2D and 3D NoC structures". In *27th Int. System-On-Chip Conf.*, IEEE, Las Vegas, p. 171-176.

Romanov, O., Lysenko, O. (2012). "The comparative analysis of the efficiency of regular and pseudo-optimal topologies of Networks-on-Chip based on Netmaker". In *Mediterranean Conf. on Embedded Computing,* IEEE, Bar, p. 13-16.

Sadawarte, Y. A., Gaikwad, M. A., Patrikar, R. M. (2011). "Comparative study of switching techniques for network-on-chip architecture". In: *Int. Conf. on Communication, Computing and Security*, ACM, New York, p. 243-246.

Silva, E. A. (2014). RedScarf: ambiente para avaliação de desempenho de Rede-em-Chip. BSCS Work, Universidade do Vale do Itajaí, Itajaí.

Silva, E. A. (2017). Análise Comparativa do Desempenho de Arquiteturas de Redes-em-Chip baseada em Simulação. M.Sc. Thesis, Universidade do Vale do Itajaí, Itajaí

Sllame, A. M., Abdelkader, A. H. (2014). "A comparative study between fat tree and mesh network-on-chip interconnection architectures". In *14th Middle Eastern Simulation and Modeling Multiconference*, Eurosis, Muscat, p. 31-37.

Tedesco, L., Mello, A., Garibotti, D., Calazans, N., Moraes, F. (2005). "Traffic generation and performance evaluation for mesh-based NoCs". In *18th Symp. On Integrated Circuits and Systems Design*. Florianópolis, p. 184-189.

The Qt Company (2017), "Qt", https://qt.io, August.

Wang, P., Ma, S., Lu, H., Wang, Z. (2014). "A comprehensive comparison between virtual cut-through and wormhole routers for cache coherent network on-chips". *Journal IEICE Electronic Express*, Tokyo, v. 11, n. 14, p. 1-12.

Yin, A., Chen, N., Liljeberg, P., Tenhunen, H. (2012). "Comparison of mesh and honeycomb network-on-chip architectures". In *Conf. on Industrial Electronics and Applications*, IEEE, Singapura, p. 1716-1720.

A Distributed GPU-based Correlation Clustering Algorithm for Large-scale Signed Social Networks

Mario Levorato[1], **Lúcia Drummond**[1], **Rosa Figueiredo**[2], **Yuri Frota**[1]

[1]Instituto de Computação
Universidade Federal Fluminense (UFF) – Niterói, RJ – Brasil

[2]Laboratoire d'Informatique d'Avignon
Université d'Avignon – Avignon, France

{mlevorato,lucia,yuri}@ic.uff.br, rosa.figueiredo@univ-avignon.fr

Abstract. *When applied to signed networks, the Correlation Clustering (CC) problem consists of an important tool to study how balanced a social group behaves and if this group might evolve to a possible balanced state. Solving such combinatorial optimization problem is a challenging task, which heavily relies on heuristic procedures, one of the few solution methods capable of analyzing large network instances. In this work, we present a novel approach to solve the CC problem on large-scale signed networks. A distributed GPU-powered version of the ILS metaheuristic, which benefits from data parallelism, has been developed. This technique provides good quality clustering results when compared to non-distributed methods. Experiments were conducted on both synthetic and real datasets. The proposed algorithm achieved improved solution values when compared to the existing parallel solution method. In particular, one of the largest graphs we have considered in our experiments contains 1 million nodes and 8 million edges – such graph can be clustered in two hours using our algorithm. The new method can process networks for which there is no efficient solution using the existing algorithms found in the literature.*

1. Introduction

A group of individuals can often be viewed as social networks, where vertices represent individuals and edges their relations. Relationships in social networks can have more than two status like presence or absence of a trust/friendship between two individuals. There may be negative links like distrust or dislike. Signed networks are used for this purpose and, within these relationships, communities are formed.

Community detection and clustering on signed networks can provide significant insights into understanding group interactions in social systems and further deducing how a social system evolves and if (or when) the system will reach balanced or relative stable status. Knowledge about communities also allows us to better understand and analyze behaviors of users inside communities, as well as their formation and disintegration.

All these aspects are part of an important theory called social balance or structural balance, originated from early studies of [Heider 1946] and later expanded by [Cartwright and Harary 1956] and [Davis 1967]. The basic ideas underlying structural balance are commonly represented with the aphorisms: "my friend's friend is my friend, my friend's enemy is my enemy, my enemy's friend is my enemy, my enemy's enemy is my friend [Aronson and Cope 1968, Schwartz 2010]. Structural balance theory affirms that human societies tend to avoid tension and conflictual relations [Facchetti et al. 2011, Srinivasan 2011]. In a signed graph which represents a social network, this translates into a level of balance greater than expected, if compared to a random signed graph of equivalent size [Facchetti et al. 2011], and such measure can help us understand interesting social phenomena, like alliances and disputes among parties

or nations [Macon et al. 2012, Doreian and Mrvar 2015] and the social situation where polarization is frequent, like national elections [Srinivasan 2011].

We study the Correlation Clustering (CC) problem applied to measuring structural balance in signed social networks. The CC problem can be stated as: given n objects where certain pairs of objects are labeled as similar and other pairs as dissimilar, find a clustering which maximizes the number of similar pairs within clusters, plus the number of dissimilar pairs between clusters. In fact, the CC problem is not only useful in social network analysis, but also in other research areas: efficient document classification [Bansal et al. 2002], detection of embedded matrix structures [Gülpinar et al. 2004], biological systems [DasGupta et al. 2007], grouping of genes [Bhattacharya and De 2008], community structure [Macon et al. 2012], portfolio analysis in risk management [Huffner et al. 2009] and image segmentation [Kim et al. 2014].

Even though large-scale unsigned social networks (like the ones built from Facebook and Twitter) have been extensively studied [Duch and Arenas 2005, Newman 2006, Brandes et al. 2008], only a few datasets of relatively large signed social networks were released [Kunegis 2013, Leskovec and Krevl 2014]. While these graphs tend to follow a power-law distribution, denser graphs can be obtained by generating edges with similarity measures between all pairs of elements, based on datasets of ratings or voting information, like Movielens [GroupLens 2017] and the European Parliament [Mendonça et al. 2015]. Though these networks may contain less vertices than typical world-scale social networks, they can be harder to analyze, for their high edge density.

Measuring structural balance in large signed networks is complex and time consuming. As other real-world optimization problems, the CC problem is NP-hard [Bansal et al. 2002] and the number of possible clustering configurations is exponential. To tackle this challenge, we first designed parallel master-slave algorithms based on Greedy Randomized Adaptive Search Procedure (GRASP) [Feo and Resende 1995] and Iterated Local Search (ILS) [Lourenço et al. 2003] metaheuristics for the CC problem [Drummond et al. 2013, Levorato et al. 2015b], which outperformed the previous solution methods with similar or improved solution quality. We then developed a parallel local search procedure for the CC problem, accelerated by General Purpose Graphics Processing Units (GPGPUs) [Levorato et al. 2015a]. When dense graphs need to be processed, this procedure reduces the complexity of the local search step of both GRASP and ILS, by analyzing several neighborhood movements in parallel. Recently, in [Levorato et al. 2017], the parallel ILS algorithm for the CC problem was used to cluster the large real-world social networks provided by [Leskovec and Krevl 2014], with up to 10^5 vertices and 10^6 edges.

The need to efficiently process larger and denser networks has motivated us to develop a new fully distributed algorithm, based on heterogeneous computing, using multiple processor types (CPUs and GPUs), task parallelism and distributed data parallelism at the same time. However, clustering signed graphs which are distributed over several compute nodes consists of a challenging subject, since the local search procedures and heuristics in the literature are inherently sequential, as they need to be aware of the whole graph at each step. Also, communication between machines can quickly become a performance bottleneck in many graph applications. We applied dynamic vertex repartitioning during the execution of the distributed algorithm, which demands extra communication required for reassigning data between compute nodes. Besides the local search in the optimization algorithm, dynamic repartitioning is also used as an adaptive load balancing technique, so as to avoid overload of a specific processing node.

In this paper, we propose a distributed optimization algorithm for solving the CC problem on large-scale signed graphs, whose processing would be impossible on a single machine. To the best of our knowledge, since this is the first work on distributed metaheuristics applied to a clustering problem, we do not compare to other distributed techniques as there is no existing comparable technique. As seen in the next section, the nearest works are based on a variation of the CC problem on complete graphs, whose input data and results are not directly comparable. We conduct evaluations on synthetic datasets and real networks. The solution values of the distributed CC algorithm were then compared with the same base metaheuristic (ILS-CC) described in [Levorato et al. 2017], running as a parallel program. Results show the effectiveness of the proposed method.

2. Related work

Correlation clustering was formalized for the first time by [Bansal et al. 2002]. In the general case, the problem of maximizing agreements (minimizing disagreements) is NP-hard and APX-hard (hard to approximate within an arbitrarily small constant) [Bansal et al. 2002, Charikar et al. 2003]. Two variations of the CC problem are known. It can be computed on complete graphs (i.e. all edges are present and all weights are ± 1), or on general graphs (arbitrary edge weights) – the problem studied in this work. Since these variations of the problem require distinct types of input graphs, their solutions cannot be directly compared.

When applied to complete unweighted graphs, the CC problem can be solved via approximation algorithms. [Chierichetti et al. 2014] proposed a parallel 3-approximation algorithm to the optimal CC on complete graphs, which can be implemented in a distributed framework such as MapReduce and scales to huge datasets like Twitter (41M nodes, 2.5B positive edges and 2.9M maximum degree). [Pan et al. 2015] developed parallel CC algorithms with 3-approximation factor, that, although not distributed, can scale to billion-edge graphs, returning a valid solution in less than five seconds.

In this work, we consider the CC problem on general signed graphs. In this case, Integer Linear Programming (ILP) can be used to solve the CC problem optimally, but only when the number of data points is small. For its complexity, the only available solutions for large instances are either heuristic or approximate. The best known approximation ratio for the CC problem is $O(logn)$. [Bansal et al. 2002] proposed two approximation algorithms: one to maximize "agreements" (the number of positive within clusters and negative edges between clusters) and another to minimize disagreements. An approximation algorithm based on rounding a linear program is provided by [Demaine et al. 2006]. [Ailon et al. 2008] proposed a constant factor 2.5 approximation for disagreement minimization (the best known factor so far). Greedy neighborhood-based heuristics for the problem were proposed by [Elsner and Schudy 2009] and [Wang and Li 2013], while in [Yang et al. 2007], the CC problem is known as *community mining* and an agent-based heuristic called FEC is proposed to its solution. A genetic algorithm applied to document clustering has also used the CC problem as objective function [Zhang et al. 2008].

After developing parallel GRASP [Drummond et al. 2013] and ILS [Levorato et al. 2015b] metaheuristics for the CC problem, in [Levorato et al. 2017], we presented a thorough analysis of the sequential and parallel ILS algorithms, in comparison with the aforementioned CC solution approaches proposed in the literature. When a direct comparison was possible, the results evidentiated the superiority of the ILS-CC algorithm, which presented similar or improved solution quality.

3. A Distributed algorithm for solving the Correlation Clustering problem

When it comes to signed social networks, the instances generated from Wikipedia, Slashdot and Epinions websites, available in [Leskovec and Krevl 2014], have thousands of nodes and in some cases almost a million relationships[1]. In order to scale the ILS procedure to solve larger network instances, we extended the existing ILS procedure for the CC problem [Levorato et al. 2015b] to explore the natural data parallelism present in clustering problems: the graph can be split into smaller (non-overlapping) subgraphs and ILS can be used to obtain a clustering solution for each subgraph. The basic idea is to use distributed computing so that each node runs the optimization algorithm over a subset of vertices and then combine all partial solutions (and their respective clustering) into a global solution for the whole network.

3.1. Preliminaries

Let $G = (V, E)$ be an undirected signed graph where V is the set of n vertices and E is the set of edges, where each edge has weight $w_e \geq 0$. Let $s(e)$ denote the label ($\langle - \rangle$, $\langle + \rangle$) of the edge e. Let E^- and E^+ denote, respectively, the set of negative and positive edges in G. Note that the terminology "positive" and "negative" refers to the edge label and not the weight; edge weights are always nonnegative regardless of the label.

For a vertex set $S \subseteq V$, let $E[S] = \{(i, j) \in E \mid i, j \in S\}$ denote the *subset of edges induced by* S. For two vertex sets $S, W \subseteq V$, let $E[S : W] = \{(i, j) \in E \mid (i \in S, j \in W) \vee (i \in W, j \in S)\}$ denote the *subset of edges that connect vertices from the clusters* S *and* W. Given a clustering $\mathcal{C} = \{S_1, S_2, \ldots, S_k\}$, for $1 \leq i, j \leq k$, let

$$\Omega^+(S_i, S_j) = \sum_{e \in E^+ \cap E[S_i:S_j]} w_e \text{ and } \Omega^-(S_i, S_j) = \sum_{e \in E^- \cap E[S_i:S_j]} w_e.$$

We call an edge $e = (u, v)$ a positive mistake if $s(e) = \langle + \rangle$ and $e \in E[S_i : S_j] : S_i, S_j \in \mathcal{C}, i \neq j$. We call an edge $e = (u, v)$ a negative mistake if $s(e) = \langle - \rangle$ and $e \in E[S_i : S_i]$ for some $S_i \in \mathcal{C}$. The number of mistakes or *imbalance* of a clustering $[I(\mathcal{C})]$, is given by the sum of positive and negative mistakes or, in other words, the weighted sum of unrelated pairs that are clustered together, in addition to the weighted sum of related pairs that are separate.

$$I(\mathcal{C}) = \sum_{1 \leq i \leq k} \Omega^-(S_i, S_i) + \sum_{1 \leq i < j \leq k} \Omega^+(S_i, S_j). \quad (1)$$

That being said, a formal definition to the CC problem can be provided.

Problem 3.1 (CC problem) *Let $G = (V, E)$ be a signed graph and w_e be a nonnegative edge weight associated with each edge $e \in E$. The correlation clustering problem is the problem of finding a clustering $\mathcal{C}$ of V such that the imbalance $I(\mathcal{C})$ is minimized.*

A *partition* of V is a division of V into non-overlapping and non-empty subsets and a graph $G' = (V', E')$ is called a *subgraph* of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. G' is called an *induced subgraph*, or the *subgraph induced* by V', if E' consists of all edges of G spanned by V'.

Now suppose the set V of vertices of G is *partitioned* into q subsets, one for each process running in parallel, such that $\Phi = \{V_1, V_2, \ldots, V_q\}$ denotes a *vertex-process par-*

[1]Epinions signed social network has 131828 vertices and 711210 edges.

tition of V. This way, $\forall i \in \{1, \ldots, q\}$, each process q_i will be in charge of manipulating a subgraph G'_i, induced by V_i.

3.2. Distributed GPU-powered Iterated Local Search (ILS) algorithm for the CC Problem

The Iterated Local Search [Lourenço et al. 2003] (ILS) is a metaheuristic that explores a sequence of solutions created by perturbations of the current best solution and then refines these solutions to their local optima using an embedded heuristic. According to our tests, within a 2-hour time limit, the multistart ILS procedure ($ILSMultiStartCC$) from [Levorato et al. 2015b], embedded in our distributed algorithm, is capable of efficiently solving the CC problem on real-world networks of size up to $n = 80,000$ and edge density $d \leq 0.02\%$. To process larger graphs, we opted to develop a distributed algorithm that splits large-scale network instances into smaller subgraphs, delegating the solution of the CC problem on each subgraph to a specific process. Therefore, all processes execute the same code (the $ILSMultiStartCC$ procedure) over a different subgraph.

Message passing was used for communication among processes. When q processors are used, a master process reads the problem data, divides the graph into smaller parts and passes them to the remaining $q - 1$ processes. Each process executes a copy of ILS over a specific subgraph it was assigned to, according to the vertex partitioning (each vertex subset goes to a process running on a separate CPU). Finally, the master process is in charge of gathering each search result produced in parallel and merging them to a global solution for the whole graph.

A second parallelization strategy used in this algorithm was applied in the local search phase (inside the $ILSMultiStartCC$ procedure), using parallelism to evaluate hundreds of vertex movements at once with the help of General Purpose Graphics Processing Units (GPGPUs). It consists of a parallel local search procedure, known as CUDA-VND [Levorato et al. 2015a], which outperforms the previous local search procedure (which used sequential code) in execution time, presenting similar solution quality.

We also applied distributed data parallelism to our algorithm by the use of special data structures from the Boost Parallel Graph Library [Gregor and Lumsdaine 2005]. Distributing graph vertices and edges between several compute nodes allowed the processing of extremely large graphs which would not fit into a single machine. This allows the algorithm to efficiently scale to graphs in the order of 10^6 nodes and number of edges above 10^9, without the limitations of a single shared memory address space.

The distributed algorithm includes an initial distribution of vertices among processes (Algorithm 1) and an optimization algorithm executed in a distributed fashion with a load balance mechanism, which also includes termination detection (Algorithms 2 and 3).

3.2.1. Initial distribution

The signed graph is split according to the number of vertices $n = |V|$ and the number of processes running in parallel (q parameter). Each process will be initially in charge of finding a solution for the subgraph induced by the set V_i, where $|V_i| = |V| / q$.

The $SplitGraphBetweenProcesses$ procedure (Algorithm 1) generates an initial vertex-process partition Φ, comprised of q subsets, from V_1 to V_q, and delegates to each process q_j the task of finding the solution of the CC problem on the corresponding subgraph induced by V_j. Let R be the set of vertices not yet associated to any process. For each process q_j, the procedure initially adds to the subset V_j the vertex $v \in R$ with the smallest negative-edge sum considering the residual subgraph induced by set R (line 4).

Then, at each step, the vertex v_{max} is added to the referred partition. This vertex, which has not been previously inserted in V_j, is chosen so that it presents the maximum cardinality of positive edges between itself and V_j (line 6). In other words, this criterion tends to minimize the number of mistakes of the initial partition, since vertices with higher affinity (i.e. more positive edges) will be included in the same subgraph (same process). At the end, each subset V_j in the initial partition Φ is forwarded to the corresponding process (line 10).

Algorithm 1: $SplitGraphBetweenProcesses$ (master procedure)

Variables: $G = (V, E)$ and number of processes q

1 $R \leftarrow V$;
2 $\overline{\mathcal{C}} \leftarrow \emptyset, \Phi \leftarrow \emptyset$;
3 **for** *each process* q_j *such that* $1 \leq j \leq q$ **do**
4 $\quad V_j \leftarrow \{\arg\min\{\Omega^-(R, \{v\}) \mid v \in R\}\}$;
5 $\quad$ **while** $\left(|V_j| < \dfrac{|V|}{q}\right)$ **do**
6 $\quad\quad v_{max} \leftarrow \arg\max\{\Omega^+(V_j, \{v\}) \mid v \in (R \setminus V_j)\}$;
7 $\quad\quad V_j \leftarrow V_j \cup \{v_{max}\}$;
8 $\quad R \leftarrow R \setminus V_j$;
9 $\quad \Phi \leftarrow \Phi \cup V_j$;
10 $\quad$ **send message** $(\langle InitialDistribution\rangle, V_j)$ to process q_j;

In a second version of the initial distribution, Algorithm 1 was replaced by a uniformly random distribution of vertices between the processes, which scales to huge graph instances. We applied such distribution approach only when the graph was too big for a single machine's memory. As seen in the next section, experiments performed with the most dense graph instances confirmed the solution quality of our method.

3.2.2. Distributed optimization and load balance

Algorithm 2: $DistributedILS$ (master procedure)

Variables: $G = (V, E)$, vertex-process partition Φ, clustering $\mathcal{C}$, number of processes q

1 $r \leftarrow 1$, $\mathcal{C} \leftarrow \emptyset$, $I(\mathcal{C}) \leftarrow \infty$;
2 **while** $r \leq 4$ and time limit not exceeded **do**
3 $\quad \overline{\Phi} \leftarrow \Phi$;
4 $\quad$ `UpdatePartition` ($\overline{\Phi}$, *MovementType(r)*);
5 $\quad$ **for** *each process* q_j *where* $1 \leq j \leq q$ **do**
6 $\quad\quad$ **send message** $(\langle MovementType(r)\rangle, \overline{\Phi})$ to process q_j;
7 $\quad \overline{\mathcal{C}} \leftarrow \emptyset$;
8 $\quad$ **for** *each process* q_j *where* $1 \leq j \leq q$ **do**
9 $\quad\quad$ **receive message** $(\langle ILSResult\rangle, \mathcal{C}_j)$ from process q_j;
10 $\quad\quad \overline{\mathcal{C}} \leftarrow \overline{\mathcal{C}} \cup \mathcal{C}_j$;
11 $\quad$ **if** $I(\overline{\mathcal{C}}) < I(\mathcal{C})$ **then**
12 $\quad\quad \mathcal{C} \leftarrow \overline{\mathcal{C}}$, $\Phi \leftarrow \overline{\Phi}, r \leftarrow 1$;
13 $\quad$ **else**
14 $\quad\quad r \leftarrow r + 1$; `// Go to the next VND neighborhood`
15 **return** $\Phi, \mathcal{C}$;

After the initial distribution of vertices, the master executes $DistributedILS$ (Algorithm 2), which is based on the Variable Neighborhood Descent (VND) [Mladenović and Hansen 1997] procedure. It is in charge of exploring four different distributed neighborhood structures (vertex/cluster movements between process partitions), listed in Figure 1. The master generates new vertex partitionings Φ, distributed among the processes, allowing that a better global solution be reached if and only if an improvement of imbalance for the whole graph is obtained. The master procedure

sends, to each worker process q_j, the requested neighborhood movement type and the new vertex-process partition $\overline{\Phi}$. After receiving the processing results, the individual CC solution of each modified subgraph ($\mathcal{C}_j$) is then merged into a global solution $\overline{\mathcal{C}}$ for the whole graph (lines 9-10) and the corresponding global imbalance $I(\overline{\mathcal{C}})$ is tested for improvement (line 11). If this is the case, the new incumbent is updated and r is returned to its initial value (line 12). Otherwise, the next neighborhood is considered (line 14). The algorithm halts when no better clustering solution is found in the most distant neighborhood of the current best solution ($\mathcal{C}$) or if the time limit is exceeded.

Algorithm 3 summarizes the tasks performed by each worker process. The initial distribution of vertices is received in the beginning of the program. The main loop is then responsible for processing the messages received from the master process, which contain the neighborhood movement requested. A load balancing procedure (line 4) tries to rebalance the number of vertices in each subgraph, to prevent a process from being overloaded. This first step is fast and computationally inexpensive, since moving clusters of the current solution between processes causes no change in the CC solution value. Afterwards, the neighborhood movement (requested by the master process) is performed (line 5), as depicted in Figure 1. Then, each worker process runs the $ILSMultiStartCC$ procedure on its modified subgraph $G'(V_j)$ (line 7). Finally, on the next line, the local ILS clustering result $\mathcal{C}_j$ is sent back to the master process q_m.

Algorithm 3: $DistributedILS$ (worker procedure)

```
   Variables: G = (V, E), clustering C_j, number of processes q
1  receive message (⟨InitialDistribution⟩, V_j) from process q_m;
2  while not terminate do
3      receive message (⟨MovementType⟩, Φ) from process q_m;
4      Φ̄ ← LoadBalance (Φ,q);
5      V_j ← Repartition (⟨MovementType⟩, Φ̄);
6      G' ← subgraph induced by V_j ∈ Φ̄;
7      C_j ← ILSMultiStartCC (G');
8      send message (⟨ILSResult⟩, C_j) to process q_m;
9      if termination detected then
10         terminate ← true;
```

Remark that the objective function calculation also benefits from parallelism. For every change in the clustering, a reduction operation is in charge of receiving the local solution value of each subgraph (lines 9-10 in Algorithm 2), from every participating process. Based on these values, the change in global solution is then computed according to the new CC clustering $\mathcal{C}$, as well as the vertex-process partition Φ and the edges that connect vertices from different processes.

The most complex neighborhood structure is $MoveQuasiClique$ (MovementType(4) in Figure 1). The algorithm identifies vertex-overloaded processes, i.e. processes with more than (n/q) vertices. Then, for each process that falls into this category, the procedure tries to identify and move a subset of vertices $\mathbb{K}$ (here defined as a *quasi-clique* – an almost complete subgraph [Brunato et al. 2007]) from an existing cluster to a new cluster in another process. A *quasi-clique* $\mathbb{K}$ exhibits high affinity, that is, high density of internal positive edges and, at the same time, low density of internal negative edges, and therefore constitutes a good subset of vertices to be moved.

3.2.3. Termination detection

The master process finishes its work and terminates when no additional improvement can be found based on the current best solution or if the time limit is exceeded. In other words, the procedure in Algorithm 2 returns when no better clustering solution

is found in the most distant neighborhood of the current solution $\mathcal{C}$. This propagates a termination message to all worker processes as well.

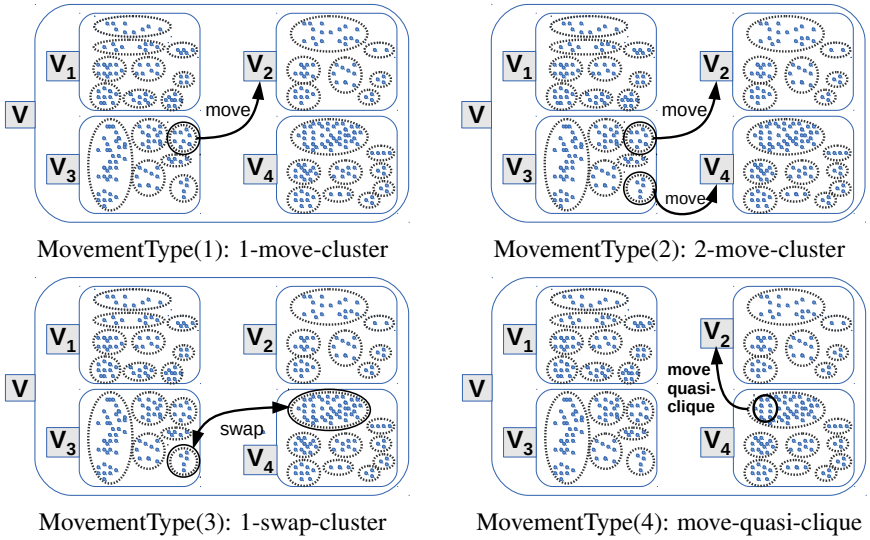


Figure 1. Distributed ILS neighborhood structures. A cluster movement between processes involves at most three modified subgraphs, thus requiring three processes to execute the local ILS procedure (e.g. *2-move-cluster* moves two clusters – and corresponding vertices – from one process subgraph to two different processes). The remaining (idle) processes can be then used to compute other neighborhood movements in parallel.

4. Experiments

The goal of these experiments is to assess the performance of the $DistributedILS$ algorithm using different sets of signed graph instances, when compared to our previous best solution technique, the Independent Parallel $ILS - CC$ algorithm ($ParILS$) [Levorato et al. 2017], which from now on will be called *baseline*. The local search procedure used by both algorithms (*baseline* and $DistributedILS$) runs on the GPU [Levorato et al. 2015a].

4.1. Computational environment

The algorithms described in the previous section were implemented in ANSI C++ (GCC v4.4.7-11), MPI (OpenMPI v1.2.8.4) for message passing and Boost Parallel BGL (v1.61.0) for graph data parallelism. All experiments were performed (with exclusive access) on the SDumont cluster [LNCC 2017] with 198 nodes, each one with two Intel Xeon E5-2695v2 Ivy Bridge @2.4GHz processors (12 cores each) and 64GB of RAM under RedHat Linux 6.4. Each node is equipped with 2 NVIDIA Tesla K40 GPUs containing 12GB of RAM and 2880 CUDA cores. CUDA code was written in "C for CUDA V6.5" [NVIDIA 2014]. The presented results were obtained from 50 independent runs.

4.2. Test problems

Computational experiments were undertaken on (i) a set of 2 social networks from the literature, (ii) a set of 3 network instances generated from MovieLens movie ratings website, and (iii) a set of 6 random signed networks with a predefined community structure. We will briefly describe these instances[2].

(i) This set of instances is composed by two signed networks widely used in the related literature: the first representing the large scale social network of technology-related news website Slashdot [Leskovec et al. 2010, Facchetti et al. 2011], and

[2]All instances are available in `http://www.ic.uff.br/~yuri/CCinst-large.html`.

the second one contains the Epinions [Epinions 1999] signed network, an on-line review website which allows users to either like or dislike other people's reviews. Since both networks are originally signed digraphs, they were converted to undirected graphs.

(ii) We proposed three new signed social networks based on movie ratings given by each user from the MovieLens Website. The dataset used was MovieLens 20M (20 million ratings and 465,000 tag applications applied to 27,000 movies by 138,000 users; Released on 4/2015) [GroupLens 2017]. The signed graphs were generated according to the following algorithm. Let the movie rating be an integer between 1 and 5. For each pair of users, we identify the list of movies both users have rated. Based on this list, we totalize the number of similar ratings each user gave to the same movie. A movie rating from users a and b is similar if both users gave a rating of 1 or 2 (bad movie), or if both users rated the movie with 3 (regular movie), or if both users rated the movie with 4 or 5 (good movie). We then normalize the ratings by subtracting from each user rating the its own average rating, in order to prevent problems with the difference of scaling between users. Based on this data, a user-user matrix $D = \{d_{uv} : u, v \in V\}$ of cosine distance similarities is then calculated. Each value is in the range of [-1,1], where -1 represents perfect disagreement and 1 means perfect agreement between users. If the absolute value of d_{uv} is greater than a given threshold mw, we add a positive or negative edge between users u and v, according to the sign of d_{uv}.

(iii) We generated six large random signed networks with a predefined community structure, according to [Yang et al. 2007]. The random signed network is defined as $SG(c, n, k, p_{in}, p_-, p_+)$, where c is the number of communities in the network, n is the number of nodes in each community, k is the degree of each node, p_{in} is the probability of each node connecting other nodes in the same community, p_- denotes the probability of negative links appearing within communities, and p_+ denotes that of positive links appearing between communities. Each generated instance has a different value for the parameter c. The other parameters were fixed to $n = 4096$, $k = 16$, $p_{in} = 0.8$, $p_- = 0.8$ and $p_+ = 0.6$.

Instance		$\|V\|$	$\|E\|$	**Instance**		$\|V\|$	$\|E\|$
Slashdot		82144	500481	Random	$c = 8$	32768	262144
Epinions		131828	711210		$c = 16$	65536	524284
MovieLens	$mw = 0.90$	138494	371118		$c = 32$	131072	1048576
	$mw = 0.85$	138494	3451683		$c = 64$	262144	2097152
	$mw = 0.80$	138494	10211818		$c = 128$	524288	4194304
					$c = 256$	1048576	8388608

Table 1. Dimensions of each signed graph instance $|V|$: number of vertices, $|E|$: number of edges.

4.3. Obtained results

Baseline was configured to use 10 processes running in parallel, equally dividing the number of multistart iterations of the original $ILSMultiStartCC$ procedure. In $DistributedILS$, the runtime configuration used 8 processes for instances with the number of vertices $|V| \leq 5 \times 10^5$, and 16 processes for larger graphs. The evaluation of the solution values was carried out by means of an ANOVA analysis [Montgomery 2005] at 99.9% confidence level. In the presented results, since the p-value is is always less than the significance level of 0.001, we can reject the null hypothesis and conclude that the means are significantly different. After running both algorithms for 2 hours, $DistributedILS$ has enhanced the solution quality on Epinions instance (Figure 2-b) with an average improvement of 9.9%. On the other hand, when solving the Slashdot instance (Figure 2-a),

the *baseline* achieved the best solution value, but with a slight average gap (smaller than 0.8%) when compared to $DistributedILS$.

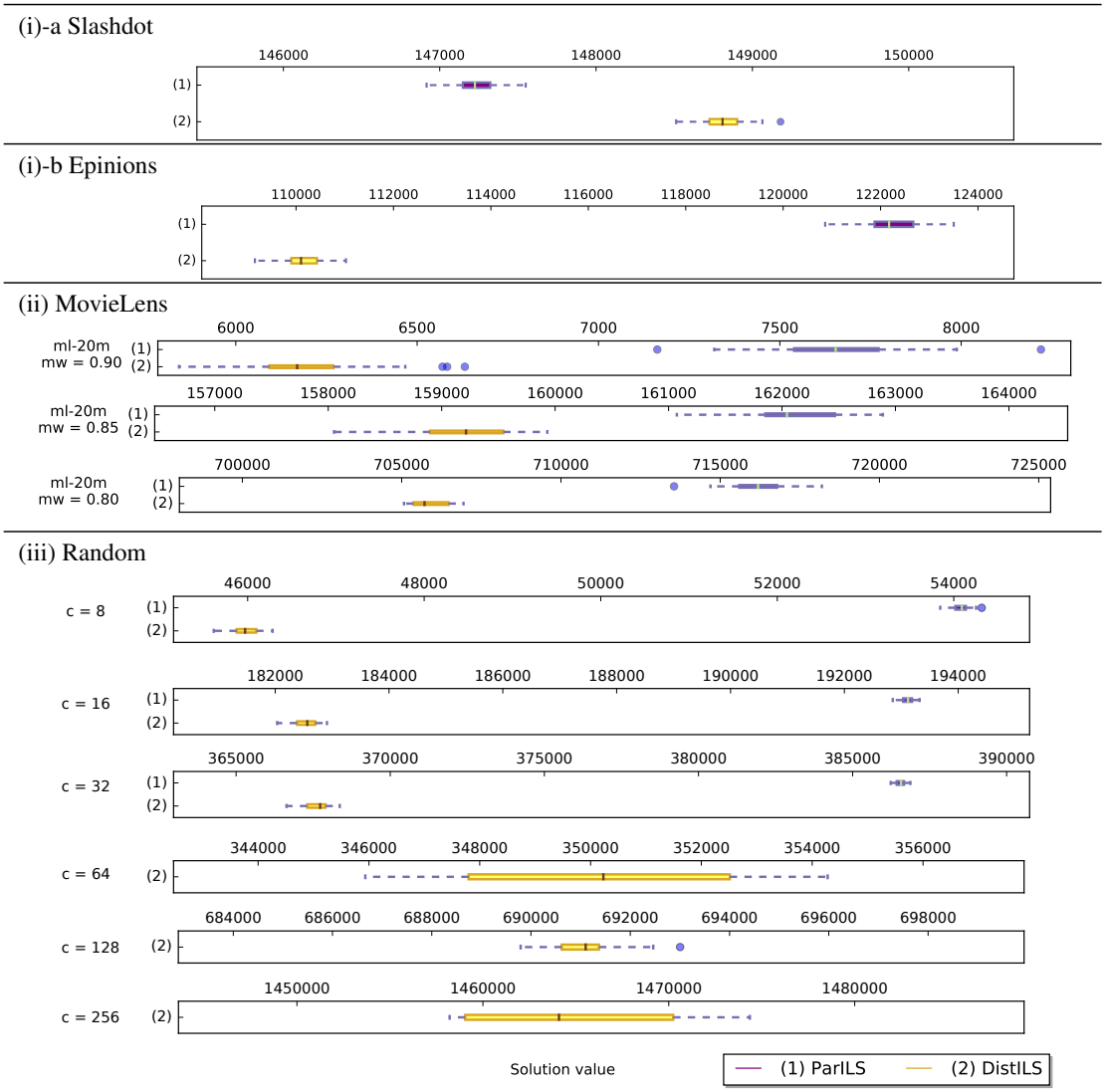


Figure 2. $Baseline$ **(ParILS) and** $DistributedILS$ **(DistILS) CC results for literature instances (i), for Movielens network instances (ii), and for random networks with predefined community structure (iii). The boxplot shows the distribution of the solution values obtained by each algorithm after running for** 2 **hours, considering** 50 **independent executions.**

Regarding the solution of the MovieLens instances in (ii), as shown on Figure 2, an analysis of the gap in solution values between $DistributedILS$ and *baseline* also indicates the superiority of the proposed method. For the $mw = 0.90$ instance, the solution values were 19.23% smaller than those obtained by *baseline*. When solving larger instances with $mw = 0.85$ and $mw = 0.80$, $DistributedILS$ presented solutions with average improvements of 1.83% and 1.38%. Finally, when solving the random instances with predefined community structure in (iii), the proposed algorithm returned an improved solution on test instances $c = 8$, $c = 16$ and $c = 32$ (Figure 2-(iii)). The average gaps in solution values were -14.96%, -5.47% and -4.87%, respectively. The *baseline* method was not able to process the remaining instances ($c \geq 64$), either for not providing an initial solution within the $2h$ time limit or due to lack of memory. Therefore, in these cases, we only display the solution values obtained by $DistributedILS$.

5. Concluding remarks

We developed the first distributed algorithm for efficiently solving the CC problem, based on a heterogeneous computing platform. It iteratively repartitions the graph between processes, invoking the ILS metaheuristic locally on each node and merging individual results into a global solution. Experiments were conducted on both synthetic and real datasets. On the synthetic dataset our approach is able to scale to 1,048,576 nodes and 8,388,608 edges. The proposed algorithm can provide efficient solutions to the CC problem when traditional metaheuristics fail due to the need to be aware of the entire graph, relying on a single memory space.

Many future research topics could be built upon this framework, including a fully distributed heuristic for building an initial global solution. Larger signed graph instances (both in the number of vertices and edges) may be generated as well.

Acknowledgement

The authors acknowledge the National Laboratory for Scientific Computing (LNCC/MCTI, Brazil) for providing HPC resources of the SDumont supercomputer, which have contributed to the research results reported within this paper. URL: `http://sdumont.lncc.br` We also thank the National Council for Scientific and Technological Development (CNPq) Cnpq-Universal project number 443883/2014-9.

References

Ailon, N., Charikar, M., and Newman, A. (2008). Aggregating inconsistent information: ranking and clustering. *Journal of the ACM (JACM)*, 55(5):23.

Aronson, E. and Cope, V. (1968). My enemy's enemy is my friend. *Journal of personality and social psychology*, 8(1p1):8.

Bansal, N., Blum, A., and Chawla, S. (2002). Correlation clustering. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 238–250, Vancouver, Canada. IEEE.

Bhattacharya, A. and De, R. K. (2008). Divisive correlation clustering algorithm (dcca) for grouping of genes: detecting varying patterns in expression profiles. *bioinformatics*, 24(11):1359–1366.

Brandes, U., Delling, D., Gaertler, M., Gorke, R., Hoefer, M., Nikoloski, Z., and Wagner, D. (2008). On modularity clustering. *Knowledge and Data Engineering, IEEE Transactions on*, 20(2):172–188.

Brunato, M., Hoos, H. H., and Battiti, R. (2007). On effectively finding maximal quasi-cliques in graphs. In *International conference on learning and intelligent optimization*, pages 41–55. Springer.

Cartwright, D. and Harary, F. (1956). Structural balance: A generalization of heider's theory. *Psychological Review*, 63(5):277–293.

Charikar, M., Guruswami, V., and Wirth, A. (2003). Clustering with qualitative information. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 524–533. IEEE.

Chierichetti, F., Dalvi, N., and Kumar, R. (2014). Correlation clustering in mapreduce. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM.

DasGupta, B., Encisob, G. A., Sontag, E., and Zhanga, Y. (2007). Algorithmic and complexity results for decompositions of biological networks into monotone subsystems. *BioSystems*, 90(1):161–178.

Davis, J. (1967). Clustering and structural balance in graphs. *Human Relations*, 20(2):181–187.

Demaine, E. D., Emanuel, D., Fiat, A., and Immorlica, N. (2006). Correlation clustering in general weighted graphs. *Theoretical Computer Science*, 361(2):172–187.

Doreian, P. and Mrvar, A. (2015). Structural balance and signed international relations. *Journal of Social Structure*, 16:1.

Drummond, L., Figueiredo, R., Frota, Y., and Levorato, M. (2013). Efficient solution of the correlation clustering problem: An application to structural balance. In *Lecture Notes in Computer Science*, pages 674–683. Springer Nature.

Duch, J. and Arenas, A. (2005). Community detection in complex networks using extremal optimization. *Physical review E*, 72(2):027104.

Elsner, M. and Schudy, W. (2009). Bounding and comparing methods for correlation clustering beyond ilp. In *Proceedings of the Workshop on Integer Linear Programming for Natural Langauge Processing*, ILP '09, pages 19–27, Stroudsburg, PA, USA.

Epinions (1999). Website. URL http://www.epinions.com. Accessed on March 2015.

Facchetti, G., Iacono, G., and Altafini, C. (2011). Computing global structural balance in large-scale signed social networks. In *Proceedings of the National Academy of Sciences of the United States of America*,

volume 108, pages 20953–20958.

Feo, T. A. and Resende, M. G. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133.

Gregor, D. and Lumsdaine, A. (2005). The parallel bgl: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2:1–18.

GroupLens (2017). Movielens dataset collection. `https://grouplens.org/datasets/movielens`.

Gülpinar, N., Gutin, G., Mitra, G., and Zverovitch, A. (2004). Extracting pure network submatrices in linear programs using signed graphs. *Discrete Applied Mathematics*, 137:359–372.

Heider, F. (1946). Attitudes and cognitive organization. *Journal of Psychology*, 21(1):107–112.

Huffner, F., Betzler, N., and Niedermeier, R. (2009). Separator-based data reduction for signed graph balancing. *Journal of Combinatorial Optimization*, 20(4):335–360.

Kim, S., Yoo, C. D., Nowozin, S., and Kohli, P. (2014). Image segmentation UsingHigher-order correlation clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(9):1761–1774.

Kunegis, J. (2013). Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM.

Leskovec, J., Huttenlocher, D., and Kleinberg, J. (2010). Signed networks in social media. In *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*, pages 1361–1370. Association for Computing Machinery (ACM).

Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

Levorato, M., Drummond, L., Frota, Y., and Figueiredo, R. (2015a). A GPU-accelerated local search algorithm for the Correlation Clustering problem. In *Proceedings of the XLVII Brazilian Symposium on Operations Research*, Porto de Galinhas, PE, Brazil.

Levorato, M., Drummond, L., Frota, Y., and Figueiredo, R. (2015b). An ils algorithm to evaluate structural balance in signed social networks. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1117–1122.

Levorato, M., Figueiredo, R., Frota, Y., and Drummond, L. (2017). Evaluating balancing on social networks through the efficient solution of correlation clustering problems. *EURO Journal on Computational Optimization*, pages 1–32.

LNCC (2017). Santos dumont supercomputer. http://sdumont.lncc.br.

Lourenço, H. R., Martin, O. C., and Sttzle, T. (2003). Iterated local search. In *Handbook of Metaheuristics*, pages 320–353. Springer Nature.

Macon, K., Mucha, P., and Porter, M. (2012). Community structure in the united nations general assembly. *Physica A: Statistical Mechanics and its Applications*, 391(1-2):343–361.

Mendonça, I., Figueiredo, R., Labatut, V., and Michelon, P. (2015). Relevance of negative links in graph partitioning: A case study using votes from the european parliament. In *2015 Second European Network Intelligence Conference*, pages 122–129. IEEE.

Mladenović, N. and Hansen, P. (1997). Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100.

Montgomery, D. (2005). Design and analysis of experiments (6th ed) john wiley and sons. *New York, NY*.

Newman, M. E. (2006). Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582.

NVIDIA, C. (2014). Toolkit v6. 5. *URL http://docs.nvidia.com/cuda/cuda-c-programming-guide*.

Pan, X., Papailiopoulos, D., Oymak, S., Recht, B., Ramchandran, K., and Jordan, M. I. (2015). Parallel correlation clustering on big graphs. In *Advances in Neural Information Processing Systems*, pages 82–90.

Schwartz, T. (2010). The friend of my enemy is my enemy, the enemy of my enemy is my friend: Axioms for structural balance and bi-polarity. *Mathematical Social Sciences*, 60(1):39–45.

Srinivasan, A. (2011). Local balancing influences global structure in social networks. In *Proceedings of the National Academy of Sciences of the United States of America*, volume 108, pages 1751–1752.

Wang, N. and Li, J. (2013). Restoring: A greedy heuristic approach based on neighborhood for correlation clustering. In *Advanced Data Mining and Applications*, pages 348–359. Springer.

Yang, B., Cheung, W., and Liu, J. (2007). Community mining from signed social networks. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1333–1348.

Zhang, Z., Cheng, H., Chen, W., Zhang, S., and Fang, Q. (2008). Correlation clustering based on genetic algorithm for documents clustering. In *2008 IEEE Congress on Evolutionary Computation*, pages 3193–3198.

Algoritmo Paralelo para Árvore Geradora usando GPU

Jucele F. A. Vasconcellos[1], **Edson N. Cáceres**[1], **Henrique Mongelli**[1], **Siang W. Song**[2]

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

[2]Instituto de Matemática e Estatística – Universidade de São Paulo (USP)
São Paulo – SP – Brazil

{jucele,edson,mongelli}@facom.ufms.br, song@ime.usp.br

Resumo. *Neste trabalho, usando o modelo BSP/CGM, propomos um algoritmo paralelo, com uma implementação em CUDA, para obter uma árvore geradora de um grafo. Trabalhos anteriores para este problema são baseados na solução do problema de list ranking que, embora eficiente na teoria, não produz bons ganhos na prática. Num trabalho posterior, baseado na ideia do cálculo de uma estrutura chamada esteio, Cáceres et al. propuseram um algoritmo paralelo no modelo BSP/CGM para obter uma árvore geradora sem a utilização de list ranking. O cálculo do esteio é obtido com a utilização de um grafo bipartido auxiliar, com o uso de ordenação inteira. Neste artigo melhoramos aquele trabalho em vários aspectos. No algoritmo proposto, para implementação em GPGPU, não é mais necessário calcular o grafo bipartido, e a construção do esteio não necessita do algoritmo de ordenação. A eficiência e escalabilidade do algoritmo proposto são verificadas por experimentos.*

1. Introdução

A computação da árvore geradora de um grafo é um dos principais problemas da área de Teoria dos Grafos, e com grande quantidade de aplicações na área de Computação. Vários algoritmos usam a computação de uma árvore geradora como um procedimento intermediário, o que motiva a busca por algoritmos eficientes para a determinação da árvore geradora de um dado grafo. Existem algoritmos sequenciais ótimos, baseados em busca em profundidade e em busca em largura [Karger et al. 1995], para computar a árvore geradora de um dado grafo. Considerando que os grafos de vários problemas reais têm um tamanho muito grande, apesar do fato de os algoritmos sequenciais para a computação da árvore geradora terem complexidade linear com relação a entrada, torna-se imperioso a busca por algoritmos paralelos eficientes para esse problema.

Um dos desafios para a obtenção de um algoritmo paralelo eficiente é o fato que os algoritmos de busca em largura e busca em profundidade não possuem uma versão paralela equivalente a sequencial, o que dificulta a sua utilização. Os principais algoritmos paralelos para esse problema são baseados na solução proposta por Hirschberg et al. [Hirschberg et al. 1979], onde os vértices do grafo são sucessivamente combinados em super vértices maiores, ou no algoritmo de Borůvka [Borůvka 1926]. Usando essas abordagens, vários algoritmos paralelos para o problema da árvore geradora foram propostos [Chin et al. 1982, Johnson and Metaxas 1995].

No início dos anos 2000, o acesso mais fácil a máquinas paralelas de memória compartilhada (Beowulf´s) possibilitou que os algoritmos paralelos teóricos fossem im-

plementados. Esses algoritmos quando implementados em máquinas paralelas reais não obtiveram bons *ganhos* ou *speed-ups*. Nesse período um esforço considerável foi dispendido na obtenção de algoritmos paralelos eficientes, não só do ponto de vista teórico, mas que também obtivessem bons *speed-ups*. Utilizando modelos de computação realista BSP/CGM [Valiant 1990, Dehne et al. 1996], Dehne et al. propuseram um algoritmo BSP/CGM para computar uma árvore geradora e os componentes conexos de um grafo [Dehne et al. 2002], que utiliza $O(\log p)$ rodadas de comunicação, onde p é o número de processadores. Esse algoritmo, como os baseados nos algoritmos de Hirschberg et al. e no de Borůvka , utilizam o algoritmo de *list ranking* [Dehne and Song 1997] em vários de seus passos. Esse fato acaba por impactar o *speed-up* desses algoritmos.

Baseado na ideia do cálculo de um esteio [Cáceres et al. 1993], Cáceres et al. propuseram uma nova abordagem para o cálculo de uma árvore geradora sem a utilização de *list ranking* [Cáceres et al. 2004]. O algoritmo utiliza $O(\log p)$ rodadas de comunicação, mas tem a vantagem prática de evitar a computação do *list ranking*. O cálculo do esteio é obtido com a utilização de um grafo bipartido auxiliar, onde as arestas do esteio são selecionadas com a utilização de um algoritmo de ordenação inteira, que pode ser implementada de forma eficiente no modelo BSP/CGM [Chan and Dehne 1999]. Um dos limitantes para o *speed-up* desse algoritmo é a necessidade de computar um grafo bipartido do grafo de entrada. O grafo bipartido é obtido com a inserção de um vértice em cada uma das arestas do grafo original.

Com o aumento da capacidade das placas gráficas *General Purpose Graphics Processing Units* (GPGPU´s), surge a oportunidade de analisar o comportamento dos algoritmos BSP/CGM nessa nova arquitetura. Um dos principais limitantes dos *speed-ups* dos algoritmos BSP/CGM, quando implementados em máquinas de memória distribuída é o tempo gasto com a comunicação. Para problemas irregulares como o da computação da árvore geradora de um grafo, são necessárias $O(\log p)$ rodadas de comunicação para a obtenção da árvore geradora do grafo de entrada.

A abordagem proposta por Lima et al. [Lima et al. 2016] estabelece que os superpassos do modelo BSP/CGM sejam representados pelas invocações sequenciais cada *kernel* do CUDA. A execução paralela de cada *kernel* pelas diversas *threads* criadas pelo CUDA representa uma rodada de computação, que pode ser intercalada por comunicação entre as *threads* através da memória da GPU e comunicação entre a GPU e a CPU. Considerando esta abordagem, podemos prever que o comportamento teórico do nosso algoritmo paralelo terá um desempenho compatível quando implementado numa GPGPU. No caso do problema de árvore geradora mínima, existem diversos trabalhos que propõem soluções paralelas usando GPGPU, como os apresentados por [Vineet et al. 2009, Nobari et al. 2012, Li and Becchi 2013, Nasre et al. 2013].

Neste trabalho, utilizando o modelo BSP/CGM, propomos um algoritmo BSP/CGM para a computação da árvore geradora de um dado grafo. O algoritmo proposto é eficiente no modelo BSP/CGM e utiliza $O(\log p)$ rodadas de computação. O algoritmo é baseado no algoritmo proposto por Cáceres et al. [Cáceres et al. 2004] e no algoritmo de Borůvka [Graham and Hell 1985]. Nosso algoritmo não precisa calcular o grafo bipartido auxiliar, e a construção do esteio não necessita de algoritmos de ordenação. Esses dois passos utilizam um tempo considerável na execução total do algoritmo. Para demonstrar que o algoritmo também tem um bom desempenho na prática, a implementação foi

executada com a GPGPU Nvidia Quadro M4000 (que possui 1.664 núcleos e oito GB de memória). Os resultados de *speed-up* obtidos pelo algoritmo são competitivos, demostrando que o modelo BSP/CGM é adequado para o projeto de algoritmos paralelos realistas.

2. Algoritmo paralelo para o problema da árvore geradora

Nosso algoritmo paralelo foi projetado usando o modelo BSP/CGM [Valiant 1990, Dehne et al. 1996]. Resumidamente, esse modelo consiste de um conjunto de p processadores, cada um tendo uma memória local de tamanho $O(n/p)$.

Um algoritmo nesse modelo executa um conjunto de rodadas (superpassos) de computação local alternadas com fases de comunicação global, separadas por uma barreira de sincronização. O custo da comunicação considera o número de rodadas necessárias para a execução do algoritmo.

O modelo BSP/CGM é adequado para o projeto e análise de algoritmos paralelos onde há muita comunicação entre os processos. Essa é uma característica de problemas irregulares, ou seja a entrada em cada rodada do programa muda e os processadores necessitam das informações que foram computadas nos diversos processadores para a próxima rodada. O problema da árvore geradora se enquadra nessa classe, o que motiva a utilização do modelo para prever o comportamento e a complexidade do algoritmo.

Como estamos interessados em analisar não só os aspectos teóricos do nosso algoritmo, temos que mapear os passos do algoritmo BSP/CGM na arquitetura da GPGPU. As rodadas (superpassos) do modelo BSP/CGM são representados pelas chamadas de cada *kernel* do CUDA. Além disso, associamos o conjunto de processadores (p) do modelo BSP/CGM ao conjunto de *streaming multiprocessors* (SM´s) da GPGPU.

Vamos agora descrever os conceitos básicos necessários que serão utilizados em nosso algoritmo. Seja $G = (V, E)$ um **grafo**, onde $V = \{v_1, v_2, \ldots, v_n\}$ é um conjunto de n **vértices** e E é um conjunto de m **arestas** (v_i, v_j), sendo v_i e v_j vértices de V. Um **caminho** em G é uma sequência de arestas $(v_1, v_2), (v_2, v_3), (v_3, v_4) \ldots, (v_{n-1}, v_n)$ conectando vértices distintos $v_1, \ldots, v_n$ de G. Um **ciclo** é um caminho conectando vértices distintos $v_1, v_2, \ldots, v_k$ tal que $v_1 = v_k$. Um grafo é considerado **conexo** se existe um caminho para todo par de vértices v_i, v_j, $1 \leq i \neq j \leq n$ em V. Uma **árvore** $T = (V, E)$ é um grafo conexo sem ciclos. Uma **floresta** é um conjunto de árvores. Uma **árvore geradora** de $G = (V, E)$ é uma árvore $T = (V, E')$ que inclui todos os vértices de G e é um subgrafo de G, ou seja todas as arestas de T pertencem a G, $E' \subset E$. Uma árvore geradora $T = (V, E')$ de $G = (V, E)$ também pode ser definida como o conjunto maximal de arestas de G, $E' \subset E$ e $|E'| = |V| - 1$, que não contém nenhum ciclo.

No algoritmo é utilizado o conceito denominado esteio, mas diferente da definição apresentada em [Cáceres et al. 1993]. No contexto deste artigo, um **esteio**, representado por S, é definido como uma floresta geradora de G, tal que cada vértice $v_i \in V$ é incidente em S com pelo menos uma aresta (v_i, v_j) tal que v_j é o menor vértice ligado a v_i. Uma aresta (v_i, v_j) do esteio é considerada uma **aresta zero-diferença** se ela for escolhida para os dois vértices, tanto para v_i quanto para v_j. A Figura 1 apresenta um grafo G com cinco vértices e oito arestas. O esteio para este grafo é composto por quatro arestas, onde uma delas é zero-diferença. No caso desse exemplo o esteio gerado na primeira iteração do algoritmo é uma árvore geradora de G.

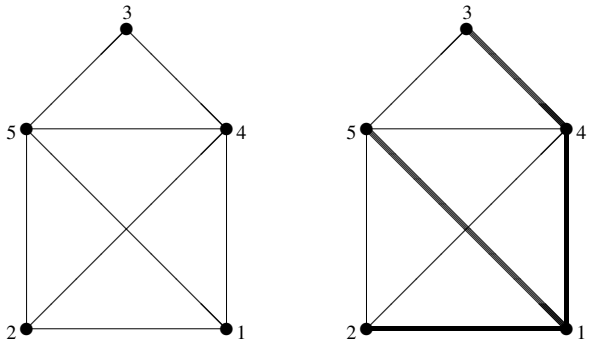


Figura 1. À esquerda é apresentado o grafo $G = (V, E)$, sendo $V = \{1, 2, 3, 4, 5\}$, e à direita é ilustrado o esteio correspondente de G, formado pela aresta $(1, 2)$ (escolhida pelos vértices 1 e 2), aresta $(1, 4)$ (escolhida pelo vértice 4), aresta $(1, 5)$ (escolhida pelo vértice 5) e aresta $(3, 4)$ (escolhida pelo vértice 3), sendo $(1, 2)$ a única aresta zero-diferença.

Algoritmo 1 apresenta a ideia de funcionamento da solução proposta. Este consiste basicamente em encontrar o esteio do grafo e se necessário, caso a árvore geradora não esteja completa, compactá-lo para nova iteracão do algoritmo. A solução consiste de um algoritmo paralelo, ou seja, os passos são executados por diversos processadores encontrando a solução de forma colaborativa.

Algoritmo 1: Algoritmo para árvore geradora

Entrada: Um grafo conexo $G = (V, E)$
Saída: Uma árvore geradora de G cujas arestas estão em $Solucao$.

```
1  início
2      Solucao := vazia
3      repita
4          escolher as arestas do esteio
5          adicionar as arestas do esteio ao conjunto Solucao
6          verificar o número de arestas zero-diferença
7          se número de arestas zero-diferença != 1 então
8              compactar o grafo
9          fim
10     até número de arestas zero-diferença = 1;
11 fim
```

O número de iterações do algoritmo e a árvore geradora produzida depende da rotulação dos vértices. A Figura 2 apresenta o mesmo grafo da Figura 1 mas a rotulação dos vértices é diferente. Neste exemplo, o esteio gerado na primeira iteração do algoritmo, formado pelas arestas $(1, 4)$, $(1, 5)$ e $(2, 3)$, apresenta duas arestas zero-diferença $(1, 4)$ e $(2, 3)$, implicando em outra iteração do algoritmo.

Para o exemplo da Figura 2 é necessário fazer a compactação do grafo G, visto que o número de arestas zero-diferença é diferente de um. Essa compactação inicia com o cálculo das componentes conexas a partir das arestas do esteio e a eliminação das arestas que interligam vértices de uma mesma componente. O grafo resultante da compactação terá dois vértices (1 e 2) e quatro arestas.

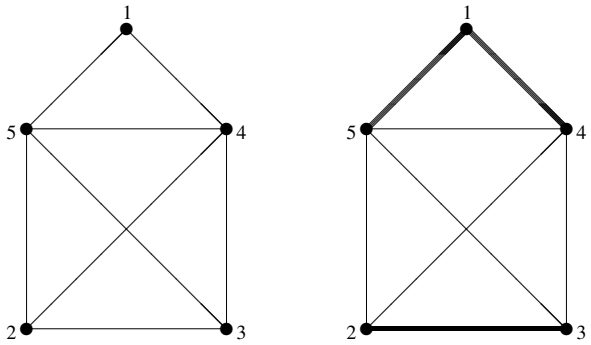


Figura 2. À esquerda é apresentado o grafo $G = (V, E)$, sendo $V = \{1, 2, 3, 4, 5\}$, e à direita é ilustrado o esteio correspondente de G, formado pela aresta $(1, 4)$ (escolhida pelos vértices 1 e 4), aresta $(1, 5)$ (escolhida pelo vértice 5) e aresta $(2, 3)$ (escolhida pelos vértices 2 e 3), sendo $(1, 4)$ e $(2, 3)$ arestas zero-diferença.

3. Correção, Complexidade e Detalhes da Implementação

Nesta seção, abordamos a correção do algoritmo proposto.

Lema 1. *Considere um grafo conexo $G = (V, E)$. Seja G' o grafo obtido pela adição das arestas escolhidas para compor o esteio S (passo 5 do Algoritmo 1). Então G' é acíclico. Mais ainda, se S contém exatamente uma aresta zero-diferença então G' é uma árvore geradora de G.*

Demonstração. Considerando a forma como as arestas são selecionadas para construir o esteio S, onde para cada vértice v_i, é selecionada a aresta com o menor v_j, o conjunto de arestas selecionadas (v_i, v_j), tal que $v_i > v_j$ não formam um ciclo. Se a aresta (v_k, v_l) tal que $v_k < v_l$, ou a aresta (v_l, v_k) foi selecionada ou todas arestas (v_k, v_s) adjacentes a v_k tem que $v_s > v_l$, como $v_k < v_l$, temos que as arestas adjacentes a v_s só se conectarão a v_l usando a aresta v_k, não formando um ciclo. Se apenas uma das arestas for selecionada duas vezes, temos que o esteio S é uma árvore geradora de G. □

Teorema 1. *A cada iteração o número de arestas zero-diferença é no mínimo divido por 2.*

Demonstração. Considerando que cada vértice v_i seleciona uma aresta adjacente (v_i, v_j), onde v_j é o menor vértice adjacente a v_i, o número de arestas selecionadas duas vezes é no máximo $\lceil |V| \rceil / 2$. □

Teorema 2. *O algoritmo para a computação da árvore geradora utiliza $\log p$ rodadas de comunicação com computação $O(n/p)$ computação local.*

Demonstração. Pelo teorema anterior, temos que o grafo compactado após a execução do algoritmo tem no máximo $\lceil |V| \rceil / 2$ vértices, assim, após a $\log p$ rodadas, o grafo compactado terá no máximo 1 aresta zero-diferença e a árvore geradora será obtida. A computação local dos passos de cada rodada pode ser feita em tempo $O(n/p)$. □

Considerando que os trabalhos mais recentes sobre árvores geradoras tais como [Vineet et al. 2009, Nobari et al. 2012, Li and Becchi 2013, Nasre et al. 2013], resolvem problemas mais gerais, além de usarem em sua implementação recursos computacionais muito diferentes, nosso objetivo nesse trabalho foi o de verificar se o algoritmo

proposto também tinha um bom desempenho em máquinas paralelas reais de memória compartilhada, e qual o ganho obtido pelo algoritmo com relação a sua versão sequencial. Para efetuar essa comparação, uma versão sequencial do algoritmo foi desenvolvida usando ANSI C. A versão paralela foi implementada para GPGPU usando CUDA (*Compute Unified Device Architecture*). Ambas as implementações estão disponíveis para download em https://github.com/jucele/ArvoreGeradora.

A implementação CUDA implementa os passos do algoritmo utilizando nove funções do tipo *kernel*. Logo após a leitura dos dados do grafo de entrada estes dados são copiados para a memória global da GPGPU. Duas funções do tipo *kernel* são utilizadas para escolher a aresta do esteio para cada um dos vértices. Após a seleção de uma aresta, utilizamos uma função para marcá-la como uma aresta do esteio. Também usamos uma função para calcular o número de arestas zero-diferença, critério de parada do algoritmo, e copiar as arestas do esteio para o vetor Solução. Outras cinco funções são utilizadas para a compactação do grafo, incluindo o cálculo dos componentes conexos, onde empregamos a proposta apresentada em [Hawick et al. 2010]. Também usamos funções atômicas disponíveis na biblioteca CUDA na implementação de alguns *kernels*.

A implementação do algoritmo não utilizou nenhuma técnica especial de programação em CUDA, pois o objetivo principal do nosso trabalho foi o de apresentar um algoritmo eficiente no modelo BSP/CGM que pudesse ser facilmente implementado numa máquina paralela real.

4. Resultados Experimentais

Como descrevemos anteriormente, o principal objetivo da implementação é o de demonstrar a funcionalidade do algoritmo proposto numa máquina paralela real.

Para isso usamos uma estação de trabalho Intel® Xeon® E5-1620 v3, 3.50GHz, com 8 cores, 10 MB de cache, 32 GB de memória e uma GPGPU Nvidia Quadro M4000 (com 1.664 núcleos e oito GB de memória). Para analisar a eficiência e escalabilidade do algoritmo utilizamos como entrada dois conjuntos de grafos. O primeiro conjunto composto de grafos construídos artificialmente por meio de um gerador de grafos aleatórios. E o segundo composto pelos grafos das redes rodoviárias dos Estados Unidos, disponibilizados no Nono Desafio de Implementação DIMACS.

Para o primeiro conjunto de grafos de entrada utilizamos o gerador aleatório [Johnsonbaugh and Kalin 1991] disponível em http://condor.depaul.edu/rjohnson/source/graph ge.c. Esse gerador possibilitou gerar grafos conexos com um conjunto bem variado de vértices e arestas. Foram gerados 28 grafos conexos com 10.000, 15.000, 20.000, 25.000 e 30.000 vértices. A Tabela 1 apresenta as principais características dos grafos, onde n é o número de vértices e m o número de arestas.

O Nono Desafio de Implementação DIMACS, apresentado no sítio http://www.dis.uniroma1.it/challenge9/, disponibiliza doze grafos de redes rodoviárias dos Estados Unidos. Visto que nossa implementação trabalha com grafos não dirigidos, e como os arquivos dos grafos disponibilizados apresentam as arestas duplicadas (uma para representar o arco entre o vértice a e b e outra para representar a ligação entre b e a), reduzimos o número de arestas dos grafos pela metade. A Tabela 2 mostra as informações desse conjunto de grafos.

Tabela 1. Características básicas dos grafos de entrada gerados.

Grafo de entrada	n	m	densidade	m/n
graph10a	10.000	1.000.000	0,020	100
graph10b	10.000	2.500.000	0,050	250
graph10c	10.000	5.000.000	0,100	500
graph10d	10.000	7.500.000	0,150	750
graph10e	10.000	10.000.000	0,200	1.000
graph15a	15.000	2.500.000	0,022	166,7
graph15b	15.000	5.500.000	0,049	366,7
graph15c	15.000	11.500.000	0,102	766,7
graph15d	15.000	17.000.000	0,151	1.133,3
graph15e	15.000	22.500.000	0,200	1.500
graph15f	15.000	56.300.000	0,500	3.753,3
graph15g	15.000	84.350.000	0,750	5.623,3
graph15h	15.000	112.492.500	1,000	7.499,5
graph20a	20.000	4.000.000	0,020	200
graph20b	20.000	10.000.000	0,050	500
graph20c	20.000	20.000.000	0,100	1.000
graph20d	20.000	30.000.000	0,150	1.500
graph20e	20.000	40.000.000	0,200	2.000
graph25a	25.000	6.200.000	0,020	248
graph25b	25.000	15.500.000	0,050	620
graph25c	25.000	32.000.000	0,100	1.280
graph25d	25.000	47.000.000	0,150	1.880
graph25e	25.000	62.500.000	0,200	2.500
graph30a	30.000	9.000.000	0,020	300
graph30b	30.000	22.500.000	0,050	750
graph30c	30.000	45.000.000	0,100	1.500
graph30d	30.000	67.500.000	0,150	2.250
graph30e	30.000	90.000.000	0,200	3.000

Tabela 2. Características básicas dos grafos do nono desafio DIMACS, considerando os grafos não dirigidos e eliminando as arestas duplicadas.

Grafo de entrada	n	m	densidade	m/n
USA-road-d.NY	264.346	366.648	0,0000105	1,4
USA-road-d.BAY	321.270	399.652	0,0000077	1,2
USA-road-d.COL	435.666	527.767	0,0000056	1,2
USA-road-d.FLA	1.070.376	1.354.681	0,0000024	1,3
USA-road-d.NW	1.207.945	1.417.704	0,0000019	1,2
USA-road-d.NE	1.524.453	1.946.326	0,0000017	1,3
USA-road-d.CAL	1.890.815	2.325.452	0,0000013	1,2
USA-road-d.LKS	2.758.119	3.438.289	0,0000009	1,2
USA-road-d.E	3.598.623	4.382.787	0,0000007	1,2
USA-road-d.W	6.262.104	7.609.574	0,0000004	1,2
USA-road-d.CTR	14.081.816	17.120.937	0,0000002	1,2
USA-road-d.USA	23.947.347	29.120.580	0,0000001	1,2

Para cada um dos grafos de entrada, as implementações sequencial e CUDA foram executadas 20 vezes, sendo usada a média do tempo de execução para analisar o comportamento experimental do algoritmo. A Tabela 3 apresenta os resultados obtidos dos testes para o primeiro conjuntos de grafos de entrada (Tabela 1), onde cada linha

mostra o número de iterações do algoritmo, o tempo de execução da implementação sequencial e o tempo de execução da implementação CUDA e o *speed-up* da implementação em CUDA relativa a implementação sequencial. Vale salientar que o número de iterações do algoritmo necessárias para encontrar a árvore geradora para esse conjunto de testes não passou de dois.

Tabela 3. Resultados dos testes para os grafos gerados artificialmente.

Grafo de entrada	Número de Iterações	Tempo Sequencial (s)	Tempo CUDA (s)	speed-up
graph10a	2	0,031	0,005	6,862
graph10b	2	0,082	0,025	3,286
graph10c	2	0,165	0,018	9,309
graph10d	2	0,259	0,025	10,294
graph10e	2	0,361	0,032	11,389
graph15a	2	0,077	0,009	8,200
graph15b	2	0,170	0,020	8,729
graph15c	2	0,395	0,038	10,500
graph15d	2	0,551	0,052	10,653
graph15e	1	0,278	0,051	5,445
graph15f	1	0,693	0,096	7,183
graph15g	1	1,694	0,132	12,858
graph15h	1	2,858	0,167	17,075
graph20a	2	0,120	0,014	8,598
graph20b	2	0,303	0,034	9,013
graph20c	2	0,668	0,062	10,824
graph20d	2	0,977	0,085	11,490
graph20e	2	1,445	0,106	13,653
graph25a	2	0,186	0,021	8,702
graph25b	2	0,503	0,051	9,870
graph25c	2	1,001	0,094	10,688
graph25d	1	0,584	0,100	5,859
graph25e	2	1,962	0,156	12,613
graph30a	2	0,273	0,031	8,892
graph30b	2	0,695	0,072	9,606
graph30c	2	1,535	0,127	12,053
graph30d	2	2,363	0,173	13,653
graph30e	2	3,852	0,214	18,013

A Tabela 4 mostra os resultados dos testes para o conjunto de grafos do nono desafio DIMACS (Tabela 2). Para essas entradas o *speed-up* da implementação em CUDA em relação à sequencial variou de 3,248 a 19,689. Como o número de vértices desse conjunto de grafos de entrada é bem maior do que o anterior são necessárias de 7 a 10 iterações do algoritmo para que a árvore geradora seja encontrada.

O gráfico da Figura 3 ilustra como o tempo da implementação sequencial é pior do que o tempo da implementação paralela utizando CUDA. A Figura 4 mostra o crescente *speed-up* da implementação CUDA a medida que o tamanho da entrada aumenta em termos de arestas para grafos com 30.000 vértices.

Os resultados mostram a funcionalidade do algoritmo numa máquina paralela real. Os *speed-ups* obtidos também mostram a eficiência do algoritmo, uma vez que os tempos

Tabela 4. Resultados dos testes para os grafos do conjunto nono desafio DIMACS.

Grafo de entrada	Número de Iterações	Tempo Sequencial (s)	Tempo CUDA (s)	speed-up
USA-road-d.NY	7	0,056	0,017	3,248
USA-road-d.BAY	7	0,064	0,018	3,464
USA-road-d.COL	8	0,088	0,020	4,425
USA-road-d.FLA	9	0,263	0,052	5,083
USA-road-d.NW	8	0,283	0,051	5,590
USA-road-d.NE	9	0,475	0,060	7,903
USA-road-d.CAL	9	0,560	0,061	9,245
USA-road-d.LKS	9	0,880	0,084	10,426
USA-road-d.E	10	1,356	0,127	10,665
USA-road-d.W	10	2,299	0,214	10,746
USA-road-d.CTR	10	13,277	0,674	19,689
USA-road-d.USA	10	9,302	0,776	11,985

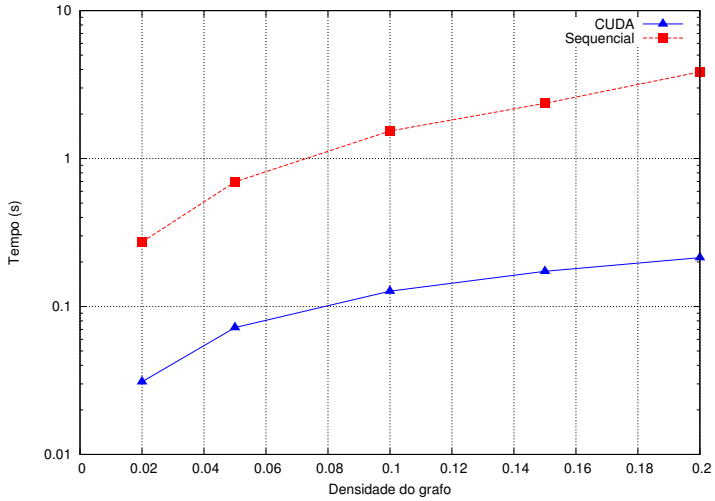


Figura 3. Tempo de Execução para grafos com 30.000 vértices.

das execuções paralelas são bem melhores que das sequenciais. Relembramos que o problema tratado tem uma solução sequencial ótima que é linear para o tamanho da entrada, e que essa a solução não possui uma implementação paralela direta.

Uma das dificuldades da implementação desse algoritmo numa arquitetura de memória distribuída é a quantidade de mensagens trocadas entre os processadores durante a fase de computação, pois em várias computações, há a necessidade de percorrer a lista das arestas e essa lista está distribuída entre os diversos processadores. Na arquitetura de memória compartilhada esse problema é minimizado, além do que existem melhorias implementadas com relação ao algoritmo anterior [Cáceres et al. 2004], não sendo mais necessárias a construção do grafo bipartido auxiliar nem a ordenação das arestas.

Como destacamos anteriormente, a comparação dos tempos de execução com algoritmos para árvore geradora utilizando CUDA que foram publicados recentemente não foi possível, pois os algoritmos são para problemas mais específicos e utilizam recursos computacionais bem diferentes.

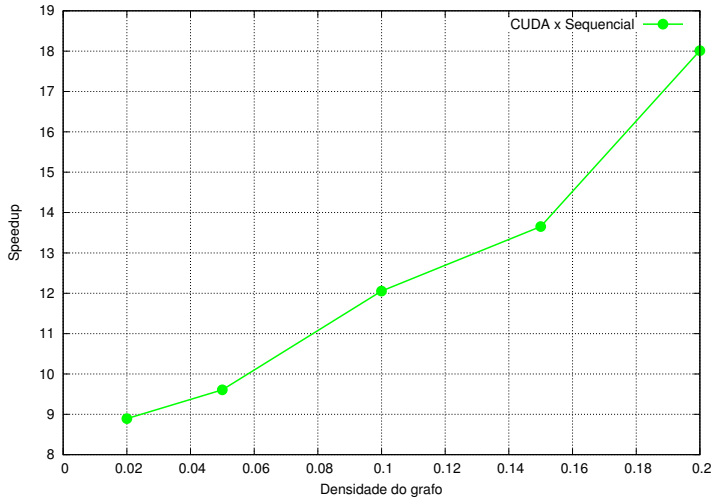


Figura 4. Speedups obtidos pela implementação CUDA em relação a implementação sequencial para grafos com 30.000 vértices.

5. Conclusões e trabalhos futuros

Neste trabalho, utilizando o modelo BSP/CGM, apresentamos um algoritmo paralelo para o cálculo de árvore geradora. O modelo BSP/CGM tem se mostrado bem adequado para o projeto de algoritmo paralelos, principalmente os que utilizam muita comunicação entre os processadores. Além disso, os algoritmos projetados nesse modelo têm obtido bons *speed-ups* quando implementado em máquinas reais.

Considerando que os principais algoritmos sequenciais para o problema da árvore geradora utilizam estratégias de busca em profundidade ou busca em largura em um grafo, e que essas estratégias não possuem uma paralelização eficiente, os algoritmos paralelos propostos para esse problema usam outras abordagens. Várias das soluções fazem uso do algoritmo de *list ranking*, o que prejudica o desempenho final dos algoritmos.

O algoritmo proposto é baseado no trabalho de Cáceres et al. [Cáceres et al. 2004], que não utiliza *list ranking* e utiliza um grafo bipartido auxiliar e ordenações do conjunto de arestas. O algoritmo proposto neste trabalho não faz uso de um grafo bipartido, nem necessita ordenar o conjunto de arestas, e utiliza $\log p$ rodadas de comunicação com $O(n/p)$ computação local em cada unidade de processamento.

Para demonstrar a eficiência do algoritmo proposto, uma implementação CUDA foi desenvolvida e testada em uma GPGPU. Utilizamos um gerador de grafos para analisar a escalabilidade da implementação. Também realizamos testes utilizando como entrada os grafos disponibilizados pelo novo desafio DIMACS. Os tempos e *speed-ups* obtidos são competitivos, e mostram que o modelo BSP/CGM é apropriado para o projeto e desenvolvimento de algoritmos paralelos para máquinas paralelas reais.

Como trabalhos futuros, pretendemos desenvolver uma abordagem de poda para reduzir o conjunto de arestas do grafo de entrada, que pode melhorar os resultados do algoritmo e permitir trabalhar com grafos de entrada maiores. Pretendemos avaliar também o algoritmo com outros conjuntos de dados e analisar o seu desempenho em equipamentos computacionais com maior poder de processamento, tais como múltiplas GPUs.

Agradecimentos

Esta pesquisa foi parcialmente financiada pelo CNPq Proc. No. 482736/2012-7, 30.2620/2014-1 e 465446/2014-0, e FAPESP Proc.2014/50937-1.

Referências

Borůvka, O. (1926). On a minimal problem. *Prace Moravské Pridovedecké Spodecnosti*, 3:37–58.

Cáceres, E. N., Dehne, F., Mongelli, H., Song, S. W., and Szwarcfiter, J. (2004). A coarse-grained parallel algorithm for spanning tree and connected components. In *Euro-Par 2004. Lecture Notes in Computer Science*, volume 3149, p. 828–831. Springer-Verlag.

Cáceres, E. N., Deo, N., Sastry, S., and Szwarcfiter, J. L. (1993). On finding Euler tours in parallel. *Parallel Processing Letters*, 3(3):223–231.

Chan, A. and Dehne, F. (1999). A note on coarse grained parallel integer sorting. *Parallel Processing Letters*, 9(4):533–538.

Chin, F. Y., Lam, J., and Chen, I.-N. (1982). Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659–665.

Dehne, F., Fabri, A., and Rau-Chaplin, A. (1996). Scalable Parallel Computational Geometry for Coarse Grained Multicomputers. *International Journal on Computational Geometry & Applications*, 6(3):298–307.

Dehne, F., Ferreira, A., Cáceres, E., Song, S. W., and Roncato, A. (2002). Efficient parallel graph algorithms for coarse grained multicomputers and BSP. *Algorithmica*, 33(2):183–200.

Dehne, F. and Song, S. W. (1997). Randomized parallel list ranking for distributed memory multiprocessors. *International Journal of Parallel Programming*, 25(1):1–16.

Graham, R. L. and Hell, P. (1985). On the history of of the minimum spanning tree problem. In *Annals of the History of Computing*, volume 7, p. 43–57.

Hawick, K. A., Leist, A., and Playne, D. (2010). Parallel graph component labelling with GPUs and CUDA. *Parallel Computing*, 36(12):655–678.

Hirschberg, D. S., Chandra, A. K., and Sarwate, D. V. (1979). Computing connected components on parallel computers. *Comm. ACM*, 22(8):461–464.

Johnson, D. and Metaxas, P. (1995). A parallel algorithm for computing minimum spanning trees. *Journal of Algorithms*, 19(3):383 – 401.

Johnsonbaugh, R. and Kalin, M. (1991). A graph generation software package. In *Proceedings of the twenty-second SIGCSE technical symposium on Computer Science Education*, volume 23, p. 151–154.

Karger, D. R., Klein, P. N., and Tarjan, R. E. (1995). A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328.

Li, D. and Becchi, M. (2013). Deploying graph algorithms on gpus: an adaptive solution. In *Proceedings of IEEE 27th International Parallel and Distributed Processing Symposium*, IPDPS 2013, p. 1013–1024.

Lima, A. C. d., Branco, R. G., Ferraz, S., Cáceres, E. N., Gaioso, R. R. A., Martins, W. S., and Song, S. W. (2016). Solving the maximum subsequence sum and related problems using BSP/CGM model and multi-GPU CUDA. *Journal of The Brazilian Computer Society (Online)*, 22:1–13.

Nasre, R., Burtscher, M., and Pingali, K. (2013). Morph algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, p. 147–156.

Nobari, S., Cao, T.-T., Karras, P., and Bressan, S. (2012). Scalable parallel minimum spanning forest computation. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, p. 205–214.

Valiant, L. G. (1990). A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111.

Vineet, V., Harish, P., Patidar, S., and Narayanan, P. J. (2009). Fast minimum spanning tree for large graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, p. 167–171.

Vetorização e Análise de Algoritmos Paralelos para a Migração Kirchhoff Pré-empilhamento em Tempo

Rodrigo Alves Prado da Silva[1], Maicon Melo Alves[1], Cristiana Barbosa Bentes[2] e Lúcia Maria de Assumpção Drummond[1]

[1]Universidade Federal Fluminense (UFF), Niterói, RJ, Brasil
[2]Universidade do Estado do Rio de Janeiro (UERJ), Maracanã, RJ, Brasil

{rprado, mmelo, lucia}@ic.uff.br, cris@eng.uerj.br

Resumo. *A Migração Kirchhoff Pré-empilhamento em Tempo (ou PKTM, do inglês Pre-stack Kirchhoff Time Migration) é parte central do processo de exploração de petróleo. Como o PKTM é computacionalmente intensivo, muitos trabalhos propuseram o uso de aceleradores como GPU (Graphical Processing Units) para reduzir o seu tempo de execução. Embora os processadores modernos possuam um recurso de aceleração vetorial, apenas um trabalho avaliou o uso deste recurso para acelerar o PKTM. Contudo, este trabalho avaliou apenas a versão sequencial desta aplicação. Nesse trabalho, propõe-se uma analise da vetorização de duas versões paralelas do PKTM. Para a primeira versão paralela, foi utilizado OpenMP e para a segunda, foi utilizado MPI. Em relação à vetorização, foram consideradas a vetorização automática, executada pelo compilador, e a vetorização manual, implementada pelo programador. Uma análise experimental mostrou que a vetorização automática no código com o OpenMP produziu melhores resultados do que os obtidos no código sequencial e no código com MPI. Assim, foram propostas algumas otimizações que permitiram que as versões sequenciais e com MPI obtivessem um desempenho similar ao alcançado no código com OpenMP.*

1. Introdução

O primeiro passo na exploração de petróleo e gás consiste em realizar um processamento nos dados sísmicos que foram adquiridos em uma determinada área de interesse. Esse processo fornece imagens da subsuperfície a fim de permitir a interpretação das estruturas geológicas presentes nessa área. A partir do conhecimento dessas estruturas, é possível identificar áreas onde o petróleo possa ser encontrado e extraído. Dentre as etapas executadas neste processamento sísmico, a migração sísmica é considerada como sendo o passo central de todo este processo. A migração sísmica produz uma imagem mais fiel das estruturas geológicas encontradas em subsuperfície ao colapsar as difrações hiperbólicas e mover as estruturas mergulhantes para suas reais posições.

Um dos métodos mais populares de migração sísmica é a Migração Kirchhoff Pré-empilhamento em Tempo (PKTM, do Inglês *Pre-stack Kirchhoff Time Migration*) que baseia-se no procedimento de soma de difrações. O PKTM é amplamente utilizado por sua simplicidade, eficiência, confiabilidade e flexibilidade de I/O [Xu et al. 2014]. No entanto, esta migração é computacionalmente intensiva, ou seja, mesmo em supercomputadores, sua execução pode levar semanas ou até meses para ser concluída [Shi et al. 2011].

Devido às inerentes características que permitem o paralelismo de dados do PKTM [Rizvandi et al. 2011], muitos trabalhos como [Xu et al. 2014], [Shi et al. 2011], [Panetta et al. 2012] e [Sun and Shi 2012] propuseram soluções para reduzir o tempo de execução deste algoritmo ao adotar dispositivos aceleradores como as *Graphical Processing Units* (GPU) ou *Field-programable Gate Array* (FPGA). Outros trabalhos como [Dai 2005] propuseram a divisão e distribuição do processamento do PKTM em *cluster* computacional usando o padrão MPI (*Message Passing Interface*). Já outros trabalhos como [Yang et al. 2011] propuseram uma solução híbrida que faz uso tanto do paralelismo de memória distribuída (MPI) quanto do paralelismo de memória compartilhada (OpenMP), além de utilizar também os recursos de aceleração da GPU.

Contudo, nenhum destes trabalhos considerou um importante recurso de aceleração disponível na maioria dos processadores modernos. Este recurso, chamado de unidade de processamento vetorial, permite explorar o paralelismo de dados com granularidade fina ao realizar operações simultâneas em diferentes elementos de um mesmo vetor [Lomont 2011]. A segunda geração do processador Xeon Phi da Intel, Knights Landing, por exemplo, possui duas dessas unidades de vetorização em cada um dos seus *cores*, onde cada uma dessas unidades de vetorização permite realizar operações simultâneas em um vetor de 512 bits. Em outras palavras, pode-se executar, ao mesmo tempo, 16 operações de ponto-flutuante de precisão simples ou 8 operações de ponto-flutuante de precisão dupla.

Esse recurso de vetorização pode ser explorado por meio de um processo de vetorização automática provido por compiladores como GCC (*GNU Compiler Collection*) ou ICC (*Intel C/C++ Compiler*). Quando a opção de vetorização automática é ativada, o compilador procura por estruturas de código que possam ser vetorizadas, ou seja, por laços internos que não apresentem, por exemplo, controle de fluxo ou dependência de dados entre os elementos dos vetores que estão sendo processados. Portanto, a vetorização automática é transparente e não requer qualquer esforço por parte do programador [Mitra et al. 2013] [Schuchart et al. 2015] [Intel 2012]. Entretanto, por conta de decisões mais conservadoras tomadas pelo compilador, o código automaticamente vetorizado pode não extrair todo o potencial fornecido pela unidade de vetorização. Nestes casos, existe a possibilidade de realizar uma vetorização manual do código-fonte na qual o próprio programador utiliza funções de alto nível para acessar diretamente as instruções vetoriais disponíveis no processador [Hofmann et al. 2014].

Embora o trabalho descrito em [Melo Alves et al. 2016] tenha analisado o custo-benefício de adotar diferentes abordagens de vetorização para acelerar o PKTM sequencial, não há conhecimento a respeito de outro trabalho da literatura que tenha analisado o efeito da vetorização sobre códigos paralelos do PKTM. Estamos interessados em investigar como o uso de bibliotecas de paralelização, como OpenMP e MPI podem afetar a vetorização do código. Assim, neste trabalho, propõe-se avaliar o efeito da vetorização manual e automática sobre duas novas versões do PKTM, uma paralelizada com OpenMP e outra com MPI. Adicionalmente, o trabalho também apresenta algumas técnicas de otimização que foram aplicadas na versão sequencial e paralelizada com MPI com a finalidade de aumentar a quantidade de laços automaticamente vetorizados nessas versões do PKTM.

Uma análise experimental, considerando uma seção sísmica sintética e o com-

pilador ICC, revelou que o ganho de desempenho alcançado ao se combinar as duas abordagens, vetorização e paralelização, foi sempre superior ao desempenho obtido ao se aplicar isoladamente uma outra ou outra abordagem. Ao ser executado com 6 *threads*, a versão manualmente vetorizada e paralelizada com o OpenMP alcançou um speedup de 9,4 quando comparado com uma versão sequencial e escalar do PKTM. Além disso, os resultados também indicaram que, a exemplo da vetorização do código sequencial, a vetorização manual foi capaz de obter melhores resultados quando comparada com a versão automaticamente vetorizada.

2. Migração Kirchhoff com Pré-empilhamento em Tempo

O processamento de dados sísmicos consiste em produzir uma imagem de subsuperfície que reflita, o mais precisamente possível, as características físicas e a estrutura geológica de uma dada área de interesse. Essa etapa é executada a partir de dados brutos coletados durante o processo de aquisição sísmica.

Em uma aquisição de dados sísmicos, um emissor de ondas e um conjunto de receptores são posicionados em uma determinada área de interesse. A distância entre o emissor e cada um dos receptores é chamado de afastamento (ou *offset*, em Inglês). Periodicamente, o emissor propaga uma onda através da subsuperfície da Terra que, ao atingir uma camada subsuperficial de rocha, é refratada e refletida em direção à superfície. Esse processo, conhecido como *aquisição de tiro comum*, é executado várias vezes durante o processo de aquisição sísmica. Em cada tiro, o emissor e os receptores são alocados em diferentes posições da área de interesse a fim de que possam ser coletadas informações redundantes sobre um mesmo ponto da subsuperfície [Panetta et al. 2012][Yilmaz and Doherty 1987] [Xu et al. 2014].

Durante períodos discretos de tempo, os receptores coletam a energia refletida pelas camadas de rocha presentes em subsuperfície. Os dados coletados por um receptor durante um período discreto de tempo t são conhecidos como *amostra* e representam a amplitude da energia refletida por um ponto em subsuperfície. Considerando um refletor plano e paralelo, esse ponto, conhecido como *Ponto Médio Comum* (ou apenas CMP do Inglês *Commom Mid Point*), está localizado na posição central entre o emissor e o receptor. Um conjunto de amostras coletadas por um receptor durante um tiro é chamado de *traço sísmico* e representa a energia refletida por um CMP durante todo o tempo de propagação da onda. O conjunto de traços sísmicos de uma determinada área denomina-se *seção sísmica* e é geralmente gravado em um formato de arquivo padrão [Yilmaz and Doherty 1987].

Após a execução das etapas de pré-processamento, a seção sísmica está pronta para ser migrada [Yilmaz and Doherty 1987]. Basicamente, a migração: (i) colapsa as difrações hiperbólicas, (ii) move as estruturas mergulhantes para suas reais posições em subsuperfície e (iii) aumenta a resolução espacial. Uma seção sísmica precisa ser migrada, porque um ponto difrator, ao receber a excitação de uma fonte de energia, produz uma onda semicircular (de acordo com a Lei de Huygens) que acaba sendo coletada pelos receptores como uma hipérbole. Assim, a imagem resultante pode apresentar estruturas geológicas que não estejam em suas posições reais ou que nem sequer existam, de fato, na área de interesse. Desse modo, a migração corrige essas distorções e gera uma imagem mais precisa a respeito das características físicas e das estruturas geológicas presentes no

subsolo [Yilmaz and Doherty 1987] [Claerbout 1985].

Mais especificamente, o PKTM recebe como entrada uma seção pré-empilhada de afastamento comum a fim de produzir uma imagem final em coordenadas de tempo. O pressuposto básico do PKTM é de que qualquer ponto na seção sísmica pode ser considerado como sendo um refletor mergulhante. Neste sentido, o método assume que cada um desses pontos estaria localizado no ápice de uma hipérbole. Sendo assim, cada ponto deve receber de volta a energia que foi dispersada durante o processo de aquisição de dados, ou seja, o ápice deve receber a contribuição de energia de todas as amostras que compõe essa hipérbole [Teixeira et al. 2013]. O comportamento desta hipérbole pode ser definido por uma equação de tempo de trânsito duplo que determina os pontos (amostras de entrada) que devem contribuir para o seu ápice (amostras de saída) [Yilmaz and Doherty 1987].

Algorithm 1 PKTM Sequencial e Escalar

```
    /* Laço de Offset */
 1: Para Todos Offsets Faça                                          ▷ Não Vetorizado
        /* Laço de Traços de Entrada */
 2:     Para Todos Traços de Entrada Faça                            ▷ Não Vetorizado
            /* Laço de Cópia de Amostras de Traço de Entrada */
 3:         Para Todos Amostras de Entrada Faça Copia_Amostra()      ▷ Vetorização Auto e Manual
            /* Laço de Filtragem */
 4:         Para Todos Frequências de Corte Faça                     ▷ Não Vetorizado
 5:             Aplica_Filtro_Anti-alias()                           ▷ Vetorização Auto e Manual
                /* Laço de Cópia de Amostras Filtradas */
 6:             Para Todos Amostras Filtradas Faça Copia_Amostra()   ▷ Vetorização Auto e Manual
 7:         Fim Para
            /* Laço de Migração */
 8:         Para Todos Amostras de Saída Faça                        ▷ Não Vetorizado
 9:             Le_Velocidade()
10:             Determina_Traços_Dentro_da_Abertura()
                /* Laço de Contribuição */
11:             Para Todos Traços de Saída na Abertura Faça          ▷ Vetorização Manual (Parcial)
12:                 Calcula_Tempo_de_Transito()
13:                 Seleciona_Amostras_para_Interpolação()
14:                 Calcula_Operador_de_Migração()
15:                 Define_Filtros()
                    /* Laço de Cópia de Amostras Selecionadas */
16:                 Para Todos Amostras Selecionadas Faça Copia_Amostra()   ▷ Vetorização Manual
17:                 Interpola_Amostras_Selecionadas()
18:                 Calcula_Fator_de_Obliquidade()
19:                 Calcula_Angulo_de_Abertura()
20:                 Calcula_Fator_de_Espalhamento_Geometrico()
21:                 Corrige_Amplitude()
22:                 Acumula_Contribuição()
23:             Fim Para
24:         Fim Para
25:     Fim Para
26: Fim Para
```

Uma versão escalar do PKTM é apresentada no Algoritmo 1. Para cada *offset*, o PKTM executa o *Laço de Traços de Entrada* (linhas 2 a 25) onde cada traço de entrada do *offset* atual é processado. Em seguida, o PKTM executa o *Laço de Filtragem* (linhas 4 a 7) a fim de obter diferentes versões filtradas do mesmo traço de entrada. Para isso, o

algoritmo aplica, de acordo com a frequência de corte atual, um filtro *anti-alias* em cada uma das amostras do traço de entrada (linha 5). Em seguida, algoritmo armazena todas as amostras filtradas em um vetor auxiliar (linha 6).

Após o processo de filtragem do traço de entrada, o algoritmo executa o *Laço de Migração* (linhas 8 a 24). Para cada amostra de saída, o PKTM lê a velocidade do CMP corrente (linha 9), além de determinar os traços que compõem a abertura (linha 10). A abertura define os traços de saída que receberão as contribuições de um dado traço de entrada. Em seguida, o PKTM executa o *Laço de Contribuição* (linhas 11 a 23). No primeiro passo desse laço, o tempo de trânsito duplo é calculado (linha 12) para identificar qual amostra de entrada deverá contribuir para a amostra de saída atual. Entretanto, como o tempo de trânsito calculado anteriormente se encontra em um domínio contínuo, o PKTM seleciona um conjunto de amostras do traço de entrada (linha 13) a fim de transpor, posteriormente, o valor calculado para um domínio discreto por meio de um processo de interpolação. Depois disso, o retardo horizontal do operador de migração é calculado (linha 14) e o seu valor é usado como parâmetro de entrada para determinar os filtros apropriados para o processo de *anti-alias* (linha 15).

Em seguida, o algoritmo calcula a amplitude da energia através de uma interpolação entre os filtros e as amostras selecionadas anteriormente (linha 17). Esse processo de interpolação visa prover um valor aproximado de amplitude de acordo com os filtros escolhidos e o conjunto de amostras previamente determinado. Então, o PKTM calcula: (i) o fator de obliquidade, (ii) o ângulo de abertura, e (iii) o fator de espalhamento geométrico (linhas 18, 19, e 20, respectivamente) a fim de realizar a correção da amplitude no passo seguinte (linha 21). Por fim, essa amplitude corrigida é acumulada na amostra de saída atual (linha 22). Uma descrição mais detalhada do PKTM pode ser encontrada em [Yilmaz and Doherty 1987].

Conforme indicado no Algoritmo 1, o *Laço de Cópia de Amostras de Traço de Entrada* (linha 3), o *Laço de Cópia de Amostras Filtradas* (linha 6) e todos os laços dentro do procedimento *Aplica_Filtro_Anti-alias* (linha 5) foram automática e manualmente vetorizados. Já o *Laço de Cópia de Amostras Selecionadas* (linha 16) e o *Laço de Contribuição* (linha 11) foram vetorizados apenas manualmente. Mais precisamente, o *Laço de Contribuição* foi parcialmente vetorizado, já que algumas etapas executadas neste laço não puderam ser vetorizadas por conta de restrições como controle de fluxo e dependência de dados. Assim, esse laço foi dividido em dois novos laços, um para executar as tarefas vetorizadas e outro para executar as tarefas escalares. Maiores detalhes sobre a vetorização manual e automática do PKTM sequencial estão descritos em [Melo Alves et al. 2016].

3. Versões Paralelas do PKTM

Com o intuito de comparar o efeito da vetorização sobre versões paralelas do PKTM, foram desenvolvidas duas novas versões paralelizadas do PKTM. A primeira utilizando OpenMP e a segunda utilizando MPI, como descrito a seguir.

3.1. PKTM Paralelo com OpenMP

Uma versão paralela *multithreading* do PKTM, implementada com OpenMP, é apresentada no Algoritmo 2. Foi verificado que o *Laço de Traços de Entrada* desse algoritmo

é responsável pela maior parte do tempo de processamento desta migração sísmica. De forma a reduzir esse tempo de execução, foi utilizada a diretiva de compilação *omp parallel for* para que o trabalho pudesse ser distribuído entre um dado número de *threads* (linha 2). Cada uma dessas *threads* processa uma mesma quantidade de traços de entrada, ou seja, todas as *threads* recebem uma carga de trabalho similar. A execução dos passos internos do *Laço de Traços de Entrada* segue conforme definido pelo Algoritmo 1.

Algorithm 2 PKTM Paralelo com OpenMP

```
   /* Laço de Offset */
1: Para Todos Offsets Faça
       /* Laço de Traços de Entrada */
       #pragma omp parallel for
2:     Para Todos Traços de Entrada Faça
           /* Laço de Filtragem */ [1]
3:     Fim Para
4: Fim Para
```

[1] O corpo do Laço de Filtragem foi omitido para simplificar o entendimento.

3.2. PKTM Paralelo com MPI

No caso do MPI, foi feita uma implementação considerando uma granularidade mais grossa do que a utilizada pela versão paralelizada com o OpenMP. Assim, optou-se pela distribuição do trabalho realizado pelo laço mais externo, ou seja, pelo *Laço dos Traços de Entrada*. Dessa forma, a quantidade de mensagens trocadas entre os processos é menor do que seria necessário para paralelizar a execução de cada um dos *offsets* da seção sísmica. Como a aplicação possui operações de entrada (leitura dos *offsets*) e saída (escrita dos traços migrados), optou-se por concentrar todas essas operações em um processo mestre, enquanto os demais processos, chamados trabalhadores, executam efetivamente o processo de migração dos traços de entrada.

Os Algoritmos 3 e 4 apresentam os pseudocódigos implementados em MPI dos processos mestre e trabalhador, respectivamente. No Algoritmo 4, o processo trabalhador envia uma mensagem de solicitação de *offset* ao processo mestre (linha 2) e, ao receber a resposta (linha 3), realiza o processamento do respectivo *offset* (linha 6). Este procedimento é repetido até que o processo mestre termine a leitura de todos os *offsets* da seção sísmica. Quando isto ocorre, o processo mestre responde ao processo trabalhador com uma mensagem de término (linha 7 do Algoritmo 3).

Repare que essa implementação favorece o balanceamento de carga entre os processos trabalhadores, já que um processo trabalhador, ao encerrar seu processamento, pode requisitar mais um *offset* ao processo mestre sem a necessidade de esperar que os demais processos trabalhadores terminem o seu processamento.

3.3. Análise das Vetorizações dos Códigos do PKTM Paralelo

Com o intuito de comparar o efeito da vetorização sobre as versões paralelas do PKTM, foram considerados os mesmos tipos de vetorização realizados anteriormente no código sequencial: (i) a versão automaticamente vetorizada pelo compilador e (ii) a versão manualmente vetorizada pelo programador. Portanto, essas duas abordagens de vetorização

Algorithm 3 PKTM Paralelo com MPI - Processo Mestre

1: **Enquanto** $\exists\ Fim\_processo[i] == FALSE,\ para\ i \in \{1 \ldots n\}$ **Faça**
2: **Recebe** Mensagem *Pedido* de processo i
3: **Lê** *Offset*
4: **Se** $not\ Fim\ do\ Arquivo\ de$ *Offsets* **Então**
5: **Envia** Mensagem *Offset* para processo i
6: **Senão**
7: **Envia** Mensagem *Fim* para processo i
8: $Fim\_processo[i] = TRUE$
9: **Fim Se**
10: **Fim Enquanto**

Algorithm 4 PKTM Paralelo com MPI - Processo Trabalhador

1: **Enquanto** $not\ Terminou$ **Faça**
2: **Envia** Mensagem *Pedido* Para Processo Mestre
3: **Recebe** Mensagem *msg* do Processo Mestre
4: **Se** *msg* $== \{$*Fim*$\}$ **Então**
5: $Terminou = TRUE$
6: **Senão**
/ Laço de Offset */* [1]
7: **Fim Se**
8: **Fim Enquanto**

[1] O corpo do Laço de Offset foi omitido para simplificar o entendimento.

foram aplicadas sobre as duas versões paralelas do PKTM apresentadas nas Seções 3.1 e 3.2.

Surpreendentemente, ao utilizar o OpenMP para efetuar a paralelização do código do PKTM, o compilador foi capaz de aumentar o número de laços vetorizados. Com isso, além de vetorizar automaticamente o *Laço de Cópia de Amostras de Traço de Entrada*, o *Laço de Cópia de Amostras Filtradas* e todos os laços dentro do procedimento chamado *Aplica_Filtro_Anti-alias*, o compilador foi capaz vetorizar automaticamente também o *Laço de Cópia de Amostras Selecionadas*. Este laço não foi vetorizado automaticamente porque o compilador acusou dependência de dados. O fato é que o OpenMP cria novas variáveis para o contexto de cada *thread* dentro do laço. Por conta desta criação de novas variáveis, o compilador provavelmente descarta a possibilidade de dois ponteiros apontarem para a mesma área de memória e consegue vetorizar o laço.

Quanto à vetorização manual, não houve diferença, uma vez que a implementação manual da vetorização é a mesma empregada na versão sequencial.

Ao utilizar o MPI para efetuar a paralelização do código do PKTM, o compilador teve a capacidade de vetorização reduzida uma vez que foi capaz de vetorizar somente o *Laço de Cópia de Amostras de Traço de Entrada* e o *Laço de Cópia de Amostras Filtradas*. O compilador acusou o impedimento à vetorização por dependência de dados nos laços que estão dentro do procedimento *Aplica_Filtro_Antialias()*. Quanto à vetorização manual, não houve mudanças, uma vez que a implementação manual da vetorização é a mesma empregada na versão sequencial.

A vetorização manual seguiu os mesmos passos descritos em [Melo Alves et al. 2016], enquanto que a vetorização automática foi realizada pelo ICC. Com a vetorização manual, foi possível vetorizar, em ambas as versões paralelas, os mesmos laços que foram vetorizados manualmente na versão sequencial. Todas as versões manualmente vetorizadas (Sequencial, MPI e OpenMP) apresentam a mesma quantidade de laços vetorizados.

3.3.1. Otimizações Para Vetorização Automática dos Códigos Sequencial e com MPI

Para fazer com que as versões sequencial e MPI vetorizassem automaticamente os mesmos laços que a versão OpenMP, foram utilizadas duas técnicas de otimização nessas versões.

A primeira delas visou resolver o problema de dependência de dados relacionado ao uso de ponteiros. Por ser conservador em suas decisões de vetorização, o compilador, ao analisar expressões que envolvam ponteiros, pode erroneamente supor que os endereços de memória apontados por esse tipo de variável possam se sobrepor, o que implicaria, em muitos casos, em dependência de dados e na não vetorização do laço de execução.

Para contornar esse problema, algumas variáveis do tipo ponteiro, localizadas dentro do *Laço de Offset* das versões sequencial e MPI, foram declaradas com a palavra-chave *restrict*. Essa palavra-chave, que foi originalmente introduzida na especificação ISO C99, foi usada para informar ao compilador que uma determinada variável do tipo ponteiro seria a única a apontar para um determinado endereço de memória. Desta forma, o compilador foi capaz de vetorizar automaticamente nas versões sequencial e com MPI os mesmos laços que foram vetorizados na versão OpenMP, já que não havia mais o risco de haver dependência de dados nestes laços de execução.

Além do uso da palavra-chave *restrict*, foi necessário aplicar mais uma técnica de otimização na versão MPI. Nesta versão, o compilador não foi capaz de vetorizar automaticamente laços presentes no processo de filtragem dos traços de entrada (procedimento *Aplica_Filtro_Anti-alias()*). Para vetorizar automaticamente esses laços, foi preciso impedir que o compilador efetuasse o *inlining* do procedimento de filtragem para dentro do código da migração. Dessa forma, as chamadas de procedimentos foram efetuadas para a execução desse procedimento, ao invés da realização da cópia dos mesmos. Em consequência disso, foi acrescentado um *overhead* da chamada de procedimento de filtragem na execução da migração do *offset*. Mas esse *overhead* foi pequeno se comparado a não vetorização automática da versão MPI do PKTM.

4. Resultados Experimentais

Os testes foram realizados em uma máquina com processador Intel i7-5930K 3.50 GHz, 32 GB de memória RAM e sistema operacional Ubuntu 14.04. Este processador possui seis núcleos de processamento e é equipado com o conjunto de instruções vetoriais *AVX2* (*Advanced Vector Extensions 2*). Esse conjunto de instruções permite realizar operações simultâneas em um vetor de 256 bits. O compilador utilizado foi o ICC versão 17.0.2 e a versão de API OpenMP utilizada foi a 4.5. Para o MPI foi utilizada a versão 1.6.5 do OpenMPI.

Todas as versões do PKTM foram compiladas com a opção *O3* do ICC a qual permite que o compilador efetue, além de diversas otimizações, a vetorização automática do código-fonte. Para as versões escalar e manualmente vetorizadas, a opção *no-vec* foi utilizada para que o processo de vetorização automática não fosse executado pelo compilador.

Todas as versões do PKTM implementadas e analisadas neste trabalho foram baseadas no *Seismic Unix* versão 44R4, que é um pacote de ferramentas *Open Source* amplamente utilizado pela indústria e academia para realizar processamento sísmico. Como dado de entrada, foi utilizada uma seção sísmica sintética com as seguintes características: 63 *offsets*, 128 traços por *offset*, 512 amostras por traço, intervalo para cada *offset* de 20 metros e intervalo de obtenção das amostras de 0,004 segundos. Ao todo, essa seção sísmica é composta por 8064 traços de entrada. Os arquivos fontes e o conjunto de dados de entrada utilizados podem ser obtidos no endereço `https://github.com/rodrigo-prado/kirchhoff`.

Para os testes com OpenMP foram utilizadas 2, 4 e 6 *threads* e para os testes usando o MPI foram utilizados 1 processo mestre e 2, 4 e 6 processos trabalhadores.

A Tabela 1 apresenta (i) o número de *threads* e processos trabalhadores das versões do PKTM, (ii) o tempo médio de cinco execuções, (iii) o *speedup* da vetorização e (iv) o *speedup* geral, que é o combinado da paralelização com a vetorização. O *speedup* da vetorização compara, para cada versão sequencial ou paralela, o tempo de execução das versões vetorizadas em relação a versão escalar. O *speedup* geral compara cada versão com a versão sequencial e escalar do PKTM. Como a variação do tempo de execução das cinco rodadas foi insignificante em todos os casos, optou-se por não apresentar intervalos de confiança para a média do tempo de execução e para os valores de *speedup*.

Analisando a Tabela 1, inicialmente, pode-se observar que as otimizações propostas conseguiram melhorar o tempo de execução da versão automática do código sequencial. É possível observar também que a versão OpenMP Manual com 6 *threads* obteve o melhor desempenho, com *speedup* geral igual a 9,4, seguida da versão MPI Manual que alcançou *speedup* geral de 9,1. Esse resultado mostra que a vetorização manual foi mais eficiente do que a vetorização automática por conseguir vetorizar uma quantidade maior de laços de execução do PKTM. Além disso, as versões com MPI apresentaram tempos de execução maiores do que as versões com OpenMP, o que pode ser explicado pelo tempo gasto com as operações de trocas de mensagens do MPI.

Em relação à vetorização automática, observa-se que diferentes técnicas de paralelismo (OpenMP e MPI) resultaram em diferentes níveis de vetorização do código. O compilador conseguiu vetorizar mais laços do algoritmo com OpenMP do que com MPI. A dificuldade de vetorizar o código com MPI ainda foi maior do que na versão sequencial. Embora o ganho de desempenho tenha se mantido constante entre as diferentes técnicas de vetorização e graus de paralelismo, a dificuldade de vetorização, por parte do compilador, foi maior quando se utilizou o padrão MPI.

Considerando o impacto da vetorização automática otimizada no MPI, verifica-se que, quando otimizado, o código com MPI foi capaz de alcançar *speedups* de vetorização iguais ao do código com OpenMP, isto é, 1,3 nas execuções com 2, 4 e 6 *threads*, no caso do OpenMP, e nas execuções com 2, 4 e 6 processos trabalhadores, no caso do MPI. É

Threads ou Proc. Trab.	Versão do PKTM	Tempo (em segundos)	Speedup da Vetorização	Speedup Geral
1	Sequencial Base	92,5	1,0	1,0
	Sequencial Auto	74,0	1,3	1,3
	Sequencial Auto Otimizado	69,7	1,3	1,3
	Sequencial Manual	43,6	2,1	2,1
2	OpenMP Base	46,5	1,0	2,0
	OpenMP Auto	35,1	1,3	2,6
	OpenMP Manual	23,5	2,0	3,9
	MPI Base	48,5	1,0	1,9
	MPI Auto	48,6	1,0	1,9
	MPI Auto Otimizado	37,1	1,3	2,5
	MPI Manual	23,6	2,1	3,9
4	OpenMP Base	24,2	1,0	3,8
	OpenMP Auto	18,2	1,3	5,1
	OpenMP Manual	12,7	1,9	7,3
	MPI Base	24,9	1,0	3,7
	MPI Auto	24,9	1,0	3,7
	MPI Auto Otimizado	19,2	1,3	4,8
	MPI Manual	12,8	2,0	7,3
6	OpenMP Base	16,6	1,0	5,6
	OpenMP Auto	12,9	1,3	7,1
	OpenMP Manual	9,8	1,7	9,4
	MPI Base	19,7	1,0	4,7
	MPI Auto	19,8	1,0	4,7
	MPI Auto Otimizado	15,6	1,3	5,9
	MPI Manual	10,1	1,9	9,1

Tabela 1. Tempo, *Speedup* da Vetorização e *Speedup* Geral obtidos.

possível observar também que, nestes casos, não houve alteração no ganho de desempenho alcançado pela vetorização diante da variação do número de *threads* ou processos trabalhadores.

Considerando o impacto da vetorização manual nas versões paralelas, nota-se uma leve redução do ganho de desempenho a partir do aumento da quantidade de *threads*, no caso do OpenMP e da quantidade de processos trabalhadores, no caso do MPI.

Finalmente, vale notar que o ganho de desempenho obtido ao se combinar a paralelização com a vetorização foi sempre maior do que o desempenho obtido com uso de apenas uma das duas abordagens. Portanto, pode-se concluir que, para o caso do PKTM, o uso em conjunto dessas duas técnicas diminuem significativamente o tempo total de execução dessa aplicação.

5. Conclusão e Trabalhos Futuros

Neste artigo, foram avaliadas diversas abordagens para vetorização em duas versões paralelas do PKTM, uma com OpenMP e outra com MPI. Através de diversos experimentos foi possível observar que os ganhos com as vetorizações automática e manual nas versões paralelas foram similares aos obtidos na versão sequencial. Além disso, verificou-se que

a vetorização automática produziu melhores resultados na versão com OpenMP do que nas demais, sequencial e com MPI.

As otimizações propostas para se alcançar a mesma eficiência de vetorização em todos os códigos se mostraram eficientes e apontam que seu uso em outros códigos, em que se deseje utilizar a vetorização automática, seja promissor.

Para trabalhos futuros, pretende-se implementar uma versão paralela híbrida utilizando tanto o MPI quando o OpenMP. Além disso, seria interessante avaliar vetorizações automática e manual do PKTM nas arquiteturas Intel MIC (*Many Integrated Core*) que permitem processar, ao mesmo tempo, dezesseis números de ponto flutuante com precisão simples. Por fim, pretende-se avaliar os efeitos da vetorização em outras aplicações paralelas de migração como a Migração Reversa no Tempo e a Migração por Mínimos Quadrados.

Referências

Claerbout, J. F. (1985). *Fundamentals of Geophysical Data Processing*. Pennwell Books.

Dai, H. (2005). Parallel Processing of Prestack Kirchhoff Time Migration on a PC cluster. *Computers & geosciences*, 31(7):891–899.

Hofmann, J., Treibig, J., Hager, G., e Wellein, G. (2014). Comparing the performance of different x86 simd instruction sets for a medical imaging application on modern multi and manycore chips. In *Proceedings of the Workshop on Programming models for SIMD/Vector processing*, páginas 57–64. ACM.

Intel (2012). A Guide to Vectorization with Intel® C++ Compilers. Relatório técnico.

Lomont, C. (2011). Introduction to Intel Advanced Vector Extensions. *Intel White Paper*.

Melo Alves, M., Cruz Pestana, R., Alves Prado da Silva, R., e Drummond, L. (2016). Accelerating Pre-stack Kirchhoff Time Migration by Manual Vectorization. *Concurrency and Computation: Practice and Experience*.

Mitra, G., Johnston, B., Rendell, A. P., McCreath, E., e Zhou, J. (2013). Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-powered ARM and Intel Platforms. In *27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, páginas 1107–1116. IEEE.

Panetta, J., Teixeira, T., de Souza Filho, P. R., da Cunha Filho, C. A., Sotelo, D., da Motta, F. M. R., Pinheiro, S. S., Rosa, A. L. R., Monnerat, L. R., Carneiro, L. T., et al. (2012). Accelerating Time and Depth Seismic Migration by CPU and GPU Cooperation. *International Journal of Parallel Programming*, 40(3):290–312.

Rizvandi, Nikzad Babaii e Boloori, A. J., Kamyabpour, N., e Zomaya, A. Y. (2011). MapReduce Implementation of Prestack Kirchhoff Time Migration (PKTM) on Seismic Data. In *12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, páginas 86–91. IEEE.

Schuchart, J., Waurich, V., Flehmig, M., Walther, M., Nagel, W. E., e Gubsch, I. (2015). Exploiting Repeated Structures and Vectorization in Modelica. In *11th International Modelica Conference*, páginas 265–272. Modelica Association Paris, France.

Shi, X., Li, C., Wang, S., e Wang, X. (2011). Computing Prestack Kirchhoff Time Migration on General Purpose GPU. *Computers & Geosciences*, 37(10):1702–1710.

Sun, P. e Shi, X. (2012). An OpenCL Approach of Prestack Kirchhoff Time Migration Algorithm on General Purpose GPU. In *13th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, páginas 179–183. IEEE.

Teixeira, D., Yeh, A., e Sampath Gajawada, T. (2013). Implementation of Kirchhoff Prestack Depth Migration on GPU. *SEG Technical Program Expanded Abstracts*, 3683:3686.

Xu, R., Hugues, M., Calandra, H., Chandrasekaran, S., e Chapman, B. (2014). Accelerating Kirchhoff Migration on GPU using Directives. In *First Workshop on Accelerator Programming using Directives (WACCPD)*, páginas 37–46. IEEE.

Yang, C.-T., Huang, C.-L., e Lin, C.-F. (2011). Hybrid CUDA, OpenMP, and MPI Parallel Programming on Multicore GPU Clusters. *Computer Physics Communications*, 182(1):266–269.

Yilmaz, O. e Doherty, S. M. (1987). *Seismic Data Processing*, volume 2 of *Investigations in Geophysics*. Society of Exploration.