TENDÊNCIAS EM ARQUITETURAS, APLICAÇÕES E PROGRAMAÇÃO PARALELA

Minicursos **WSCAD 2016**

Organizadores:
Edson Barbosa **Lisboa**
Wanderson Roger Azevedo **Dias**
Edward David **Moreno**

EDITORA IFS

Minicursos do WSCAD 2016

Simpósio de Sistemas Computacionais de Alto Desempenho

Editora
Editora do IFS (EdIFS)

Organizadores
Edson Barbosa Lisboa (IFS)
Wanderson Roger Azevedo Dias (IFS)
Edward David Moreno (UFS)

Realização
Universidade Federal de Sergipe (UFS)
Instituto Federal de Sergipe (IFS)

Promoção
Sociedade Brasileira de Computação (SBC)
Departamento de Computação (DCOMP/UFS)
Programa de Pós-Graduação em Ciência da Computação (PROCC/UFS)

Tendências em Arquiteturas, Aplicações e Programação Paralela

-- Minicursos do WSCAD 2016 --

Edson Barbosa Lisboa, Wanderson Roger Azevedo Dias, Edward David Moreno

Capa: Jéssika Lima

Conselho editorial: EDIFS

Arte final e diagramação: Crislaine Santo Macedo, Jéssika Lima Santos, Wandeerson Roger Azevedo Dias

Revisor gramatical: Ruth Sales Gama de Andrade

ISBN: 978-85-9591-045-4

Dados Internacionais de Catalogação na Publicação (CIP)

T291 Tendências em arquiteturas, aplicações e programação paralela [recurso eletrônico]: Minicursos WSCAD 2016 / Edson Barbosa Lisboa, Wanderson Roger Azevedo Dias, Edward David Moreno, organizadores. – Aracaju: IFS, 2017.
209 p. : il.

Formato: e-book
ISBN 978-85-9591-046-1

1. Arquitetura de computadores. 2. Programação paralela. 3. WSCAD. 4. Instituto Federal de Sergipe. 5. Universidade Federal de Sergipe. I. Lisboa, Edson Barbosa. II. Dias, Wanderson Roger Azevedo. III. Moreno, Edward David.

CDU: 004.2

Ficha catalográfica elaborada pela bibliotecária Célia Aparecida Santos de Araújo
CRB 5/1030

EDITORA IFS

Instituto Federal de Educação, Ciência e Tecnologia de Sergipe – IFS
Av. Jorge Amado, 1551 – Loteamento Garcia – Bairro Jardins – Aracaju-SE
CEP: 49.025-330 – Tel: (79) 3711-1437 – Email: edifs@ifs.edu.br
Impresso no Brasil – 2017

Ministério da Educação

Instituto Federal de Educação, Ciência e
Tecnologia de Sergipe

Presidente da República
Michel Miguel Elias Temer Lulia

Ministro da Educação
José Mendonça Bezerra Filho

Secretário da Educação Profissional e Tecnológica
Eline Neves Braga Nascimento

Reitor do IFS
Ailton Ribeiro de Oliveira

Pró-reitora de Pesquisa e Extensão
Ruth Sales Gama de Andrade

Tendências em Arquiteturas, Aplicações e Programação Paralela

-- Minicursos do WSCAD 2016 --

Organização

Edson Barbosa Lisboa

Wanderson Roger Azevedo Dias

Edward David Moreno

Autores

Edward David Moreno
Felipe dos Anjos Lima
Guilherme Esmeraldo
Igor Freitas
José Augusto Miranda Nacif
Márcio Castro
Marcos Ennes Barreto
Murilo do Carmo Boratto
Pedro H. Penna
Raphael Cóbe
Ricardo Ferreira
Rogério Iope
Silvio Stanzani
Wanderson Roger Azevedo Dias

Prefácio

A área de computação de alto desempenho tem despertado cada vez mais interesse por parte da comunidade científica e da indústria porque tem sido amplamente explorada para resolver problemas de alta complexidade em diversas áreas que, em geral, demandam processamentos de grandes volumes de dados, como por exemplo: sistemas de simulação, computação científica, realidade aumentada, análise e processamento de imagens, sistemas de navegação e muitas outras. Com a popularização e o avanço das arquiteturas paralelas de hardware e a consolidação das técnicas e ferramentas de programação paralela, tais áreas podem atingir resultados ainda mais relevantes e de interesses econômicos, científicos e sociais.

No Brasil, o Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD) é um evento vinculado ao tradicional *International Symposium on Computer Architecture and High Performance Computing* (SBAC-PAD), promovido há mais de vinte anos pela Sociedade Brasileira de Computação (SBC), e é realizado anualmente com o objetivo de apresentar os principais desenvolvimentos, aplicações e tendências nas áreas de arquitetura de computadores, processamento de alto desempenho e sistemas distribuídos.

O WSCAD 2016 (**XVII Simpósio em Sistemas Computacionais de Alto Desempenho**) foi realizado em Aracaju-SE, Brasil, no período de 04 a 07 de outubro. O evento WSCAD tem a responsabilidade de reunir a comunidade nacional responsável pelas pesquisas em Computação de Alto Desempenho, Arquitetura de Computadores e Sistemas Distribuídos. O WSCAD também abriga os seguintes eventos: Concurso de teses e dissertações (WSCAD-CTD); Workshop de Iniciação Científica (WSCAD-WIC); Workshop de Educação em Arquitetura de Computadores (WEAC), a Maratona Internacional de Programação Paralela e um conjunto de

minicursos que trazem o estado da arte na área como uma oportunidade de capacitação e atualização.

Este livro é o resultado da compilação de todo o material apresentado nos minicursos realizados no WSCAD 2016. Ele é composto de 6 capítulos com os seguintes temas:

- **Capítulo 1- Desenvolvimento de Aplicações Paralelas Eficientes com OpenMP:** esse capítulo apresenta uma discussão de como desenvolver aplicações paralelas eficientes usando o OpenMP, uma interface de programação de aplicações para arquiteturas paralelas de memória compartilhada que é largamente utilizada pela indústria e academia.
- **Capítulo 2 - Programação em Arquiteturas Paralelas utilizando Multicore, Multi-GPU e Co-processadores:** apresenta alguns casos de estudo com o objetivo de discutir os principais avanços na área de Computação Paralela e Distribuída no que concerne a programação destes sistemas.
- **Capítulo 3 - Desenvolvimento de Aplicações Paralelas de Alto Desempenho com Python:** apresenta Python como uma linguagem de programação alternativa em projetos de aplicações paralelas de alto desempenho.
- **Capítulo 4 - Introdução à Vetorização em Arquiteturas Paralelas Híbridas:** apresentar algumas das características oferecidas por compiladores C e C++, bem como extensões dessas linguagens e recursos da versão 4 do OpenMP que permitem explorar vetorização automática e vetorização guiada em tais arquiteturas.
- **Capítulo 5 - Computação Heterogênea com GPUs e FPGAs:** enfatizar as tendências, metodologias, plataformas e ferramentas para as duas arquiteturas alvo – GPUs e FPGAs.
- **Capítulo 6 - Implantação e Análise de Desempenho de um Cluster com Processadores ARM e Plataforma Raspberry Pi:** Apresenta a implementação e análise de desempenho (speedups e consumo de energia) de um cluster embarcado de baixo custo composto por processadores da arquitetura ARM e plataforma Raspberry Pi, usando para os testes as bibliotecas OpenMPI e MPICH-2, executando os programas dos *benchmarks* HPCC e HPL.

Assim, trazer tópicos relacionados ao estado da arte em Tendências em Arquiteturas, Aplicações e Programação Paralela, esperamos contribuir para a atualização e capacitação dos leitores, despertando o interesse por esses temas e, consequentemente, fomentando a evolução dessa importante área da Ciência da Computação.

Os Organizadores.

EDITORA IFS

Tendências em Arquitetura, Aplicações e Programação Paralela

WSCAD 2016

XVII Simpósio em Sistemas Computacionais de Alto Desempenho

Foto: Cidade de Aracaju - SE

Sumário

Capítulo 1

Desenvolvimento de Aplicações Paralelas Eficientes com OpenMP

Pedro H. Penna e Márcio Castro
Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD)
Departamento de Informática e Estatística (INE)
Universidade Federal de Santa Catarina (UFSC)

Resumo

O emprego de arquiteturas paralelas com quantidades massivas de núcleos de processamento é uma tendência na área de Computação de Alto Desempenho. Nesse cenário, para que seja feito uso pleno e eficiente da capacidade de processamento disponível, torna-se imprescindível o uso de bibliotecas de programação paralelas, as quais oferecem uma abstração para criação e gerenciamento de threads e processos. Nesse contexto, esse capítulo apresenta uma discussão de como desenvolver aplicações paralelas eficientes usando o OpenMP, uma interface de programação de aplicações para arquiteturas paralelas de memória compartilhada que é largamente utilizada pela indústria e academia. Para realizar essa discussão, os principais mecanismos oferecidos por essa interface de programação são apresentados e analisados através de exemplos práticos.

Abstract

The use of parallel architectures with massive amounts of processing cores is a trend in High Performance Computing. In this scenario, to fully exploit the available processing capability of these parallel architectures, the use of parallel programming libraries, which provide an abstraction for creating and managing threads and processes, is mandatory. In this context, this chapter discusses how to develop efficient parallel applications with OpenMP, an application programming interface for shared memory parallel architectures that is widely used by industry and academia. To accomplish this

discussion, the main mechanisms offered by this application programming interface are presented and analyzed through practical examples.

1.1. Introdução

Durante as três últimas décadas, avanços constantes de pesquisa na área de arquitetura de computadores e de tecnologia na fabricação de semicondutores possibilitaram que o desempenho de um único processador crescesse linearmente a uma taxa anual de 40% a 50% (LARUS; KOZYRAKIS, 2008). No entanto, a dissipação de potência no *chip*, inerente ao crescente número de transistores, impôs um limite físico ao desempenho máximo possível a ser alcançado. Para contornar esse problema, grande parte da indústria agora investe no projeto de processadores com diversos núcleos de processamento (processadores *multicore*).

De fato, processadores *multicore* já são largamente utilizados na Computação de Alto Desempenho (ASANOVIC et al., 2009). Nesse nicho, a tendência atual é o emprego de centenas, ou até mesmo milhares desses processadores, no projeto de arquiteturas paralelas com uma quantidade massiva de núcleos (*cores*). Essa tendência é confirmada pelo website Top500[1], que provê um ranking atualizado dos 500 supercomputadores mais poderosos do mundo. A Figura 1.1 mostra o número de *cores* do supercomputador mais poderoso do mundo ao longo dos anos segundo o ranking do *website* Top500.

No entanto, ao passo que uma quantidade massiva de *cores* entrega um grande poder computacional a essas arquiteturas paralelas, a alta latência de acesso aos demais

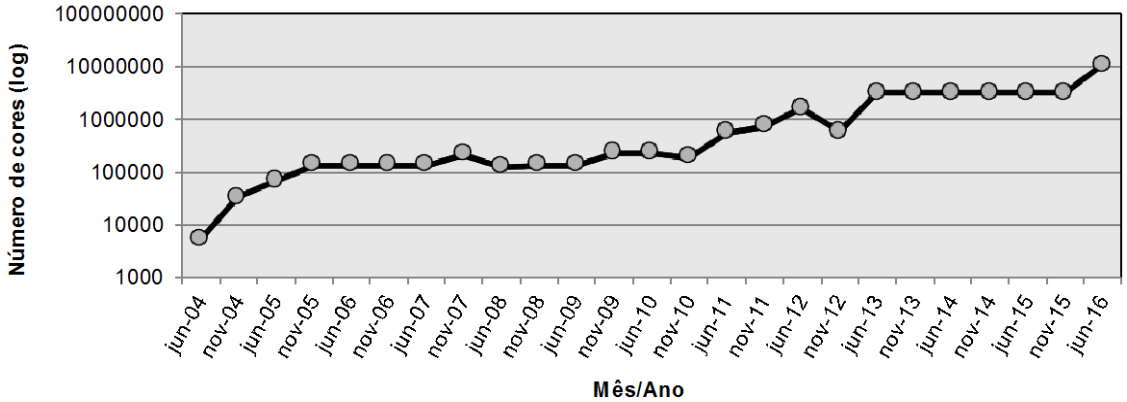


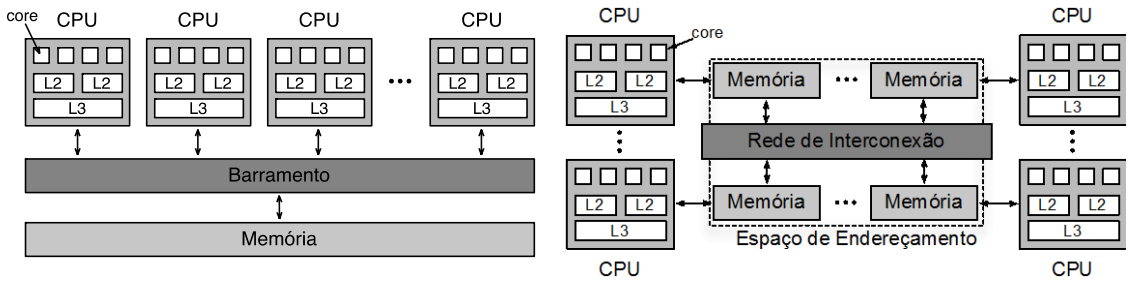
Figura 1.1 – Número total de *cores* do supercomputador mais poderoso do mundo. **Fonte:** Top500.

[1] <www.top500.org>

dispositivos, como memórias e unidades de armazenamento, impõe um desafio arquitetural no uso pleno de sua capacidade de desempenho. Para que essa barreira seja superada, o desenvolvimento de aplicações paralelas eficientes consiste em um requisito obrigatório. Felizmente, esse objetivo pode ser alcançado através do uso de bibliotecas de programação paralela, as quais oferecem uma abstração para criação, gerenciamento, sincronização e comunicação de threads e processos.

Nesse contexto, esse capítulo apresenta uma discussão de como desenvolver aplicações paralelas eficientes usando o OpenMP, uma Interface de Programação de Aplicações (*Aplication Programming Interface* – API) para arquiteturas paralelas de memória compartilhada que é largamente utilizada pela indústria e academia. Os principais mecanismos oferecidos por essa interface de programação são apresentados através de exemplos práticos, disponíveis publicamente sob licença GPLv3 em <www.github.com/ lapesd/teaching-openmp>, e analisados quantitativamente em uma máquina real com 4 processadores Intel Xeon X7 (6 *cores* cada).

Este capítulo está organizado da seguinte forma. A Seção 2 apresenta uma introdução ao modelo de programação OpenMP. A Seção 3 apresenta e discute as diretivas do OpenMP que podem ser utilizadas para realizar o paralelismo de dados. Então, a Seção 4 apresenta os mecanismos de sincronização básicos disponíveis no OpenMP. A Seção 5 apresenta e discute as diretivas do OpenMP que podem ser utilizadas para realizar o paralelismo de tarefas. Por fim, a Seção 6 apresenta as conclusões finais.



(a) UMA. (b) NUMA.

Figura 1.2 – Exemplos de arquiteturas de memória compartilhada.
Fonte: Elaborado pelos autores.

1.2. Conceitos Básicos

Essa seção irá abordar os conceitos básicos de programação paralela com OpenMP. Primeiramente, será apresentada uma discussão sobre as diferentes classes de arquiteturas paralelas existentes, salientando-se aquelas em que OpenMP poderá ser aplicado. Em seguida, será apresentado a modelo base dessa API. Por fim, serão apresentadas as diretivas básicas para a criação de regiões paralelas e as formas de compartilhamento de dados oferecidas pelo OpenMP.

1.2.1. Arquiteturas Alvo

De um modo geral, as arquiteturas paralelas podem ser classificadas em três grandes classes, de acordo com a organização de sua memória principal: (i) sistemas de memória distribuída; (ii) sistemas de memória compartilhada; (iii) sistemas de memória compartilhada e distribuída.

Em sistemas de memória distribuída, diversos computadores independentes, cada um executando seu próprio sistema operacional e com acesso exclusivo à sua memória principal, são interconectados através de uma rede e se comunicam através da troca de mensagens explícitas. Computadores, ou alternativamente nós computacionais, de um sistema de memória distribuída podem possuir ou não uma mesma configuração computacional. Quando os nós possuem uma mesma configuração e são interconectados através de uma rede de alto desempenho (baixa latência e alta largura de banda), o sistema é denominado *cluster*. Por outro lado, quando os nós possuem configurações heterogêneas e são geograficamente distantes o sistema é usualmente referido como *grid*. Sistemas de memória distribuída possuem a vantagem de ser altamente escaláveis, no entanto são de difícil programação, pois usualmente exigem o reprojeto de algoritmos. Uma API bastante utilizada nesses sistemas é a *Message Passing Interface* (MPI).

Por outro lado, em sistemas de memória compartilhada, um único computador, que é constituído por um ou mais processadores de múltiplos *cores* cada, executa apenas uma instância de um sistema operacional. A comunicação entre as *threads* que executam em *cores* distintos é feita pela memória principal, que têm seu acesso globalmente compartilhado. Sistemas de memória compartilhada possuem a vantagem

de fácil programação, no entanto não oferecem alta escalabilidade, como sistemas de memória distribuída.

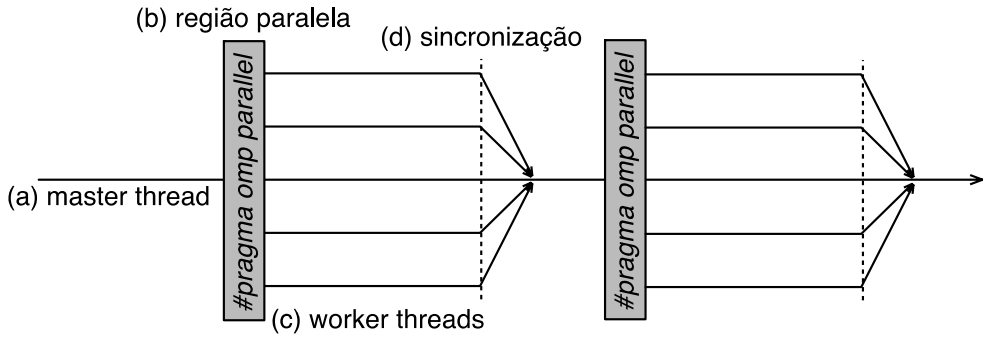


Figura 1.3 – Visão geral do modelo *Fork-Join*.
Fonte: Elaborado pelos autores.

Sistemas de memória compartilhada, por sua vez, podem ser classificados em duas classes, quanto ao tempo de acesso à memória principal compartilhada. Uma visão geral dessas classes é apresentada na Figura 1.2. Em arquiteturas do tipo *Uniform Memory Access* (UMA), o tempo de acesso de um core à memória é constante (Figure 1.2a). Por outro lado, em arquiteturas do tipo *Non-Uniform Memory Access* (NUMA), esse tempo varia em função da organização da memória em bancos (Figura 1.2b). Bancos de memória locais (próximos ao *core*) possuem uma latência de memória menor que bancos de memória remotos (distantes do *core*). Nesse sentido, o tempo de acesso à memória principal pode ser pequeno ou grande, dependendo da distância entre o processador ou *core* e o banco de memória que esse processador ou *core* está acessando. Arquiteturas do tipo NUMA possuem maior potencial de escalabilidade que arquiteturas UMA. Todavia, esses sistemas são mais complexos, caros e de difícil programação devido a questões relacionada à afinidade de memória entre *threads* (RIBEIRO et al., 2009).

Por fim, os sistemas de memória compartilhada e distribuída apresentam uma organização mista da memória (compartilhada e distribuída). Por exemplo, nós computacionais de *clusters* e *grids* podem possuir diversos *cores* cada. Nesse sentido, a memória de um computador é compartilhada exclusivamente entre seus *cores*. Alternativamente, em processadores *manycore* recentes, os *cores* são usualmente agrupados em *clusters*. Dentro de um cluster, todos os *cores* compartilham o acesso a uma memória. Porém, um *cluster* necessita realizar comunicações via rede de interconexão *intrachip* (usualmente, uma *Network-on-Chip* – NoC) para que possa acessar a memória de outros *clusters*. De fato, sistemas de memória compartilhada e

distribuída constituem uma tendência na Computação de Alto Desempenho, por consistirem em um compromisso entre as vantagens dos dois outros tipos de sistemas.

O OpenMP, assunto deste capítulo, consiste numa API para sistemas de memória compartilhada e utiliza o conceito de *threads* para possibilitar o compartilhamento de dados em memória. Por tanto, a API pode ser utilizada tanto para o desenvolvimento de aplicações paralelas em arquiteturas UMA quanto para arquiteturas NUMA.

1.2.2. Modelo Fork-Join

O OpenMP segue o modelo *Fork-Join* de execução paralela, que é ilustrado na Figura 1.3. Nesse modelo, o programa inicia sua execução com uma única *thread*, denominada *master thread* (Figura 1.3a). A *master thread* executa sequencialmente até encontrar uma região paralela (Figura 1.3b). Nesse ponto, a *master thread* cria um grupo de *threads* trabalhadoras, denominadas *worker threads* (Figura 1.3c), e cada *worker thread* executa então os comandos delimitados pela região paralela. Ao concluírem seu trabalho, as *worker threads* sincronizam suas atividades e terminam (Figura 1.3d). A *master thread* retoma então a execução sequencial do programa até que uma nova região paralela seja encontrada, momento em que todo esse processo se repete novamente.

Fragmento de Código 1.1 – Um exemplo simples com uma região paralela.

```
1  void sayhello(int nthreads)
2  {
3     void tids[nthreads]
4
5     /* Cria threads. */
6     #pragma omp parallel num_threads(nthreads)
7     {
8        int tid;
9
10       tid = omp_get_thread_num();
11       tids[tid] = tid;
12    }
13
14    /* Imprime os IDs das threads. */
15    for (int i = 0; i < nthreads; i++)
16       printf("thread %d: my ID is %d\n", i, tids[i]);
17 }
```

Como observação, é importante deixar claro que a *master thread* também executa os comandos na região paralela. Assim, se quatro *worker threads* são criadas pela *master thread*, um total de cinco *threads* irão executar a região paralela. No entanto, é possível fazer uso de estruturas condicionais alinhadas a funções internas do OpenMP para definir uma execução de comandos diferentes para a *master thread*. Esse assunto será abordado mais adiante na Seção 1.4. Por fim, vale ressaltar que, em implementações modernas do OpenMP, a criação das estruturas internas para gerência das *threads* em regiões paralelas é feita uma única vez, quando a primeira região paralela é encontrada. Dessa forma, a sobrecarga imposta na aplicação para a criação das *threads* não cresce linearmente com o número de regiões paralelas presentes.

1.2.3. Regiões Paralelas e Compartilhamento de Dados

A definição de regiões paralelas no OpenMP é feita através da diretiva `omp parallel`, como ilustrado no Fragmento de Código 1.1. Nesse exemplo, assim que a a *master thread* atinge a região paralela (linhas 6 a 12), ela cria um grupo de *worker threads* para executar os comandos especificados. Ao final da região paralela, uma barreira implícita força a sincronização das *threads*.

Por padrão, o número de *threads* no grupo que executará uma região paralela será igual ao número de *cores* do processador. No entanto, esse fator pode ser controlado através da cláusula `num_threads()` da região paralela, pela invocação da função utilitária `omp_set_num_threads()`, ou então pela definição da variável de ambiente `OMP_NUM_THREADS`. Em uma região paralela as *threads* são identificadas por um número único entre 0 e o número total de *threads*−1. Esse identificador pode ser recuperado em tempo de execução por cada thread em uma região paralela através da função utilitária do OpenMP denominada `omp_get_thread_num()`.

Variáveis declaradas fora da região paralela são compartilhadas entre todas as *threads* por padrão. No entanto, a cláusula `private()` pode ser usada para a especificação de variáveis locais privadas, e a diretiva `omp threadprivate()` pode ser empregada para a definição de variáveis globais privadas. Alternativamente, é possível definir todas as variáveis locais como privadas, através da cláusula `default()`, e então especificar as variáveis locais compartilhadas pela cláusula `shared()`.

O uso dos mecanismos apresentados anteriormente é fundamental no desenvolvimento de aplicações paralelas eficientes. O ajuste do número de *threads* em

uma região paralela evita o desperdício de recursos computacionais e possibilita um ajuste da granularidade de tarefas. A identificação de *threads* é necessária para a correta coordenação e atribuição de tarefas em uma aplicação. Por fim, mecanismos de controle de escopo de dados são inerentemente necessários em aplicações paralelas.

Fragmento de Código 1.2 – Paralelização da multiplicação de matrizes com OpenMP.

```
1  struct matrix *matrix_mult(struct matrix *a, struct matrix *b)
2  {
3     struct matrix *c;
4
5     c = matrix_create(a->nrows, b->ncols);
6
7     /* Multiplica matrizes. */
8     #pragma omp parallel for private(i, j, k)
9     for (int i = 0; i < a->nrows; i++)
10    {
11       for (int j = 0; j < a->cols; j++)
12       {
13          for (int k = 0; k < a->ncols; k++)
14             MATRIX(c, i, j) += MATRIX(a, i, k)*MATRIX(b, k, j);
15       }
16    }
17
18    return (c);
19 }
```

1.3. Paralelismo de Dados e Diretivas OpenMP

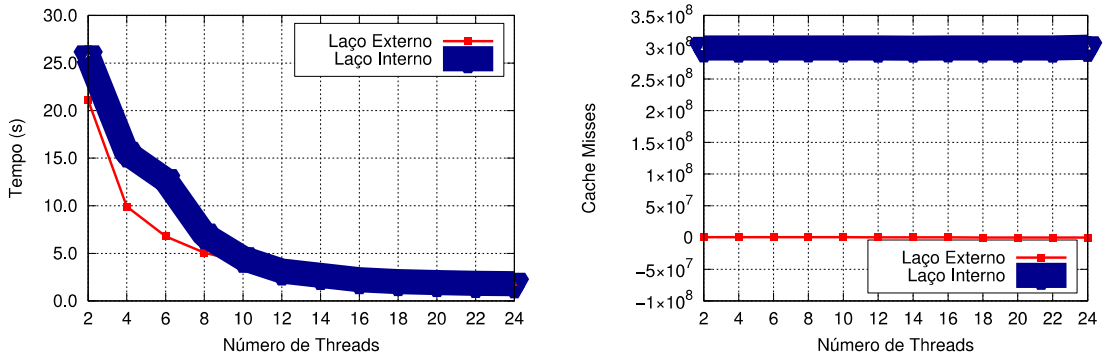
No OpenMP, o paralelismo de dados é explorado através de diretivas para paralelização de laços. Nessa seção, o uso dessas diretivas para o projeto de aplicações paralelas será discutido. Primeiro, serão apresentadas as diretivas que possibilitam explorar o paralelismo de dados. Em seguida, as estratégias de escalonamento disponíveis no OpenMP serão apresentadas, assim como suas vantagens e desvantagens. Por fim, os mecanismos de redução oferecidos pela API OpenMP serão introduzidos.

1.3.1. Paralelização de Laços

Duas diretivas de paralelização de laços estão disponíveis no OpenMP: `omp for` e `omp parallel for`. A primeira diretiva instrui que as iterações do laço seguinte de uma

região paralela devem ser distribuídas entre as *threads* do grupo em questão e, então, executadas em paralelo. A segunda diretiva tem o mesmo efeito, porém não necessita que o laço esteja dentro de uma região paralela.

O Fragmento de Código 1.2 ilustra o uso dessa última diretiva na paralelização do algoritmo clássico de multiplicação de matrizes. Nesse exemplo, um grupo de *threads* é criado para executar os comandos do laço da linha 9. Cada thread recebe um conjunto distinto de iterações e, assim, a computação é efetuada em paralelo.



(a) Tempo de execução. (b) Número de faltas de *cache*.

Figura 1.4 – Resultados das soluções de paralelização para a multiplicação de matrizes.
Fonte: Elaborado pelos autores.

Observe que nesse exemplo, assim como em diversas outras aplicações que também exploram o paralelismo de dados presente em laços, a diretiva de compilação também poderia ter sido colocada no segundo laço mais externo (linha 11) sem perda de semântica. No entanto, as duas abordagens se difeririam quanto à granularidade de paralelização. Na primeira abordagem, o grão de trabalho entregue a cada thread é mais grosso, enquanto na segunda abordagem o grão de trabalho é mais fino. O controle de granularidade consiste em uma técnica importante no projeto de aplicações paralelas eficientes, pois possibilita que a localidade temporal e de dados sejam efetivamente exploradas além de evitar sobrecargas de sincronização e contornar desbalanceamentos de carga durante a computação. Na Seção 1.3.2 serão discutidas as peculiaridades desse assunto em maiores detalhes.

Para ilustrar o impacto no desempenho que cada uma dessas abordagens teria no algoritmo clássico de multiplicação de matrizes, experimentos foram conduzidos com tamanho da matriz de entrada fixado em 1.680x1.680 e com o número de *threads* variando de 2 a 24. Durante as execuções, foram coletadas estatísticas de tempo de

execução e número de faltas na *cache* L2. A análise dos resultados de faltas na *cache* L2 (Figura 1.4b) revela que a abordagem de paralelização mais grossa conduz a um menor número de faltas, em contraste com a estratégia de paralelização do laço mais interno, para a plataforma considerada. Essa diferença conduz a um ganho de desempenho que é evidenciado nos resultados de tempo de execução da aplicação (Figura 1.4a). Além disso, é possível constatar que o ganho obtido com mais que 12 *threads* não é significantemente superior do que quando menos *threads* são usadas, indicando assim que na plataforma considerada o tamanho de problema não escala linearmente com o número de *threads*.

1.3.2. Escalonamento de Iterações

O escalonamento de iterações diz respeito ao modo como iterações de um laço paralelo são atribuídas às *threads* de um grupo. Por padrão, no OpenMP, o escalonamento de iterações é feito estaticamente e em blocos (ou *chunks*) de mesmo tamanho. Para aplicações que possuem um padrão de computação regular, isto é, o tempo de computação para cada iteração do laço paralelo é o mesmo, essa estratégia conduz a ganhos de desempenhos satisfatórios, pois esse particionamento estático é capaz de distribuir a carga de trabalho de forma uniforme entre as *threads*. O algoritmo de multiplicação de matrizes apresentado na seção anterior exemplifica a classe de algoritmos regulares. No caso, a carga de trabalho entregue a cada thread é constate e proporcional à:

$$Carga\ de\ Trabalho = \frac{Número\ de\ Linhas\ da\ Matriz}{Número\ de\ Threads}$$

No entanto, para aplicações que possuem um caráter irregular de computação, isto é, o tempo de computação para cada iteração no laço paralelo difere; ou então para aplicações regulares que possuem afinidade de memória entre diferentes iterações; essa estratégia não se mostra eficiente e escalável (CARIÑO; BANICESCU, 2008). Então, para atacar esses cenários problemáticos, o OpenMP disponibiliza a cláusula `schedule()` que possibilita: (i) selecionar a estratégia de escalonamento a ser empregada para escalonar as iterações de um laço paralelo; e (ii) definir quantas iterações são escalonadas por vez a uma única thread (tamanho do *chunk*). A estratégia

de escalonamento permite contornar o problema do desbalanceamento de carga presente em aplicações irregulares, enquanto o controle do tamanho de bloco de iterações a ser escalonado por vez permite ajustar a granularidade de paralelização, possibilitando assim que a afinidade de memória existente entre iterações seja explorada. O tamanho de bloco de iterações pode ser escolhido arbitrariamente, já a estratégia de escalonamento pode ser selecionada dentre as seguintes, em uma implementação padrão do OpenMP:

- `static`: particiona igualitariamente as iterações de um laço paralelo em *chunks* de iterações. A atribuição de *chunks* é feita em tempo de compilação e nenhuma sobrecarga é adicionada a aplicação. Essa estratégia é indicada para aplicações regulares.
- `dynamic`: atribui *chunks* de iterações às *threads* sob demanda em tempo de execução. Para que a atribuição seja feita dinamicamente durante a execução da aplicação, uma sobrecarga de gerência é introduzida na aplicação. Essa estratégia é indicada para uso em aplicações com alto grau de irregularidade.
- `guided`: similar à estratégia `dynamic`, porém o tamanho do *chunk* de iterações decresce com o tempo. Essa estratégia também impõe uma sobrecarga na execução da aplicação, porém relativamente menor que a estratégia `dynamic`. Essa estratégia é indicada para uso em aplicações com baixo e médio graus de irregularidade, onde as iterações iniciais do laço paralelo podem ser atribuídas em *chunks* maiores, e assim alguma sobrecarga de sincronização é evitada durante uma porção significativa do tempo total de execução da aplicação.

O Fragmento de Código 1.3 ilustra o uso desses dois mecanismos na paralelização do algoritmo clássico para multiplicação de matrizes esparsas, uma típica aplicação irregular. Nesse exemplo, o escalonamento de iterações é feito em blocos (*chunks*) de tamanho 1 com a política de escalonamento `dynamic`. Uma avaliação quantitativa do desempenho dessa estratégia nesse algoritmo particular, frente à estratégia `static`, é apresentada na Figura 1.5a. Nesses experimentos utilizou-se uma matriz de tamanho 1.680x1.680. O número de *threads* (*n*) foi variado de 2 à 12 e o tamanho dos *chunks* foi fixado em 1.680/*n* e 1, para as estratégias escalonamento `static` e `dynamic`, respectivamente. Para essa aplicação, tamanho de problema e plataforma, a análise dos resultados revela que a estratégia de escalonamento `dynamic`

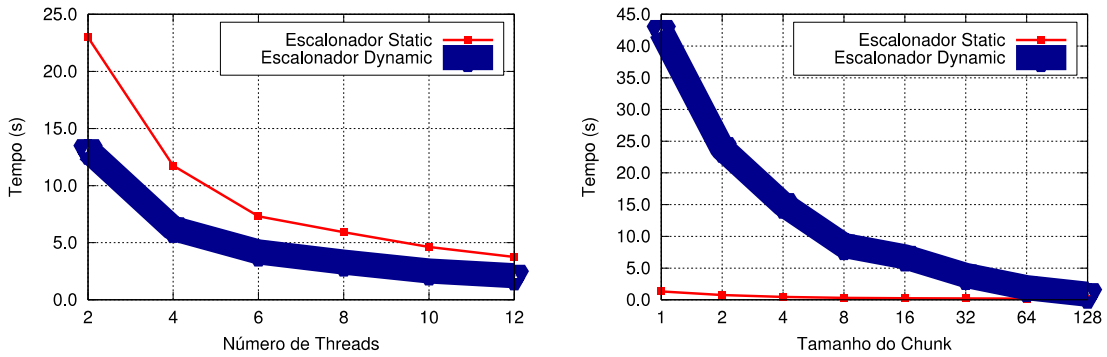
conduz a um melhor desempenho do que a estratégia `static`, se mostrando assim mais escalável. De fato, esse comportamento é recorrente em aplicações irregulares e pode ser utilizado para guiar o projeto de soluções paralelas eficientes nesse domínio.

Fragmento de Código 1.3 – Paralelização da multiplicação de matrizes esparsas com OpenMP.

```
1  struct matrix *sparsematrix_mult(struct matrix *a, struct matrix *b)
2  {
3     struct matrix *c;
4
5     c = matrix_create(a->nrows, b->ncols);
6
7     /* Multiplica matrizes esparsas. */
8     #pragma omp parallel for private(i, j, k) schedule(dynamic, 1)
9     for (int i = 0; i < a->nrows; i++) {
10       for (int j = 0; j < a->ncols; j++) {
11          for (int k = 0; k < a->ncols; k++) {
12             if (MATRIX(a, i, k) != 0)
13                MATRIX(c, i, j) += MATRIX(a, i, k)*MATRIX(b, k, j);
14          }
15       }
16    }
17
18    return (c);
19 }
```

No entanto, é pertinente observar que a granularidade de *chunks* consiste em um fator que impacta diretamente no desempenho entregue pela estratégia `dynamic` e, então, deve ser cuidadosamente considerado.

Para enfatizar a influência do tamanho do *chunk* no desempenho do escalonador `dynamic`, experimentos foram conduzidos com um *kernel* sintético que realiza uma computação regular (uma soma de um conjunto grande de números inteiros). Diferentes tamanhos de *chunk* foram analisados e as estratégias de escalonamento de iterações `static` e `dynamic` foram consideradas. Os resultados obtidos são apresentados na Figura 1.5b e evidenciam que na estratégia `dynamic`, *chunks* muito pequenos introduzem uma expressiva sobrecarga de sincronização na execução, que diminui com o aumento do tamanho do *chunk*. Em contraste, a estratégia `static` entrega um desempenho praticamente constante à aplicação, independente do tamanho do *chunk*.



(c) Escalabilidade das estratégias. (d) Análise do tamanho de *chunks*.

Figura 1.5 – Desempenho das estratégias `static` e `dynamic` na versão paralela do algoritmo clássico de multiplicação de matrizes esparsas.

1.3.3. Redução de Operações

Em aplicações que exploram o paralelismo de dados, frequentemente existe uma necessidade de que as *threads* combinem seus resultados privados de forma a produzir um resultado final para o problema sendo computado. Um suporte eficiente a essa operação, denominada redução, é oferecido pelo OpenMP através da cláusula `reduction`. O Fragmento de Código 1.4 ilustra o uso da operação de redução para o cálculo paralelo do produto escalar entre dois vetores a e b. A definição algébrica do produto escalar entre dois vetores ($a \cdot b$) é mostrada abaixo, onde $a = [a_1, a_2, \ldots, a_n]$ e $b = [b_1, b_2, \ldots, b_n]$:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

No Fragmento de Código 1.4, devido ao uso da diretiva `omp parallel for`, cada thread fica responsável por calcular o produto de um subconjunto dos elementos dos vetores a e b. Então, os resultados parciais encontrados por cada *thread* são somados devido ao uso da cláusula `reduction(+:prod)`, obtendo-se assim o cálculo do resultado final na própria varável `prod`.

Fragmento de Código 1.4 – Produto escalar com OpenMP.

```
1  int scalar_product(struct vector *a, struct vector *b)
2  {
3    int prod;
4
5    #pragma omp parallel for private(i) schedule(static) reduction(+:prod)
6    for (int i = 0; i < a->size; i++)
7       prod += VECTOR(a, i) * VECTOR(b, i);
8
9    return (prod);
10 }
```

Por padrão, o OpenMP oferece suporte para redução de operações aritméticas básicas sobre tipos primitivos da linguagem. Para tanto, uma cópia privada da variável a ser reduzida é criada em cada *thread* e, então, ao final da computação as variáveis são combinadas hierarquicamente segundo o operador especificado. Para tipos e operadores que o OpenMP não oferece suporte, é possível fornecer uma implementação eficiente e portável fazendo o uso de um laço paralelo, que efetua a computação de interesse, seguido por um laço sequencial que realiza a redução dos valores parciais computados.

1.4. Sincronização

Além de oferecer diretivas úteis para paralelização de porções de código, a API OpenMP também oferece diretivas para sincronização de *worker threads*. Essas diretivas são utilizadas dentro de regiões paralelas para: (i) indicar a necessidade de serialização de um trecho de código; (ii) garantir exclusão mútua ao acesso a dados compartilhados; ou (iii) realizar uma sincronização global entre *worker threads*. A seguir, são discutidas as principais diretivas de sincronização disponíveis na API OpenMP.

Serialização. A diretiva `omp single` permite que a execução de um trecho de código dentro de uma região paralela seja serializada, isto é, seja executada por apenas uma *worker thread*. O OpenMP também oferece uma diretiva similar para serialização de trechos de código denominada `omp master`. A diferença entre as primitivas `omp single` e `omp master` reside no fato de que, na primeira, qualquer *worker thread*

pode executar o trecho de código serializado, ao passo que, na segunda, somente a *master thread* executará o trecho de código serializado.

Exclusão mútua. Em muitos casos, dados compartilhados necessitam ser modificados dentro de uma região paralela por todas as *worker threads*. Nesses trechos de código, denominados *seções críticas*, faz-se necessária a utilização de um mecanismo de exclusão mútua para evitar que duas ou mais *worker threads* modifiquem simultaneamente dados compartilhados. Para esses casos, a API OpenMP oferece duas diretivas principais. A primeira delas, denominada `omp critical`, implementa um mecanismo de exclusão mútua clássico com uso de *locks* similar ao *mutex*. A segunda delas, denominada `omp atomic`, é semelhante à primeira, porém somente pode ser utilizada para proteger operações simples sobre uma variável compartilhada. Por exemplo, para uma variável compartilhada `x`, as operações permitidas pela diretiva `omp atomic` são: (i) leitura, e.g., `v = x`; (ii) escrita, e.g., `x = expr` (onde `expr` é uma expressão aritmética); (iii) atualização, e.g., `x++`; ou (iv) incremento/decremento, e.g., `v = x++`. É importante notar que, para o caso de uma seção crítica contendo uma única operação aritmética, ambas as diretivas podem ser utilizadas. Porém, nesse caso, a diretiva `omp atomic` oferecerá um melhor desempenho, pois a sua implementação utiliza *spin-locks* e instruções de *hardware* para garantir a atomicidade das operações.

Sincronização global. A API OpenMP também oferece uma diretiva que permite realizar uma sincronização global entre as *worker threads* na forma de uma *barreira*. Essa primitiva, denominada `omp barrier`, garante que todas as *worker threads* em uma região paralela somente possam executar as instruções posteriores à barreira quando todas as *worker threads* tiverem chegado a ela.

1.5. Paralelismo de Tarefas e Diretivas OpenMP

A partir da versão 3.0 foi introduzido o conceito de tarefas (*tasks*) no OpenMP. A criação de tarefas pode ser feita por uma ou mais *threads* em uma região paralela. Ao serem criadas, as tarefas que ainda não foram computadas são inseridas em uma estrutura de dados compartilhada entre as *threads* denominada “saco de tarefas” (*bag of tasks*). Dentro de uma região paralela, ao ficarem ociosas, as *threads* retiram tarefas ainda não computadas do saco de tarefas e as processam uma a uma.

A Figura 1.6 apresenta uma visão geral do modelo de tarefas implementado no OpenMP. Nesse modelo, é possível que algumas *threads* somente criem tarefas (representadas por flechas unidirecionais com linhas tracejadas), outras somente executem tarefas (representadas por flechas unidirecionais com linhas pontilhadas) e outras criem e executem tarefas (representadas por flechas bidirecionais com linhas contínuas). A criação e execução de tarefas pelas *threads* é feita dentro de uma região paralela utilizando a diretiva `omp task`. Quando uma tarefa é criada, a mesma é inserida no saco de tarefas e fica disponível para ser computada por qualquer *worker thread* ociosa dentro da região paralela.

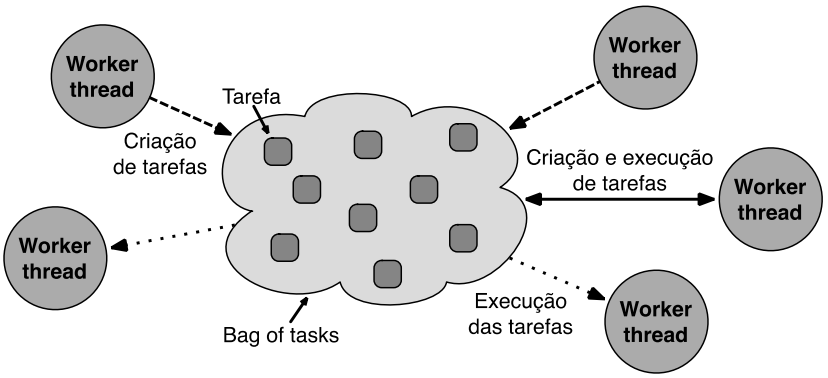


Figura 1.6 – Paralelismo de tarefas do OpenMP. Threads criam tarefas e as inserem em uma estrutura de dados compartilhada que contém tarefas ainda não computadas. Threads ociosas retiram tarefas dessa estrutura e as processam uma a uma.
Fonte: Elaborado pelos autores.

Em muitos problemas computacionais paralelizados com o uso de tarefas, determinadas tarefas não podem ser computadas pois dependem de resultados de outras tarefas ainda não finalizadas. Essas dependências podem ser vistas na forma de um grafo direcionado de dependências, onde cada nó do grafo representa uma tarefa e as arestas representam as dependências entre elas. A dependência entre tarefas é determinada no OpenMP através da diretiva `omp taskwait`. Essa diretiva permite que uma instrução ou um bloco de instruções em uma tarefa só seja executado após o término de todas as tarefas criadas anteriormente.

Um exemplo clássico de um problema recursivo que pode ser paralelizado com o uso de tarefas é o cálculo da sequência de Fibonacci. Matematicamente, os números da sequência de Fibonacci são gerados através da seguinte fórmula recursiva:

$$F_n = F_{n-1} + F_{n-2}, \text{ onde } F_1 = 1 \text{ e } F_0 = 0$$

Se considerarmos que para calcular F_n são necessárias duas tarefas F_{n-1} e F_{n-2}, é evidente que os resultados obtidos pelas tarefas F_{n-1} e F_{n-2} só poderão ser somados após o término de ambas tarefas. Nesse caso, nota-se que F_n depende de F_{n-1} e F_{n-2}. A Figura 1.7 mostra um exemplo de um grafo de dependências de tarefas para o cálculo da sequência de Fibonacci usando tarefas. Nesse exemplo, considera-se uma solução básica recursiva para o cálculo da sequência de Fibonacci com computações redundantes. Porém, é importante salientar que existem soluções mais otimizadas para esse problema que utilizam a técnica de *programação dinâmica* para evitar computações redundantes.

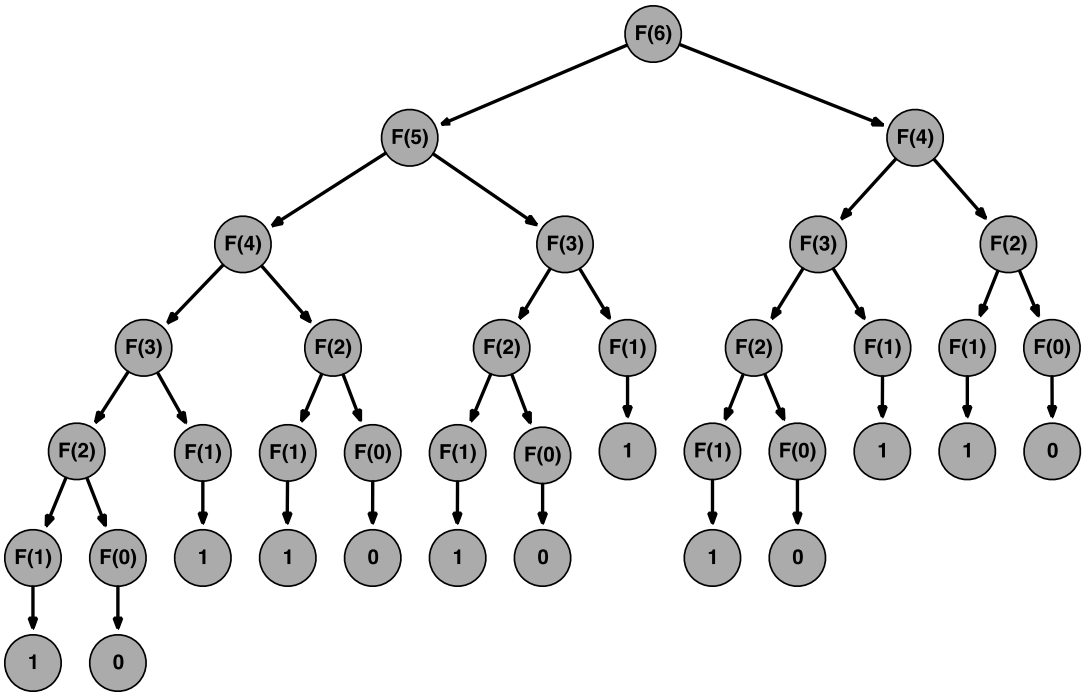


Figura 1.7 – Grafo de dependências entre tarefas para o cálculo da sequência de Fibonacci.
Fonte: Elaborado pelos autores.

O Fragmento de Código 1.5 mostra um exemplo de implementação básica (com computação redundante) para o cálculo da sequência de Fibonacci usando tarefas em OpenMP. Primeiramente é criada uma região paralela com a diretiva `omp parallel` (linha 5).

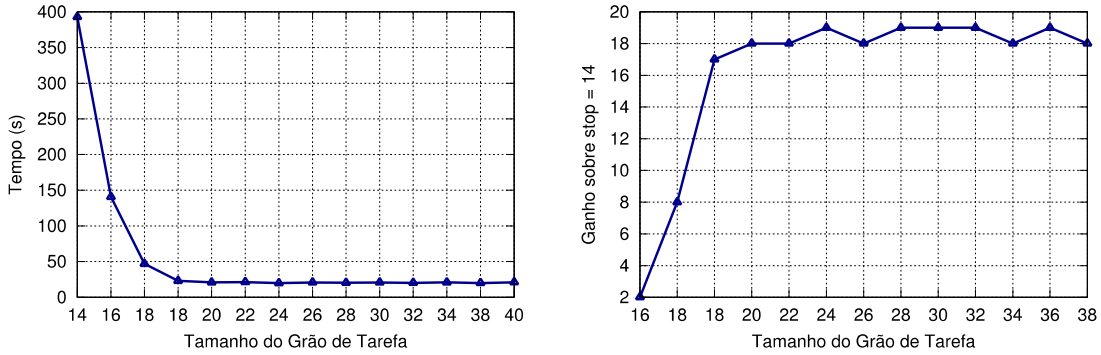
Fragmento de Código 1.5 – Implementação recursiva do cálculo da sequência de Fibonacci com tarefas em OpenMP.

```
1  unsigned long int fibonacci(int n, int stop)
2  {
3     unsigned long int result;
4
5     #pragma omp parallel
6     {
7        #pragma omp single nowait
8           result = recursive_fibonacci(n, stop);
9     }
10
11    return (result);
12 }
13
14 unsigned long int recursive_fibonacci(int n, int stop)
15 {
16    unsigned long int f1, f2, fn;
17
18    /* Condição de parada da recursão. */
19    if (n == 0 || n == 1)
20       return (n);
21
22    /* Condição de parada da criação de tarefas. */
23    if (n < stop)
24       return (recursive_fibonacci(n-1, stop) + recursive_fibonacci(n-2,
   stop));
25
26    /* Criação de tarefas e suas dependências. */
27    else {
28       #pragma omp task shared(f1)
29       f1 = recursive_fibonacci(n-1, stop);
30
31       #pragma omp task shared(f2)
32       f2 = recursive_fibonacci(n-2, stop);
33
34       #pragma omp taskwait
35       fn = f1 + f2;
36
37       return (fn);
38    }
39 }
```

Para permitir que somente uma única *worker thread* inicie a computação, e consequentemente, a criação da primeira tarefa, utiliza-se a diretiva `omp single` (linha 7). A primeira *worker thread* que ganhar acesso à região `omp single` executa uma chamada à função recursiva `recursive_fibonacci()`. A cláusula `nowait` permite que as demais *worker threads* avancem até a barreira implícita ao final da região paralela. Na função `recursive_fibonacci()`, tarefas correspondentes ao cálculo F_{n-1} e F_{n-2} são criadas com o uso da diretiva `omp task` (linhas 27-28 e 30-31) assim como as dependências são criadas com a diretiva `omp taskwait` (linhas 33-34). Como as tarefas são chamadas recursivas, novas tarefas poderão ser criadas.

Note que o Fragmento de Código 1.5 inclui uma condição de parada específica para a criação de tarefas (linha 23), evitando-se explicitamente a criação de tarefas quando o valor da variável `n` é inferior a um valor específico (`stop`). Basicamente, essa condição garante uma carga *mínima* para computação de uma tarefa. Em outras palavras, a computação será realizada sequencialmente por cada *worker thread* quando `n < stop`. Logo, a variável `stop` pode ser vista como uma forma de controle de granularidade das tarefas da aplicação. A condição de parada para criação de tarefas é muito importante, tendo em vista que a criação e gestão de tarefas no OpenMP possui um sobrecusto. Portanto, é importante que as tarefas tenham uma carga grande o suficiente de forma a reduzir o sobrecusto anteriormente mencionado. Todavia, tarefas contendo cargas extremamente grandes podem gerar desbalanceamentos de carga entre as *worker threads*.

Uma avaliação do impacto da granularidade das tarefas no desempenho da aplicação é apresentada na Figura 1.8. Nesses experimentos, o número de *threads* foi fixado em 12 e o valor da variável `n` foi fixado em 50. Os resultados mostram um ganho significativo de desempenho quando a granularidade das tarefas é aumentada (i.e., quando o valor usado na variável `stop` é aumentado) de 14 para 26 (ganho de desempenho de 19,8x). Com `stop = 26` tem-se então o melhor compromisso entre sobrecusto da geração/controle de tarefas e desbalanceamento de carga entre as *threads*.



(a) Tempos de execução com diferentes tamanhos de grão.

(b) Ganhos de desempenho com diferentes tamanhos de grão.

Figura 1.8 – Impacto da granularidade de tarefas no cálculo da sequência de Fibonacci.
Fonte: Elaborado pelos autores.

A partir desse ponto, o desempenho da aplicação volta a piorar ligeiramente. Esse comportamento é esperando, pois um grão muito grande pode gerar importantes desbalanceamentos de carga entre as *worker threads*. A perda de desempenho devido ao desbalanceamento de carga ocorre pois o tempo total da aplicação passa a ser determinado pelo tempo de execução da *worker thread* mais sobrecarregada.

1.6. Considerações Finais

Com a crescente tendência do emprego de arquiteturas paralelas na Computação de Alto Desempenho, o desenvolvimento de aplicações paralelas eficientes tornou-se um requisito obrigatório. Felizmente, essa demanda pode ser suprida através do uso de bibliotecas de programação paralela, as quais oferecem uma abstração para a criação, gerenciamento, sincronização e comunicação de *threads* e processos. Nesse contexto, esse capítulo apresentou uma discussão de como desenvolver aplicações paralelas de alto desempenho usando o OpenMP, uma API largamente utilizada pela indústria e academia para arquiteturas paralelas de memória compartilhada.

O OpenMP utiliza o conceito de *threads* e recai sobre o modelo paralelo de computação *Fork-Join*. Através de exemplos práticos, os principais mecanismos oferecidos por essa interface de programação foram apresentados e discutidos. O OpenMP oferece um conjunto de diretivas de compilação e funções utilitárias que dão suporte para exploração do paralelismo de dados e de tarefas. Além disso, mecanismos

oferecidos para o controle de escalonamento e o ajuste de granularidade de tarefas possibilitam o desenvolvimento de aplicações paralelas eficientes.

Referências

ASANOVIC, K. et al. A View of the Parallel Computing Landscape. Communications of the ACM, v. 52, n. 10, p. 56–67, 2009.

CARIÑO, R.; BANICESCU, I. Dynamic Load Balancing with Adaptive Factoring Methods in Scientific Applications. Journal of Supercomputing, Springer US, v. 44, n. 1, p. 41–63, 2008.

LARUS, J.; KOZYRAKIS, C. Transactional Memory: Is TM the Answer for Improving Parallel Programming? Communications of ACM, ACM, New York, USA, v. 51, n. 7, p. 80–88, 2008. ISSN 0001-0782.

RIBEIRO, C. P. et al. Memory Affinity for Hierarchical Shared Memory Multiprocessors. In: International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). São Paulo, Brazil: IEEE Computer Society, 2009. p. 59–66. ISBN 978-0-7695-3857-0.

Capítulo 2

Programação em Arquiteturas Paralelas utilizando Sistemas Multicore, Multi-GPU e Co-processadores

Murilo Boratto
Núcleo de Arquitetura de Computadores e Sistemas Operacionais (ACSO)
Universidade do Estado da Bahia (UNEB)
Salvador – Bahia – Brasil
muriloboratto@uneb.br

Marcos Barreto
Laboratório de Sistemas Distribuídos (LaSiD)
Universidade Federal da Bahia (UFBa)
Salvador – Bahia – Brasil
marcoseb@dcc.ufba.br

Resumo

Uma estratégia viável para acelerar a solução de problemas complexos com altos custos computacionais é explorar sistemas computacionais heterogêneos, formados por proces-sadores multicore, múltiplas unidades de processamento gráfico (GPU) e co-processadores. Neste texto, nós apresentamos alguns casos de estudo com o objetivo de discutir os principais avanços na área de Computação Paralela e Distribuída no que concerne a programação destes sistemas.

Abstract

A viable strategy for accelerating the solution of complex problems with high computati-onal costs is to exploit heterogeneous computing systems consisting of multicore CPUs, multiple (many-core) GPUs, and co-processors. In this text, we present some applied case studies in order to discuss some major advances in the area of Parallel and Distributed Computing regarding the programming of such systems.

2.1. Programação em Arquiteturas Paralelas: Perspectivas e Aplicações

A Programação de Alto Desempenho (PAD) surgiu da necessidade iminente de substituição da tecnologia de fabricação dos computadores, que atingiu um patamar limitado. Enquanto que a evolução histórica dos computadores, independente dos fabricantes, remonta para a ampliação da frequência de operação das máquinas como fator sistemático de suas capacidades, verifica-se que é de longe o fator determinante para a evolução das mesmas. Este fato torna-se inverídico já que um computador é um composto arquitetônico de componentes físicos que interagem de forma intrínseca. Não basta ampliar a frequência de operação de um processador sem aumentar sua capacidade de comunicação com a memória principal ou mesmo com os dispositivos com os quais interage. Sob esta ótica, o desenvolvimento de um computador está fisicamente relacionado com seus componentes e a forma como estão organizados.

A arquitetura atual dos computadores estava comprometida. Diversos fatores foram estudados, inúmeras organizações foram propostas e distintas abordagens foram implementadas, todas com o mesmo limitador físico. Não restaram opções senão modificar radicalmente a organização dos computadores para a exploração eficiente do paralelismo em nível de instruções (ILP) e de threads (TLP), através de técnicas de *pipelining*, processadores com *hyperthreading* [5], emulação de duas unidades lógico-aritméticas e, finalmente, processadores multicore com a materialização de múltiplos núcleos de processamento em um único chip.

Esta nova organização interna de um computador, valendo-se de uma arquitetura dotada de capacidade de processamento simultâneo, impulsionou uma série de novos requisitos que atingiram tanto os desenvolvedores de software quanto os projetistas de hardware. Os desenvolvedores foram implicados no que tange ao modelo de programação e de implementação de seus sistemas. A execução de um sistema projetado para um monoprocessador dificilmente executaria em um sistema multiprocessado, devido às particularidades de um frente ao outro. Os requisitos não conduzem apenas em adequações das aplicações ou da forma como o usuário final se relaciona com ela; implicam, para sua eficiente utilização, em um conhecimento apurado e específico da arquitetura do computador por parte do desenvolvedor. Neste aspecto, a Programação de Alto Desempenho abre um novo leque de oportunidades para

o setor da computação ao mesmo tempo em que impõe uma série de desafios para sua concretização.

Os projetistas necessitam definir uma arquitetura que permita a execução mais rápida e mais precisa das aplicações. Para tanto, se faz necessária a análise da demanda, observando o nível de complexidade apresentado pelas aplicações para decidir que tipo de solução pode ser utilizada: soluções proprietárias de supercomputadores ou soluções alternativas de baixo custo. Os supercomputadores são máquinas desenvolvidas com um propósito específico, valendo-se da integração massiva de processadores paralelos utilizando um único espaço de endereçamento para seus dados [7, 13]. Estes computadores foram utilizados durante algum tempo por serem a única solução plausível com poder de processamento para as demandas existentes. Entretanto, o custo deste tipo de solução é muito elevado para que possa ser popularizado entre os usuários dos sistemas de computação, além de apresentarem problemas pontuais como escalabilidade.

Uma alternativa é a Computação em clusters. Um cluster pode ser definido como um conjunto de computadores interligados via rede local que trocam informações, compartilham recursos e cooperam entre si para executar aplicações de forma simultânea. Funciona através da união do poder de processamento individual de cada equipamento sob a gestão de um *middleware* para que este conjunto opere como um único equipamento de capacidade mais elevada, escalabilidade ilimitada e custo proporcional à capacidade computacional [11].

A utilização de Programação de Alto Desempenho para disponibilização de recursos computacionais a custos aceitáveis tem sido um tema de relevância mundial e tem despertado o interesse de grandes empresas, universidades e governos. A chave da Programação de Alto Desempenho é prover uma estrutura capaz de fornecer o desempenho necessário para suprir as limitações técnicas das arquiteturas convencionais e financeiras das soluções proprietárias.

As seções a seguir consistem em um material didático do minicurso de Programação em Arquiteturas Paralelas utilizando Sistemas Multicore, Multi-GPU e Co-processadores. São abordadas novas tendências e perspectivas nas questões da programação de aplicações, caracterizando a arte de programar em paralelo utilizando arquiteturas com múltiplos núcleos, através de exemplos e aplicações, e de algumas tendências atuais em pesquisa.

2.2. Arquiteturas Multicore, Multi-GPU e Co-processadores

As arquiteturas com múltiplos núcleos atuais podem conter dois, quatro, seis ou mais núcleos complexos de processamento, sendo muito empregadas na exploração do paralelismo em nível de instruções e de threads. Em teoria, tais arquiteturas poderiam ser estendidas para dezenas ou mesmo centenas de núcleos no futuro, mas esbarram em dois obstáculos principais: a diferença entre as velocidades do processador e da memória, conhecida como *memory wall* e a geração excessiva de calor devido às altas frequências necessárias ao funcionamento de tais arquiteturas [3].

Tais obstáculos também impulsionaram o desenvolvimento das arquiteturas *many-core* baseadas em unidades de processamento gráfico (GPU). Uma GPU pode ser vista como um acelerador ou co-processador gráfico montado em uma placa de vídeo, o qual é composto por centenas de processadores simples capazes de explorar eficientemente o paralelismo em nível de dados (DLP) através da execução concorrente de múltiplas threads. Arquiteturas *many-core* atuais são largamente usadas para a execução de aplicações intensivas de dados (*data-intensive*) e são empregadas em sistemas embarcados, servidores de alto desempenho, dispositivos móveis e consoles de jogos. Por um outro lado, também surgiram avanços tecnológicos que introduziram uma capacidade de processar mais de um elemento por vez na família de processadores Pentium, tendo como base suas instruções SIMD (*Single Instruction Multiple Data*). A esta solução proprietária chamou-se de co-processadores, sendo que a ideia base consiste em implementar grandes vetores de 512 bits para realizar instruções SIMD.

Além do benefício para *multi-threading*, sua carga de trabalho pode também fazer uso intenso destas unidades de vetorização para obter um aumento significativo de desempenho.

Logo com o advento das arquiteturas multicore, multi-GPU e co-processadores, o emprego de métodos de programação paralela para o desenvolvimento de aplicações é absolutamente necessário para se alcançar um bom desempenho. Para apoiar os desenvolvedores nas tarefas envolvidas neste processo, várias ferramentas com diferentes capacidades precisam ser utilizadas no desenvolvimento de aplicações com problemáticas específicas. Dentro desse contexto, ferramentas computacionais permitem a utilização da Computação de Alto Desempenho por um maior número de usuários, auxiliando programadores iniciantes na construção de programas com

características de escalabilidade, desempenho e eficiência na utilização de recursos, bem como favorecendo a produtividade no desenvolvimento de *software* paralelo.

2.3. Ferramentas

A programação de arquiteturas paralelas não é uma tarefa trivial, uma vez que o programador precisa conhecer detalhes do *hardware*, escolher um modelo de programação adequado à aplicação e ao hardware e, por fim, dominar uma ferramenta de programação capaz de implementar o modelo de programação escolhido na arquitetura alvo. Historicamente, inúmeras ferramentas de programação paralela e distribuída têm sido propostas com o objetivo comum de prover eficiência no uso dos recursos da arquitetura paralela e um nível satisfatório de facilidade de uso (ou abstração).

Praticamente todos os fabricantes de arquiteturas paralelas fornecem bibliotecas de programação especificamente desenvolvidas para suas arquiteturas, como é o caso do *Threading Building Blocks* (TBB) [12] da Intel, *OpenCL* [4] da Apple e *CUDA* [8] pertencente a NVIDIA. Diversas propostas de ambientes e ferramentas de programação paralela de código aberto estão igualmente disponíveis para os mais diversos tipos e modelos de arquiteturas paralelas. Neste contexto destacam-se os padrões MPI [6] e *OpenMP* [10], em suas diversas implementações.

Atualmente, com a crescente adoção de arquiteturas híbridas formadas por processadores multicore, múltiplas GPUs e co-processadores, faz com que as ferramentas de programação paralela sejam cruciais para que tais arquiteturas possam ser eficientemente exploradas por um número cada vez maior de usuários na execução de aplicações em diversas áreas do conhecimento. Nesta seção são descritas algumas das principais ferramentas de programação paralela usadas atualmente nestas arquiteturas.

2.3.1. Programação OpenMP

OpenMP pode ser resumido como um conjunto de diretivas de compilação, funções de uma biblioteca e variáveis de ambiente que podem ser usadas para a expressão de paralelismo em linguagens de alto nível tais como C e C++. Atualmente, é um padrão de programação para memória compartilhada utilizado em sistemas multicore e

computadores de alto desempenho baseados em memória virtual. Nesta seção são analisadas algumas características básicas da ferramenta, tendo como base os exemplos que se encontram em [2].

2.3.1.1. Exemplo básico: aproximação da integral definida

Um exemplo típico é o cálculo do valor de *pi* por integração numérica, a partir da integral:

$$\int_0^1 \frac{1}{1+x^2}\,dx = \frac{\pi}{4}$$

Uma possível solução é aproximar da área da integral as áreas de retângulos de uma certa base. Quanto menor for a base, mais retângulos são obtidos, logo haverá uma melhor aproximação ao valor final da área. O algoritmo sequencial para este problema contém um laço for no qual se acumulam as áreas do retângulo aproximando-se do valor da integral. Uma versão paralela do mesmo problema para *OpenMP* é apresentada no **Algoritmo 1**. É possível incluir paralelismo no código apenas indicando a forma com que se deve distribuir o trabalho dos laços (loops) para os diferentes *threads*.

A partir do **Algoritmo 1** podemos fazer alguns comentários:

- Há a inclusão da biblioteca *OpenMP* (omp.h) no código (linha 2).
- A diretiva de paralelismo para *OpenMP* junto à interface C é indicada com *#pragma omp* (linha 16).
- A diretiva *parallel* indica que se inicializam vários *threads* para trabalharem em paralelo dentro do bloco de sentenças no laço *for* (linha 18).

Algoritmo 1 Cálculo do valor de *p* por integração numérica utilizando OpenMP.

```
1:  #include <stdio.h>
2:  #include <omp.h>
3:
4:  double f(double a){
5:    return (4.0 / (1.0 + a . a));
6:  }
7:
8:  int main(int argc, char ..argv){
9:    int n = atoi(argv[1]), i;
10:   double PI25DT = 3.14159265358979323846264 3;
11:   double pi, h, sum, x, t1, t2;
12:
13:   h = 1.0 / (double) n;
14:   sum = 0.0;
15:
16:   #pragma omp parallel for reduction(+:pi)
17:     private(x,i)
18:     for (i = 1; i <= n; ++i) {
19:         x = h . ((double)i - 0.5);
20:         sum += f(x);
21:     }
22:
23:     pi = h . sum;
24:
25:     printf("     pi  %.16f\t,
26:                  erro  %.16f\t",
27:                  pi, fabs(pi - PI25DT));
28:
29:   return 0;
30:   }
```

O *OpenMP* emprega o modelo de execução chamado *fork-join*, o qual consiste em executar-se inicialmente um *thread* principal até que apareça o primeiro construtor paralelo, a partir do qual são criados outros. Ao final do construtor, todos os *threads* são sincronizados e a execução continua de forma sequencial até o próximo construtor paralelo. A execução do **Algoritmo 1** tem os seguintes passos:

- No início da execução do código, existe um único *thread*, que tem uma série de variáveis (**int** *n*, *i*; **double** *pi*, *h*, *sum*, *x*) que estão na memória do sistema. Este thread recebe como parâmetro de linha de comando o número de intervalos a serem usados (variável *n*) e inicializa as variáveis *h* e *sum*.

- Ao chegar no construtor *#pragma omp parallel*, vários *threads* escravos são inicializados (parte *fork* do modelo). Os *threads* estão numerados de 0 até o número de *threads*-1. O modelo segue um paradigma mestre-escravos, de tal

forma que, os *threads* trabalham em paralelo no bloco que aparece após o construtor.

- Por ser um construtor *parallel for*, o que se paraleliza é o trabalho do laço for entre o conjunto de threads. Como o laço tem *n* passos, se dispormos, por exemplo, de 4 *threads*, a divisão do trabalho consiste em atribuir a cada *thread* *n*=4 passos. Como não se indica como realizou-se a divisão, se atribui os *n*=4 primeiros passos a *thread* 0, os seguintes *n*=4 passos a *thread* 1, e assim sucessivamente.
- Todas as variáveis (*n*, *i*, *pi*, *h*, *sum*, *x*) estão em uma memória global acessível a todos os *threads*. Variáveis privadas aos threads são indicadas pelo construtor *private*(x, i).
- Algumas variáveis da memória são compartilhadas de uma maneira especial, como por exemplo, (**reduction**(+:*pi*)). A palavra reservada *reduction* indica o cálculo paralelizado das somas reduzidas. Cada thread tem um valor diferente de *i* e realiza cálculos para valores diferentes de *x*, resultando em áreas de retângulos diferentes.
- Ao acabar o laço *for* (linha 21), há sincronização de todos os *threads* e os escravos morrem, ficando somente o *thread* mestre (parte *join* do modelo). O mestre fica responsável por combinar as soluções parciais e mostrar o valor final.

2.3.1.2. Compilação e execução

Para a compilação faz-se necessário dispor de um compilador que possa interpretar os *#pragmas* que aparecem no código. O *gcc* (*GNU Compiler Collection*) tem esta capacidade desde a versão 4.1. Também pode-se dispor de versões comerciais, tais como o compilador *icc* da Intel. A opção de compilação em *gcc* é *-fopenmp* e em *icc* é *-openmp*. Assim, o comando:

gcc codigo.c -o objeto –fopenmp

cria o objeto a partir do **Algoritmo 1**, que poderá ser executado em paralelo inicializando vários *threads*. A execução se realiza como em qualquer outro programa, mas tem-se que determinar quantos *threads* interferirão na região paralela. Existe uma

variável de entorno (OMP_NUM_THREADS) que indica esse número. Se não for inicializada, tem-se um valor padrão definido com o número de núcleos do processador corrente. Outra possibilidade é fazer um OMP_NUM_THREADS=*valor*, com que estabelecemos o número de threads na região paralela igual a valor, independentemente do número de núcleos disponíveis.

Podemos experimentar com um único *thread* e executar o programa tomando tempos com dados de entrada de tamanho variável, e na continuação variar o número de threads, tentando medir tempos de execução com a mesma entrada:

```
OMP_NUM_THREADS=1 time ./objeto
OMP_NUM_THREADS=2 time ./objeto
OMP_NUM_THREADS=4 time ./objeto
OMP_NUM_THREADS=6 time ./objeto
```

2.3.2. Programação CUDA

Uma unidade de processamento gráfico (GPU) é um processador especializado em renderização de gráficos 3D. Usualmente é usada em videogames, computadores pessoais, estações de trabalho, chegando a estar presente, hoje em dia, em telefones celulares e equipamentos portáteis que suportam multimídia. Criadas inicialmente para processar apenas gráficos de videogames ou de computadores, as GPUs possuem diversas outras funções importantes. Devido a este propósito inicial são bastante poderosas e eficientes para manipular gráficos computacionais em operações como adição de efeitos de iluminação, suavização de contornos de objetos e criação de imagens. As GPUs recebem os objetos vindos da CPU, os transformam, segundo informações também advindas da CPU e geram as imagens para a visualização do usuário [1].

Além da capacidade de processar imagens, a GPU também é usada em cálculos matemáticos e geométricos complexos, devido a sua capacidade de processar vetores ou matrizes com extrema facilidade. Essa grande velocidade e poder para cálculo matemático vem do fato das GPUs modernas possuírem muito mais circuitos para processamento do que para caching de dados e controle de fluxo que as CPUs.

CUDA (*Compute Unified Device Architecture*) é uma arquitetura de computação paralela de propósito geral que tira proveito dos recursos de computação paralela das unidades de processamento gráfico (GPUs) para resolver muitos problemas

computacionais complexos. O modelo de programação fundamenta-se na construção de um ou mais *kernels* (funções que serão executadas na GPU), a fim de aumentar o desempenho dos algoritmos. Dessa forma, o programador pode estabelecer quais partes do código irão executar na(s) CPU(s) e quais partes do código irão executar na(s) GPU(s).

2.3.2.1. Exemplo básico: visualização das informações da GPU

Nesta subseção é apresentado um exemplo de programa escrito em CUDA. O seguinte código, adaptado de [9], implementa a visualização das informações da(s) placa(s) gráfica(s) contidas na plataforma de execução.

Algoritmo 2 Visualização das propriedades das GPUs contidas no sistema de execução.

```
1:  #include <stdio.h>
2:  #include <cuda.h>
3:
4:  int main(int argc, char *argv[]){
5:
6:     int num_gpus = 0;
7:
8:     cudaGetDeviceCount(&num_gpus);
9:
10:    if(num_gpus < 1){
11:        printf("Nenhum dispositivo compativel com CUDA foi detectado\n");
12:        return 1;
13:    }
14:
15:    printf("Quantidade de dispositivos CUDA:\t%d\n", num_gpus);
16:
17:    for(int i = 0; i < num_gpus; i++){
18:        cudaDeviceProp dprop;
19:        cudaGetDeviceProperties(&dprop, i);
20:        printf("%d: %s\n", i, dprop.name);
21:    }
22:    cudaThreadExit();
23:    return 0;
24: }
```

O **Algoritmo 2** inicializa uma variável chamada *num_gpus* (linha 6), que armazena o número de GPUs contidas no sistema de execução. Logo em seguida, faz se um tratamento de exceção, caso nenhuma GPU seja detectada no sistema (linha 10). Palavras reservadas da biblioteca CUDA, tais como *cudaDeviceProp* e *cudaGetDeviceProperties* explicitam as propriedades intrínsecas3 da GPU (modelo e tipo) (linhas 18 e 19). Ao final, o código imprime as caraterísticas do ambiente de

execução contendo GPUs.

2.3.2.2. Soma de Vetores

O seguinte código, obtido em [9], implementa a soma de dois vetores, traduzida em um produto escalar $Y = A X + Y$. Dados dois vetores (X e Y) de tamanho N e um escalar A, o produto escalar é definido como o somatório dos produtos dos elementos de mesmo índice, ou seja, *X[0] Y [0] + X[1] Y [1] + ... + X[n] Y [n]*. Utilizando CUDA, é possível paralelizar o somatório dos produtos. Analisando o **Algoritmo 3**, percebe-se o fluxo de execução de um programa CUDA. Nas linhas 29 e 30 tem-se a alocação de memória na GPU. Nas linhas 33 e 35, tem-se a cópia dos dados para a memória da GPU. Com os dados na GPU parte-se para a execução do código (linha 39). Por fim, copiam-se os dados para a CPU (linha 41) e libera-se a memória da GPU (linhas 44 e 45).

2.3.2.3. Compilação e execução

O compilador fornecido pela NVIDIA para a utilização do CUDA é o *nvcc*. O seu funcionamento é regido por diversas ferramentas para diferentes estágios de compilação. Assim, o comando

nvcc codigo.cu -o objeto

cria o objeto a partir do **Algoritmo 3**, que poderá ser executado em paralelo inicializando vários threads. Outra abordagem possível da utilização de *threads* é com a criação de *grids*. Os diversos blocos de *threads* são organizados em um nível acima de abstração, em *grids* com uma ou múltiplas dimensões, como se cada bloco de threads fosse um elemento de uma matriz. O acesso aos blocos é realizado utilizando a variável (*blockIdx*), que permite a indexação utilizando *blockIdx.x* ou *blockIdx.y*, assim como na variável *threadIdx*. A Figura 2.1 ilustra esta organização.

Algoritmo 3 kernel CUDA que realiza a soma de dois vetores.

```
1:  #include <stdio.h>
2:  #include <cuda.h>
3:
4:  __global__ void saxpy_parallel(int n,  float *x, float *y){
5:
7:    int idGrid=blockIdx.y*gridDim.x+blockIdx.x;
8:    int idBlock=threadIdx.y*blockDim.x+threadIdx.x;
9:    int i=idGrid*(blockDim.x*blockDim.y)+idBlock;
10:
11:   if(i < n)
12:     y[i] += x[i];
13:   }
14:
15:   void generateVector(float *vector, int n){
16:     for (int i=0; i < n ; ++i)
17:       vector[i] = i + 1;
18:   }
19:
20:   int main(int argc, char *argv[]){
21:     int n = atoi(argv[1]);
22:     float *xh=(float*)malloc(sizeof(float)*n);
23:     float *yh=(float*)malloc(sizeof(float)*n);
24:     float *xd, *yd;
25:
26:     generateVector(xh, n);
27:     generateVector(yh, n);
28:
29:     cudaMalloc((void **) &xd,sizeof(float)*(n));
30:
31:     cudaMalloc((void **) &yd,sizeof(float)*(n));
32:
33:     cudaMemcpy(xd, xh, sizeof(float) * n, cudaMemcpyHostToDevice);
34:
35:     cudaMemcpy(yd, yh, sizeof(float) * n, cudaMemcpyHostToDevice);
36:
37:     dim3 dimGrid(10,10);
38:     dim3 dimBlock(12,12);
39:     saxpy_parallel<<<dimGrid, dimBlock>>>(n, xd, yd);
40:
41:     cudaMemcpy(yh, yd, sizeof(float) * (n), cudaMemcpyDeviceToHost);
42:
44:     cudaFree(xd);
45:     cudaFree(yd);
46:     free(xh);
47:     free(yh);
48:
49:     return 0;
50:   }
```

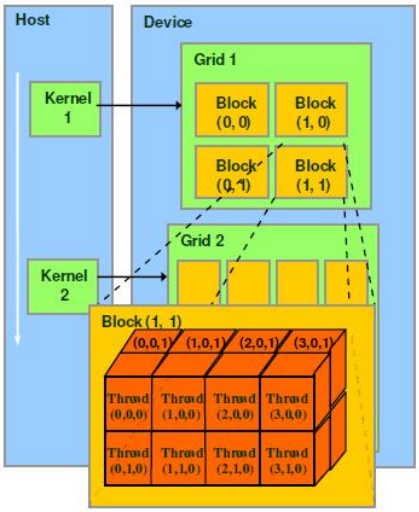


Figura 2.1 - Organização de grids, blocos e threads em CUDA.
Fonte: Elaborada pelos autores.

A etapa inicial de compilação começa pelo tratamento que é dado aos marcadores introduzidos pelo CUDA, que estão presentes em variáveis e funções ao longo do código. Em uma primeira etapa, o compilador separa os códigos específicos de cada arquitetura, neste caso da GPU (*device*) e do CPU (*host*). O resultado final desta separação é código C puro para o *host* e um objeto (formato binário) para o *device*.

2.3.3. Programação Híbrida

Com o advento das arquiteturas multicore e multi-GPU, a implementação de métodos de programação paralela no desenvolvimento de aplicações é necessária para alcançar um bom desempenho. Para apoiar os desenvolvedores em múltiplas tarefas envolvidas neste processo, ferramentas de diferentes âmbitos precisam ser utilizadas no desenvolvimento de aplicações referentes a problemáticas específicas. Dentro desse contexto, ferramentas combinadas como *OpenMP* e *CUDA* permitem a utilização da Computação de Alto Desempenho, auxiliando na construção de aplicações com características como escalabilidade, desempenho e produtividade.

2.3.3.1. Exemplo básico: OpenMP + CUDA

Para o exemplo a seguir, considera-se um sistema heterogêneo composto por múltiplos núcleos (*cores*) e múltiplas GPUs, o qual provê alta escalabilidade e permite o particionamento da carga de trabalho de forma proporcional à capacidade de

computação. O **Algoritmo 4** mostra o trecho de código utilizado para a partição para o sistema em estudo.

Algoritmo 4 Trecho de código híbrido: OpenMP + CUDA

```
1: omp_set_num_threads(nthreads);
2:
3:  #pragma omp parallel {
4:  int thread_id = omp_get_thread_num();
5:
6:  if( sizeGPU ) { /. processamento na GPU ./
7:    int gpu_id = thread_id;
8:    int first = thread_id . sizeGPU;
9:    cudaSetDevice(gpu_id);
10:   matrixGPU(...);
11:   }else {
12:     if( sizeCPU ) { /. processamento na CPU ./
13:       cpu_thread_id=thread_id;
14:       first = sizeGPU.(gpu_id+1) + cpu_thread_id.sizeThr;
15:       size = thread_id == (nthreads-1)?sizeLth:sizeThr;
16:       matrixCPU(...);
18:     }
19:   }
20: }
```

O trecho de código mostra a estratégia para utilizar o sistema heterogêneo. Inicializamos múltiplos threads de forma proporcional ao número de GPUs (linha 4). Cada thread, de forma autônoma, interliga-se a uma GPU através do identificador único das threads (linha 7). O particionamento da carga de trabalho se dá através de uma estratégia estática para enviar dados e tarefas para os núcleos de CPU e para as múltiplas GPUs. A percentagem de carga de trabalho fornecida a cada sistema é uma entrada para o algoritmo fornecido pelo usuário, sendo proporcional à potência de computação calculada previamente. Uma vez dada a percentagem desejada de computação, o tamanho dos dados é calculado antes de se chamar o algoritmo. As funções *matrixGPU* e *matrixCPU* são responsáveis pelo processamento do montante parcial destinado a cada recurso computacional.

2.3.3.2. Exemplo básico: MPI + CUDA

MPI é uma *API* padrão para comunicação de dados através de mensagens entre processos que é comumente usado em Computação de Alto Desempenho para construir aplicações que podem ser dimensionados para multi-nós. Como tal, MPI é compatível com CUDA, que é projetado para computação paralela em um único computador ou nó. Há uma série de razões para querer combinar esses paradigmas de programação paralela complementares: MPI + CUDA. E dentre elas podemos destacar:

- A resolução de problemas com um tamanho de dados muito grande para caber na memória de uma única GPU.
- A resolução de problemas que exigiriam tempo de computação excessivamente longo em um único nó.
- A aceleração de uma aplicação MPI existente com múltiplas GPU.

Para o exemplo a seguir, considera-se um sistema heterogêneo composto por múltiplos processadores em uma GPU. A ideia consiste em inicializar vários processos MPI permitindo que o particionamento da carga de trabalho de forma proporcional ao número de processos inicializados. O **Algoritmo 5** mostra o trecho de código que inicializa os processos MPI:

Algoritmo 5 Trecho de código híbrido: MPI + CUDA

```
1: #include <mpi.h>
2:
3: void run_kernel();
4:   int main(int argc, char .argv[]){
5:
6:    int rank, size;
7:
8:    MPI_Init (&argc, &argv);
9:    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
10:  MPI_Comm_size (MPI_COMM_WORLD, &size);
11:
12:  run_kernel(rank);
13:
14:  MPI_Finalize();
15:
16:  return 0;
17: }
```

O **Algoritmo 6** mostra o *kernel* utilizado por cada processo MPI para calcular o cálculo de soma de dois vetores. Para este exemplo temos que um único host pode controlar múltiplas GPU, que são conectados diretamente através do *bus* PCI, sendo que cada processo inicializado pode controlar apenas uma GPU de cada vez.

Algoritmo 6 Trecho de código híbrido: MPI + CUDA

```
1: #include <stdio.h>
2:  __global__ void kernel(int *array1, int *array2, int *array3){
3:   int index = blockIdx.x * blockDim.x + threadIdx.x;
4:       array3[index] =  array1[index] + array2[index];
5:  }
6:
7:    extern "C" void run_kernel(int node_id){
8:
9:     int i, array1[6], array2[6], array3[6]; int *devarray1,
10:       *devarray2, *devarray3;
11:
12:    for(i = 0; i < 6; i++)
13:      array1[i] = i; array2[i] = 3-i;
14:
15:      cudaMalloc((void**) &devarray1, sizeof(int)*6);
16:      cudaMalloc((void**) &devarray2, sizeof(int)*6);
17:      cudaMalloc((void**) &devarray3, sizeof(int)*6);
18:      cudaMemcpy(devarray1, array1, sizeof(int)*6, cudaMemcpyHostToDevice);
19:      cudaMemcpy(devarray2, array2, sizeof(int)*6, cudaMemcpyHostToDevice);
20:
21:       kernel<<<2, 3>>>(devarray1, devarray2, devarray3);
22:       cudaMemcpy(array3, devarray3, sizeof(int).6, cudaMemcpyDeviceToHost);
23:       for(i = 0; i < 6; i++)
24:          printf("Node %i - Suma[%i]=%d\n",node_id,i,array3[i]);
25:       cudaFree(devarray1);
26:       cudaFree(devarray2);
27:       cudaFree(devarray3);
28:  }
```

2.3.4. Programação Paralela utilizando Co-processadores

O desenvolvimento das arquiteturas dos processadores realizada pela Intel fez com que surgisse um novo formato para a Programação de Alto Desempenho para uma arquitetura com múltiplos núcleos. O aprimoramento de técnicas de vetorização de código tornou-se essencial para se obter um alto desempenho nestas arquiteturas, as quais vem oferecendo cada vez mais unidades de processamento vetoriais e com maior capacidade. Nesse sentido, é proposto nesta sub-seção mostrar os conceitos relacionados com vetorização e técnicas de acesso eficiente a memória, bem como, ferramentas para explorar tais técnicas, usando os recursos oferecidos pela arquitetura de co-processadores.

2.3.4.1. Exemplo básico: execução nativa

Existem várias formas de ver o modelo de programação em co-processadores. Uma dessas formas é trata-lo como um computador com múltiplos núcleos de forma independente. Isto é, uma vez que o executável é construído em seu sistema *host*, o

mesmo precisa ser copiado para o ’sistema de arquivos’ MIC no co-processador juntamente com quaisquer outros executáveis, bibliotecas e dados que ele necessite. O programa é então executado a partir do console via *ssh*. O código a seguir representa uma simples multiplicação de matrizes, de forma não vetorizada, ou seja, não consigo escalar várias operações por instrução para este problema.

No **Algoritmo 7** a diretiva *#pragma novector* desabilita o processo de vetorização. O resultado de desempenho para este exemplo é baixo, já que a arquitetura com co-processadores tem uma baixa eficiência para códigos sequenciais.

Algoritmo 7 Código ’native.cpp’ que representa o modelo de execução de forma nativa de uma aplicação utilizando co-processadores de forma não vetorizada.

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <malloc.h>
4: #include <vector>
5: using namespace std;
6: void mmul(float *a, int lda, float *b, int ldb, float *c, int ldc, int n){
7:
8:  for (int j = 0; j < n; ++j)
9:    for (int k = 0; k <n; ++k)
10:   for (int i = 0; i < n; ++i)
11:       c[j * ldc + i] += a[k * lda + i] * b[j * ldb + k];
12:
13: }
14: int main(int argc, char **argv){
15:
16:  int sz_align = 64;
17:  int N = atoi(argv[1]);
18:  int matrix_elements = N * N;
19:  int matrix_bytes = sizeof(float) * matrix_elements;
20:  float A = (float *)_mm_malloc(matrix_bytes,sz_align);
21:  float B = (float *)_mm_malloc(matrix_bytes,sz_align);
22:  float C = (float *)_mm_malloc(matrix_bytes,sz_align);
23:
24: #pragma novector
25: for(int i = 0; i < matrix_elements; i++) A[i] = 1.0; B[i] = 2.0; C[i] = 0.0;
26:
27: for(int i = 0; i < 1; i++){ #pragma novector
28:   for (int i = 0; i < matrix_elements; i++) C[i] = 0.0;
29:     mmul(A,N,B,N,C,N,N);
30: }
31: _mm_free(A); _mm_free(B); _mm_free(C);
32:
33: return 0;
34: }
```

2.3.4.2. Compilação e execução

O algoritmo anterior pode ser compilado e executado, seguindo os seguintes passos:

1. Compilar o programa usando *’-mmic’* para gerar o executável no MIC.
2. Agora faça o *login* no co-processador a partir do seu nó usando: *’ssh mic0’*.

3. E por fim, tentar executar o binário emitindo o comando: *'./binario 1024'*.

Algoritmo 8 Código 'vectorization.cpp' que representa o modelo de execução de forma nativa de uma aplicação utilizando co-processadores de forma vetorizada.

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <malloc.h>
4: #include <stdint.h>
5: #include <vector>
6: #include "offload.h"
7: using namespace std;
8:   __declspec(target(mic)) float .A = NULL;
9:   __declspec(target(MIC)) float .B = NULL;
10: __declspec(target(MIC)) float .C = NULL;
11:
12: void mmul(float .a, int lda, float .b, int ldb, float .c, int ldc, int n){
13:    for (int j = 0; j < n; ++j)
14:       for (int k = 0; k < n; ++k)
15: for (int i = 0; i < n; ++i)
16: c[j . ldc + i] += a[k . lda + i] . b[j . ldb + k];
17:
18: }
19: int main(int argc, char ..argv){
20:
21: int N = atoi(argv[1]);
22: int matrix_elements = N . N;
23: int matrix_bytes = sizeof(float) . matrix_elements;
24: A = (float .)malloc(matrix_bytes);
25: B = (float .)malloc(matrix_bytes);
26: C = (float .)malloc(matrix_bytes);
27:
28: #pragma novector
29: for (int i = 0; i < matrix_elements; i++)
30:    A[i] = 1.0; B[i] = 2.0; C[i] = 0.0;
31:
32: #pragma omp target
33: #pragma omp parallel
34: for(int i = 0; i<1; i++){
35:  #pragma novector
36:  for (int i = 0; i < matrix_elements; i++)
37:   C[i] = 0.0;
38:   mmul(A,N,B,N,C,N,N);
39:  }
40:  free(A); free(B); free(C);
41:  return 0;
42: }
```

2.3.4.3. Exemplo básico: vetorização

Tendo o conceito de vetorização como a capacidade de realizar uma operação matemática em 2 ou mais elementos ao mesmo tempo em várias estruturas matriciais, podemos exemplificar este conceito através do algoritmo a seguir. O **Algoritmo 8** através das diretivas *#pragma target* e *#pragma omp parallel* consegue escalar várias operações por instrução para a multiplicação de matrizes executada. A partir do incremento do tamanho do problema, consegue-se perceber o aumento de desempenho para as operações lógico aritméticas realizadas.

2.4. Considerações Finais

O texto mostrou que, dentre as principais ferramentas de desenvolvimento para obtenção de máximo desempenho de sistemas computacionais, percebe-se que os aplicativos sobre arquiteturas com múltiplos núcleos possuem uma enorme potencialidade. Características tais como classificação de tarefas, tempo de resposta e distribuição da carga de trabalho conduzem ao desenvolvimento de aplicativos computacionais eficientes. Apesar de ser um pouco complexo, usuários não expertos, programarem nestas arquiteturas, consegue-se mostrar uma métrica e metodologia de construção de algoritmos paralelos, de uma forma simples e prática.

Referências

[1] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. To-mov. Faster, cheaper, better - hybridization methodology to develop linear algebra software for GPUs. NVIDIA, 2, July 2010.

[2] F. Almeida, D. Giménez, J. Mantas, and A. Vidal. Introducción a la Programación Paralela. Paraninfo Cengage Learning, España, Madrid, 2008.

[3] Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra. Distributed and cloud computing: from parallel processing to the internet of things. 2012.

[4] Khronos Opencl and Aaftab Munshi. The OpenCL Specification Version: 1.0 Document Revision: 48. Available in: http://www.lammpi.org, 2013.

[5] Deborah T Marry, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. Intel Technology Journal, 6(1):1–12, 2002.

[6] MPI. Message Passing Interface. Available in: http://www.mpiforum.org, 2010.

[7] Bradford Nichols, Dick Buttlar, and Jacqueline Proul Farrel. Pthreads programming: A POSIX Standard for Better Multiprocessing. O'Reilly, 1996.

[8] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. Queue, 6:40–53, March 2008.

[9] John Nickolls and William J. Dally. The GPU computing era. IEEE Micro, 30(2):56–69, 2010.

[10] OpenMP. Architecture Review Board: OpenMP specifications. Available in: http://www.openmp.org/specs, 2010.

[11] Charles Severance and Kevin Dowd. High Performance Computing. O'Reilly Me-dia, 2 edition, 1998.

[12] TBB. Threading Building Blocks. Available in: http://www.threadingbuildingblocks.org, 2010.

[13] Barry Wilkinson and Michael Allen. Parallel programming - techniques and applications using networked workstations and parallel computers (2. ed.). Pearson Education, 2005.

Capítulo

3

Desenvolvimento de Aplicações Paralelas de Alto Desempenho com Python

Guilherme Esmeraldo
Laboratório de Sistemas Embarcados e Distribuídos – LEDS
Instituto Federal de Educação, Ciência e Tecnologia do Ceará – IFCE campus Crato

Resumo

Na literatura, para o desenvolvimento de aplicações de alto desempenho, tem-se utilizado os modelos de programação paralela MPI, OpenMP, CUDA e OpenCL. Esses modelos são baseados em C/C++, que são linguagens de alto desempenho, porém de difícil aprendizado e domínio. Este minicurso apresenta Python como uma linguagem de programação alternativa em projetos de aplicações paralelas de alto desempenho. Python oferece, além de uma linguagem com sintaxe bastante simplificada, diferentes recursos para análise e otimização de desempenho, bem como suporte aos modelos de programação paralela citados.

Abstract

In literature, researchers have used the parallel programming models MPI, OpenMP, CUDA and OpenCL for developing high-performance applications. These models are based on C/C++, which are high-performance languages, but on the other hand are difficult to learn and master. This short course presents Python as an alternative programming language in high performance parallel applications projects. Python provides, in addition to a language with very simplified syntax, different resources for analysis and performance optimization, as well as support to the mentioned parallel programming models.

3.1. Introdução

Ao longo dos anos, a literatura de programação paralela tem focado nos modelos de programação MPI, OpenMP, CUDA e OpenCL [DIAZ, MUNOZ-CARO and NINO 2012]. Esses modelos são codificados em C/C++, as quais são consideradas as linguagens mais eficientes e com maior desempenho [NANZ and FURIA 2015]. Contudo, são linguagens de difícil aprendizado, por incluir uma semântica indefinida e sintaxe confusa, e de difícil depuração [DIETZ, REGEHR and ADVE 2015], tornando o projeto de novas aplicações paralelas praticável apenas por programadores experientes.

Dentre as linguagens de programação, Python [PYTHON 2016] tem sido bastante utilizada em diferentes tipos de aplicações, como em projetos comerciais e acadêmicos, e, frequentemente, está situada entre as mais populares do mundo [TIOBE 2016].

No meio científico, Python vem se destacando ao longo dos anos por incluir: (i) linguagem com sintaxe simples, (ii) integração entre simulação e visualização, (iii) execução interativa de comandos, (iv) funções eficientes para manipulação de arrays, (v) operações numéricas de alto desempenho, (vi) suporte de inúmeras bibliotecas numéricas, (vii) documentação útil e abundante e grande suporte da comunidade Python, (viii) possibilidade de estender a linguagem com código compilado, (ix) habilidade de interação com outros softwares, e (x) suporte à computação paralela e distribuída. [CAI, LANGTANGEN and MOE 2005] [OLIPHANT 2007] [DALCIN et al. 2011]. Nesse cerne, seu sucesso deve-se então a todo um ecossistema – linguagem, bibliotecas, integração com ferramentas de terceiros, e, principalmente, comunidade e cultura científica –, o qual traz um alto potencial para pesquisas em diversos campos do conhecimento, como na Matemática, na Física e na Biologia [LANGTANGEN 2012].

Nesse sentido, este minicurso apresenta Python como um ambiente completo e propício ao desenvolvimento de novas aplicações paralelas de alto desempenho. Os objetivos gerais incluem o uso da linguagem para desmistificar a programação de novas aplicações de alto desempenho, tornando-a acessível a todos. Este minicurso está dividido da seguinte maneira:

- **Seção 3.2** – Introdução à Linguagem: Esta seção apresenta brevemente a linguagem Python, focando na simplicidade da sintaxe e em recursos avançados nativos.

- **Seção 3.3** – Otimizando o Desempenho: Esta seção apresenta técnicas e ferramentas complementares para análise e otimização do desempenho da linguagem.
- **Seção 3.4** – Computação Científica: Nesta seção, serão apresentadas as principais bibliotecas utilizadas na computação científica, enfatizadas as mais utilizadas na computação de alto desempenho.
- **Seção 3.5** – Programação Paralela em Memória Compartilhada: Esta seção inclui o primeiro contato com a programação paralela. Serão apresentados diferentes modelos para implementação de processos e threads, como a systemcall “fork”, os módulos “threading” e “multiprocessing” e OpenMP.
- **Seção 3.6** – Programação Paralela em Memória Distribuída: O objetivo desta seção é apresentar o suporte Python ao modelo de programação distribuída MPI.
- **Seção 3.7** – Programação de GPUs: Esta seção apresenta os conceitos básicos para programação de GPUs em Python, enfatizado o uso de OpenCL.
- **Seção 3.8** – Pesquisas e Conferências: Por fim, serão apresentados as principais conferências científicas e tendências de projetos relacionadas à linguagem. Nesse sentido, busca-se abrir espaço para discussões acerca do tema e de novos trabalhos.

3.2. Introdução à Linguagem

Criada por Guido van Rossum, a primeira versão da linguagem Python surgiu no início da década de 1990 [HETLAND 2006]. Python é uma linguagem de programação de alto nível e, como as linguagens mais modernas, inclui recursos como: suporte nativo a diferentes tipos de dados (primitivos e estruturados), suporte aos paradigmas de programação imperativo, funcional e orientado a objetos, inclui mecanismos para tratamento de exceções, programação concorrente, gerenciamento de memória, programação em redes, programação de interfaces com o usuário (e.g. linha de comando, GUI e Web), acesso a bancos de dados, entre outros. Estima-se que há suporte de mais de 100 bibliotecas padrão da linguagem, com funções em diferentes domínios [BANDYOPADHYAY 2009], e mais de 85 mil complementos (bibliotecas de terceiros) [PYPI 2016].

De acordo com Chun (2001), a linguagem em si tem como características a facilidade de aprendizado e de leitura, pois busca evitar a utilização de elementos textuais que podem prejudicar a compreensão sintática. O autor cita que, em determinadas linguagens, é necessário o uso de símbolos especiais para, por exemplo, diferenciar estruturas de dados – em determinadas linguagens, por exemplo, o identificador de uma variável deve ser precedido por um "$" e o de um array por "@" – e para definição de um bloco de código (e.g. "{" e "}"), e que podem prejudicar o aprendizado de uma linguagem.

Além disso, a linguagem Python evita flexibilizar o estilo de codificação, para evitar códigos difíceis de compreensão (*"obfuscated code"*), em função de codificação padronizada e compreensão rápida de programas de terceiros. A Figura 3.1 traz um exemplo de código Python, no qual é possível realizar uma breve explanação de alguns recursos da linguagem.

```
1.  from math import pi
2.
3.  def area(raio):
4.          a = pi*(raio**2)
5.          return a
6.
7.  print area(10)
```

Figura 3.1 - Exemplo de código em Python.
Fonte: próprio autor (2016).

O código da figura acima ilustra uma forma de calcular, em Python, a área de um círculo, a partir do seu raio. A Linha 1 mostra um exemplo de como Python implementa o conceito de *namespace*. Nela, importa-se, para o contexto de trabalho, o elemento chamado "pi", que faz parte no módulo "math" (no jargão Python, as bibliotecas de código são referenciadas como 'módulos'). Nas linhas 3-5, é definida a função "area", a qual realiza o cálculo da área do círculo. A definição de funções em Python deve ser iniciada pela construção "def", seguida por nome da função, parâmetros entre parênteses, o operador de início de bloco de código ":" e o bloco de código que implementa o comportamento (corpo) da função. Um bloco de código em Python é caracterizado por um conjunto de instruções com o mesmo espaçamento em relação à margem. Na função "area" acima, as instruções nas Linhas 4-5 implementam o corpo da função e, desta forma, devem necessariamente possuir o mesmo espaçamento em

relação à margem. A instrução na Linha 7 realiza o cálculo da área de um círculo de raio 10, com resultado impresso na saída padrão do terminal.

Python implementa o conceito de *Arbitrary-Precision Arithmetic – APS* [Ghazi 2010], o qual, em termos gerais, permite estender a precisão de representação dos elementos numéricos da linguagem (na realidade, o limite de representação é dado pela quantidade de memória disponível para alocação de dados). Por exemplo, em C/C++, em uma arquitetura de 64-bits, não seria possível realizar a operação $(2^{63} – 1) + 1$, pois como $(2^{63} – 1)$ é o maior número inteiro que poderia ser representado nessa arquitetura, a operação de soma causaria um erro de *overflow*. Em Python, com o suporte de APS, seria perfeitamente possível, por exemplo, para calcular a área de um círculo de raio 2^{150} (número pertencente ao intervalo que extrapolaria o limite de 64-bits). O suporte de APS aos números inteiros é nativo da linguagem e, para números ponto flutuante, utiliza-se a biblioteca“mpmath” [MILLMAN and AIVAZIS 2011].

Outra característica importante da linguagem Python é a sua natureza dinâmica [BANDYOPADHYAY 2009]. Python trata a maioria de seus elementos – números, coleções, funções, classes, entre outros – como objetos em memória [ZHANG 2015]. Dessa forma, pode-se criar funções genéricas que podem receber parâmetros e tratar diferentes tipos de dados. A função “area” (na Figura 3.1), por exemplo, é uma função genérica e, sem modificações, poderia calcular também a área de círculos com raios 10,1 (ponto flutuante) ou 1 + 1i (número complexo). Outros recursos dinâmicos incluem a adição “*on the fly”* de novos atributos à objetos, bem como a geração e execução de código dinamicamente [HOLKNER and HARLAND 2009].

3.2.1. Coleções

Coleções são estruturas de dados que permitem agrupar objetos Python [MODEL 2009]. Não existe limitação de quantidade e de tipagem dos objetos agrupados em coleções. Algumas coleções permitem agrupar objetos com tipagem diferente ou coleções de objetos. Coleções Python, além de seguirem o padrão da linguagem (disponibilizam uma interface simples para criação e manipulação), elas apresentam alto desempenho em suas operações [PEREZ, GRANGER and HUNTER 2011]. As coleções nativas da linguagem Python consistem de:

- **Conjunto** (**Set**) agrupa desordenadamente e sem repetição dados imutáveis [MODEL 2009]. Dados imutáveis são objetos que não mudam de valor (em

Python, os dados imutáveis são os números inteiros, ponto flutuantes e complexos, bem como as *strings* e tuplas). Conjuntos são utilizados comumente na remoção de elementos duplicados em coleções de objetos ou na verificação de relações ou realização de operações matemáticas sobre conjuntos, como as de "pertinência", "interseção", "união" e "complemento".

- **Sequência** (**Sequence**) agrupa objetos sequencialmente, dispondo de um índice para que possam ser acessados individualmente [CHUN 2001]. As sequências podem ser do tipo mutável (listas) e imutável (tuplas e *strings*). Há um conjunto de operações que podem ser realizadas em todos os tipos de sequência, que são: indexação, *slice* (ou partição), repetição, concatenação e pertinência. A Figura 3.2 ilustra o uso dessas operações sobre uma sequência do tipo lista. Na Linha 1, é criada uma lista (denotada pelos símbolos "[" e "]") com dez caracteres, de "a" à "j". A Linha 2 inclui exemplos de operações de indexação, onde são indexados o primeiro elemento, o quinto, o último e o penúltimo (índices "[0]", "[4]", "[-1]" e "[-2]", respectivamente). Na Linha 3, são realizadas operações de *slice* para selecionar os intervalos: do primeiro ao quarto elemento (os *slices* "[0:4]" e "[:4]" são equivalentes), do quinto ao último elemento (*slice* "[4:]") e do quinto ao oitavo elemento (*slice* "[4:8]"). Na Linha 4, realiza-se as operações de repetição e concatenação, as quais, neste caso, trazem resultados equivalentes (cria-se uma nova lista com a lista anterior duplicada). Nas Linhas 5 e 6, verifica-se a pertinência dos elementos "a" e "z" em "lista". Python oferece ainda um recurso conhecido como *list comprehension*, utilizado para aplicar operações ou filtrar os elementos de uma lista. *List comprehension* consiste de uma notação, que assemelha-se a denotação por compreensão, utilizada na matemática para representação de conjuntos a partir de propriedades. Por exemplo, as *list comprehensions* nas Linhas 7-8 são equivalentes às notações matemáticas "{upper(x) | x ϵ lista}" e "{upper(x) | x ϵ lista e x > 'f'}", respectivamente. Lutz (2013) cita que *list comprehension* é um recurso muito eficiente, com desempenho equiparável ao da linguagem C.

```
1.  lista = ['a','b','c','d','e','f','g','h','i','j']
2.  print lista[0], lista[4], lista[-1], lista[-2]
3.  print lista[0:4], lista[:4], lista[4:], lista[4:8]
4.  print lista*2, lista+lista
```

```
5. print 'a' in lista, 'z' in lista
6. print 'a' not in lista, 'z' not in lista
7. print [x.upper() for x in lista]
8. print [x.upper() for x in lista if x > 'f']
```

Figura 3.2 - Operações sobre sequências Python.
Fonte: próprio autor (2016).

- **Dicionários** (**Mappings**), conhecidos como *hashings*, em outras linguagens de programação, agrupam dados desordenados em formato de chave/valor. A chave consiste de um identificador que é utilizado para referenciar um determinado valor. Uma chave pode ser de qualquer tipo de objeto imutável e deve ser única dentro de um dicionário, pois é desta forma que o valor é referenciado (não seria possível, por exemplo, referenciar dois ou mais valores diferentes a partir de uma determinada chave).
- **Streams** são sequenciais de dados ordenados e possuem tamanhos indefinidos. Arquivos de texto, conexões de rede e geradores são exemplos *streams* Python. Um gerador é um tipo de objeto Python que gera sequências de valores a partir de algum tipo de processamento. Geradores são bastante versáteis e podem ser utilizados em diferentes aplicações, como na geração de sequências de números primos [LUSZCZEK et al. 2016] ou dos sinais de comunicação de modelos de hardware em software [DECALUWE 2004]. A Figura 3.3 apresenta um exemplo de gerador para sequência de Fibonacci. A definição do gerador é realizada nas instruções das Linhas 1-5, e a instrução da Linha 7 mostra a criação de um objeto gerador. Por fim, as Linhas 8-9, mostram a geração dos dez primeiros números da sequência de Fibonacci. Na definição do gerador da Figura 3.3, a declaração “yield” (Linha 5) retorna o valor de “x” e bloqueia o gerador e, quando o método “next” é invocado (Linha 9), o gerador é desbloqueado continuando sua execução a partir da instrução posterior à “yield”.

```
1. def fibonacci():
2.     x, y = 0, 1
3.     while 1:
4.         x, y = y, x + y
5.         yield x
6.
```

```
7.  fib = fibonacci()
8.  for i in range(10):
9.      print fib.next()
```

Figura 3.3 - Gerador para a sequência de Fibonacci.
Fonte: próprio autor (2016).

3.2.2. O Ambiente de Execução

Python é uma linguagem interpretada. Para que um programa seja executado, o interpretador Python compila o código em uma representação intermediária (*bytecodes*), a qual será executada pela *Python Virtual Machine* (PVM). A PVM lê sequencialmente as instruções em *bytecodes*, interpretando os elementos dinâmicos (e.g. os tipos dos dados contidos nas variáveis), convertendo-as em código de máquina para execução na CPU. O fato de utilizar uma representação intermediária do código, faz com que o desempenho de Python seja maior em relação às linguagens puramente interpretadas [LUTZ 2013], pois podem ser realizadas otimizações de código durante geração da representação intermediária, além de torná-la portável para diferentes sistemas operacionais [ROGHULT 2016].

O interpretador Python dispõe de um modelo de execução interativo, onde os ambientes de desenvolvimento e execução estão integrados. Desta forma, na medida que um programa é construído, é possível testar cada uma de suas instruções. Este recurso é interessante pois, com a ausência de ciclos de compilação antes da execução de um programa, a prototipação de uma nova aplicação pode tornar-se mais eficiente [GUPTA 2003] e o projeto mais produtivo [OLIPHANT 2007].

O ambiente de execução Python possui ainda as seguintes características [LUTZ 2006]:

- Gerenciamento automático de memória, via *Garbage Collector*;
- Documentação online, utilizando as funções “dir” e “help” é possível listar os atributos de módulos e a documentação das funções e classes;
- *Traceback* de erros, para suporte na depuração de programas. Para uma análise mais aprofundada, pode-se utilizar, em conjunto com o interpretador, bibliotecas de análise (estática) de código, como PyChecker, pylint e PyFlakes [MARTELLI 2006];
- Acesso às informações do interpretador, oferecendo nativamente serviços de metraprogramação e de introspecção de objetos Python;

- Carga dinâmica de módulos, para que programas Python possam ser modificados em tempos de projeto/execução/teste;
- Integração Python/C, permite que programas C tenham acesso ao interpretador Python. Como esse recurso, pode-se delegar partes de um programa C a código Python,
- Portabilidade, disponível para várias plataformas operacionais. Entres elas: Windows, MS-DOS, Windows CE, Linux, MacOS, HP-UX, Solaris, BeOS, MorphOS, entre outras,
- Código aberto.

3.2.3. Instalando Complementos

Python conta com um repositório aberto de complementos, chamado de PyPI (*Python Package Index*) [PYPI 2016]. Esse repositório conta, atualmente, com mais de 85 mil pacotes. No PyPI, além dos códigos fonte, nos metadados de cada pacote, há informações sobre eles, como descrições, dependências de pacotes e versões.

Existem diversas abordagens para instalação dos pacotes a partir do PyPI. A ferramenta ‘pip’ é a mais utilizada, por incluir uma interface em linha de comando bastante simples e com diversas opções para gerenciamento de complementos. Seguem seus principais comandos:

- “pip install <pacote>”, instala o complemento com nome <pacote>;
- “pip uninstall <pacote>”, remove o complemento com nome <pacote>, caso instalado;

“pip list”, lista todos os complementos instalados;

- “pip show <pacote>”, mostra informações sobre o complemento com nome <pacote>;
- “pip search <padrao>”, pesquisa no repositório PyPI por algum complemento que contenha o padrão <padrao> em seu nome.

3.3. Otimizando o Desempenho

No desenvolvimento de novas aplicações de alto desempenho, recomenda-se prototipar partes da aplicação, para compreender melhor o sistema como um todo e realizar

melhores decisões de projeto, para só então tratar os aspectos de otimização [KNUTH 1992] [LANARO 2013].

A tarefa de otimização de desempenho envolve várias etapas, as quais, segundo Rane e Browne (2011), podem ser divididas em: análise de desempenho (*measurement*), diagnóstico (*profiling* e *tracing*), seleção e implementação da otimizações (*tuning*). As subseções a seguir apresentam técnicas para suporte à otimização de desempenho em aplicações Python.

3.3.1. Análise de Desempenho

A análise de desempenho é um passo importante no desenvolvimento e otimização de aplicações de alto desempenho [KNÜPFER et al. 2008] e consiste em avaliar o desempenho de uma aplicação sob uma determinada métrica, como tempo de execução ou memória consumida.

Para a análise de desempenho de uma aplicação Python, pode-se utilizar o módulo “timeit” da biblioteca padrão da linguagem. Este modulo contém funções utilizadas para computar o tempo consumido por determinados segmentos de código durante a execução de uma aplicação.

O código da Figura 3.4 apresenta um exemplo de análise de desempenho de duas funções (uma interativa e outra recursiva) para geração de um número da sequência de Fibonacci. Nesse exemplo, utiliza-se as funções “timeit” e “repeat” para calcular o tempo necessário para que as funções computem o 30º número da sequência. A função “repeat”, na realidade, é um *wrapper* para a função “timeit”, ela é configurada para executar a função “timeit” *k* vezes, de forma que, ao final de sua execução, será gerada uma lista com os *k* tempos medidos. No exemplo da Figura 3.4, “repeat” é configurada para executar três vezes a função “timeit”.

```
1.  def fibL(n):
2.      a,b = 1,1
3.      for i in range(n-1):
4.          a,b = b,a+b
5.      return a
6.
7.  def fibR(n):
8.      if n==1 or n==2:
```

```
9.          return 1
10.     return fibR(n-1) + fibR(n-2)
11.
12. from timeit import timeit, repeat
13. from numpy import mean
14.
15. print "Fibonacci iterativo\n",30*"="
16. setup="from __main__ import fibL"
17.
18. print timeit("fibL(30)", setup, number=1)
19. print repeat("fibL(30)", setup, repeat=3, number=1)
20. print mean(repeat("fibL(30)", setup, repeat=3, number=1))
21.
22. print "\nFibonacci recursivo\n",30*"="
23. setup="from __main__ import fibR"
24.
25. print timeit("fibR(30)", setup, number=1)
26. print repeat("fibR(30)", setup, repeat=3, number=1)
27. print mean(repeat("fibR(30)", setup, repeat=3, number=1))
```

Figura 3.4 - Medindo o desempenho para geração interativa e recursiva de numero da sequência de Fibonacci.
Fonte: próprio autor (2016).

No código da Figura 3.4, as Linhas 1-5 e 7-10 incluem as definições das funções interativa e recursiva para geração de um número na sequência de Fibonacci. As Linhas 18-20 realizam, na sequência: a medida do tempo dado para computar o 30º número (função “timeit”), gerar uma lista com os tempos medidos em três computações do 30º número (função “repeat”) e o cálculo da média (função “mean”) de mais três medidas de tempo de novas computações, para a versão interativa da função de Fibonacci. Já as Linhas 25-27 realizam os mesmos procedimentos das Linhas 18-20, porém para a versão recursiva da função. Os resultados das medidas de tempo podem ser vistos na Figura 3.5.

```
1. Fibonacci iterativo
2. ==============================
3. 1.31130218506e-05
4. [5.9604644775390625e-06, 4.0531158447265625e-06, 4.0531158447265625e-06]
5. 3.89417012533e-06
6.
7. Fibonacci recursivo
```

```
8.  ============================
9.  0.449427843094
10. [0.5111749172210693, 0.5202469825744629, 0.5077059268951416]
11  0.43682106336
```

Figura 3.5 - Medida de desempenho para geração recursiva e interativa do 30o. número da sequência de Fibonacci.
Fonte: próprio autor (2016).

Os resultados apresentados na Figura 3.5 mostram que, comparando os tempos medidos para as duas funções, dentre elas, a com versão recursiva tem desempenho bastante inferior, apresentando maiores tempos em relação aos da versão iterativa.

3.3.2. Diagnóstico

Após a análise de desempenho, o passo seguinte consiste em identificar, no código, as seções que necessitam de algum tipo de otimização (*bottlenecks*).

Para o diagnóstico, comumente utiliza-se as técnicas de *profiling* e de *tracing*. Ambas são aplicadas, durante a execução do programa, para coletar informações sobre determinados eventos (os eventos capturados podem variar desde chamadas de funções até operações de comunicação entre processos) [DOGLIO 2015]. Em *profiling*, as informações obtidas sobre todos os eventos são apresentadas de forma sumarizada, enquanto que em *tracing*, são listadas informações sobre cada um dos eventos [KNÜPFER et al. 2008]. Dependendo do nível de informações e da quantidade de eventos capturados, utilizar *tracing* pode comprometer o desempenho da aplicação e resultar em *traces* bastante longos, tornando-se bastante custoso para analisá-los por ocuparem grandes espaços em memória ou em disco [LARUS 1990]. Da mesma forma, enquanto *profiling* é mais escalável, as informações apresentadas por esta técnica podem ser muito limitadas para se obter um diagnóstico completo e correto sobre problemas de desempenho de uma aplicação [MOHROR and KARAVANIC 2009].

Em Python, há diferentes mecanismos para *profiling*. Lanaro (2013) fez um comparativo entre três módulos de *profiling* da biblioteca padrão Python e concluiu que o “cProfile”, dentre os três, é o que adiciona o menor *overhead* na execução da aplicação. A Figura 3.6 mostra um código para geração recursiva do 30º número da sequência de Fibonacci com suporte de *profiling* via “cProfile”.

Na figura, as Linhas 1-4 contêm a definição da função recursiva, a Linha 6 importa o módulo “cProfile” para o contexto da aplicação, o qual é acionado, na Linha 7, para capturar um *profile* da geração do 30º número da sequência.

```
1.  def fibR(n):
2.      if n==1 or n==2:
3.          return 1
4.      return fibR(n-1) + fibR(n-2)
5.
6.  import cProfile
7.  cProfile.run("fibR(30)")
```

Figura 3.6 - *Profiling* da aplicação recursiva para geração de número da sequência de Fibonacci.
Fonte: próprio autor (2016).

O *profile* gerado pode ser visto na Figura 3.7, com as seguintes informações (da coluna da esquerda para direita): número de chamadas das funções (“ncalls”), o tempo total acumulado para a função (“tottime”), uma estimativa de tempo para cada chamada da função (“percall”), o tempo total acumulado para a função e respectivas subfunções (“cumtime”), o quociente de “cumtime” pelo número de chamadas da função (“percall”) e o registro da função chamada por “nome de arquivo”, “linha de código” e “nome da função”. Na figura, observa-se, por exemplo, que praticamente todo o tempo consumido pela aplicação está relacionado às múltiplas chamadas recursivas da função “fibR” (total de 1.664.079 chamadas).

```
1.          1664081 function calls (3 primitive calls) in 0.910 seconds
2.
3.     Ordered by: standard name
4.
5.     ncalls  tottime  percall  cumtime  percall filename:lineno(function)
6.  1664079/1  0.910    0.000    0.910    0.910   fibR.py:1(fibR)
7.          1  0.000    0.000    0.910    0.910   <string>:1(<module>)
8.          1  0.000    0.000    0.000    0.000   {method 'disable' of '_lsp
9.  rof.Profiler' objects}
```

Figura 3.7 - *Profile* da aplicação recursiva para geração de número da sequência de Fibonacci.
Fonte: próprio autor (2016).

Um *profile* gerado com o módulo “cProfile” pode ser gravado em arquivo através do comando de terminal “python -m cProfile -o fibR.cprof fibR.py”, onde “fibR.cprof” é o arquivo de *profile* gerado ao final da execução do programa “fibR.py”. Algumas ferramentas, como RunSnake, SnakeViz e KCacheGrind, utilizam o arquivo de *profile* para gerar interfaces gráficas para apoio à análise de desempenho. A figura a seguir mostra a interface do KCacheGrind para o *profile* “fibR.cprof”. Nela, pode-se ver diferentes informações, como uma representação em grafo das chamadas das funções da aplicação.

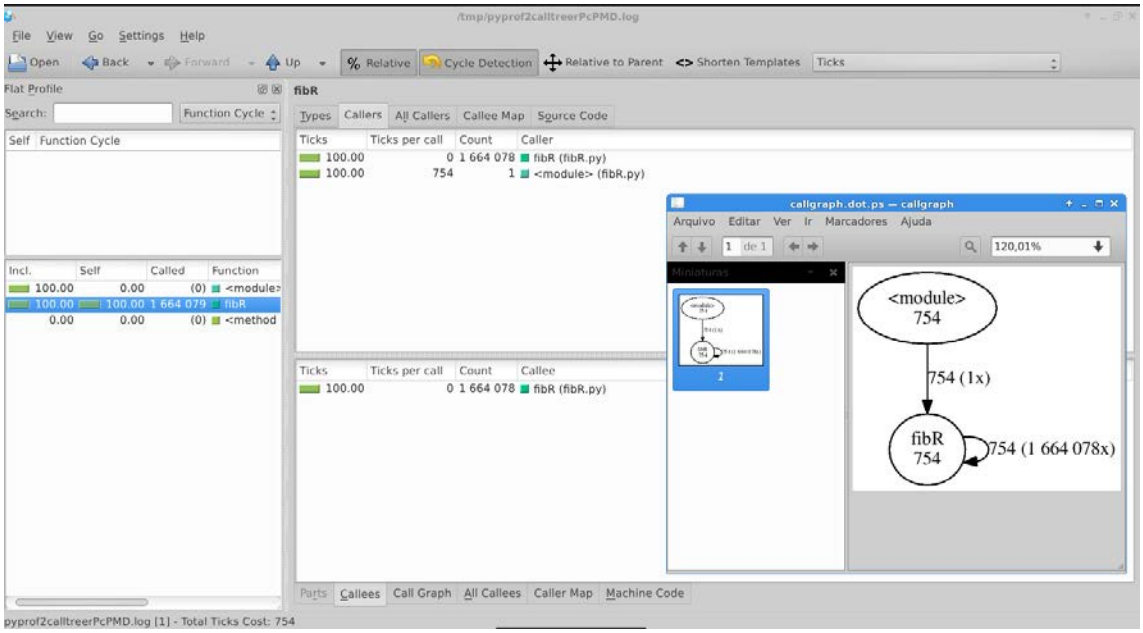


Figura 3.8 - KCacheGrind:Interface gráfica para análises de *Profile*.
Fonte: próprio autor (2016).

Estão ainda disponíveis ferramentas para *profiling* de execução das linhas de código, onde os módulos “line_profiler” e “memory_profiler” mostram estatísticas relacionadas ao tempo de execução e de consumo de memória de cada linha de código da aplicação, respectivamente. As Figura 3.9 e 3.10 mostram os *profiles* de tempo de execução e de consumo de memória, respectivamente, para a versão recursiva da aplicação de geração de número de sequência de Fibonacci.

1.	Wrote profile results to fibR.py.lprof
2.	Timer unit: 1e-06 s
3.	
4.	Total time: 3.6789 s

```
5.  File: fibR.py
6.  Function: fibR at line 2
7.
8.  Line #      Hits         Time  Per Hit   % Time  Line Contents
9.  ==============================================================
10.     2                                           @profile
11.     3                                           def fibR(n):
12.     4   1664079     1294864      0.8     35.2      if n==1 or n==2:
13.     5    832040      492323      0.6     13.4          return 1
14.     6    832039     1891716      2.3     51.4      return fibR(n-1) +
15. fibR(n-2)
```

Figura 3.9 - *Line_profiler* para a aplicação recursiva para geração de número da sequência de Fibonacci.
Fonte: próprio autor (2016).

```
1.  Filename: fibR.py
2.
3.  Line #    Mem usage    Increment   Line Contents
4.  ================================================
5.      2   16.602 MiB    0.000 MiB   @profile
6.      3                             def fibR(n):
7.      4   16.602 MiB    0.000 MiB       if n==1 or n==2:
8.      5   16.602 MiB    0.000 MiB           return 1
9.      6   16.602 MiB    0.000 MiB       return fibR(n-1) + fibR(n-2)
```

Figura 3.10 - *Memory_profiler* para a aplicação recursiva para geração de número da sequência de Fibonacci.
Fonte: próprio autor (2016).

A tarefa de *Tracing* é bastante simples em Python, quando utiliza-se o módulo "trace". Para captura do *trace* da aplicação recursiva de geração de um número da sequência de Fibonacci, basta executar o comando de terminal "python -m trace --trace fibR.py > fibR.trace", onde "fibR.py" é o programa que será analisado, e "fibR.trace" é o arquivo de *trace* gerado ao final da execução do programa. O *trace* gerado contém um registro para cada execução de todas as instruções, na ordem em que são chamadas, e ocupa, em disco, aproximadamente 172 MB.

3.3.3. Ajustando o Código

Após análise e diagnóstico de desempenho, o passo seguinte consiste em modificar o código e avaliar as alterações no desempenho da aplicação.

Lanaro (2013) afirma que deve-se observar a formulação e a sequência lógica das instruções de um programa Python, pois até simples operações de atribuição de dados mal formuladas, podem impactar no desempenho final da aplicação. Nesse caso, para análise e otimizações algorítmicas – como, por exemplo, a seleção de uma estrutura de dados ou a seleção de algum algoritmo de pesquisa/ordenação – recomenda-se aprofundamento teórico em algoritmos [CORMEN et al. 2009].

Na seção anterior, citou-se que, para execução de um programa Python, realiza-se a compilação das instruções para uma representação intermediária (*bytecodes*) para serem executadas pela PVM. Lanaro (2013) cita ainda que pode-se utilizar o módulo 'dis' (*disassembler*) para aprofundar a análise das instruções em nível de *bytecodes*. Essa análise pode direcionar a reestruturação do código Python para reduzir o número de *bytecodes* gerados, e consequentemente aumentar o desempenho da aplicação. O mesmo autor cita ainda algumas dicas para ajustar o código, como utilizar estruturas de dados nativas da linguagem, e dar preferência ao uso de geradores e de *list comprehensions* ao invés de *loops* (ver "Coleções" na Seção 3.2), bem como utilizar bibliotecas com código Python otimizado ou com código compilado (ver subseção a seguir).

Outras abordagens complementares podem ainda ser adotadas, como: utilizar interpretadores Python alternativos, como PyPy, Jython, Numba, Nuitka e Falcon [MURRI 2013][ROGHULT 2016], que implementam diferentes técnicas de otimização em código Python, como compilação *Just-in-Time* (JITing) [AICOCK 2003], o uso do compilador externo LLVM [LATTNET and ADVE 2004] e geração de *bytecodes* para execução na Java Virtual Machine (JVM) [HUNT and JOHN 2011]; ou aproveitar o potencial das novas plataformas de hardware e paralelizar o código da aplicação (neste minicurso, diferentes modelos de paralelização são tratados nas Seções 3.5, 3.6 e 3.7).

3.3.4. Utilizando Extensões Compiladas

O suporte de bibliotecas compiladas pode ser realizado de duas formas: 1) compilar a biblioteca partir de código Python; ou 2) criar um *wrapper* (envoltório) Python para uma biblioteca C pré-compilada.

Para a primeira abordagem, utiliza-se Cython [BEHNEL 2011]. Cython é uma linguagem que estende Python e possui um compilador que permite traduzir código Python para C (o qual, por sua vez, é compilado para compor uma biblioteca Python).

Para ilustrar o uso de Cython, será gerada uma biblioteca compilada a partir do código da função recursiva de geração de número da sequência de Fibonacci, (ver código da biblioteca na Figura 3.11).

```
1.  def fibR(n):
2.      if n==1 or n==2:
3.          return 1
4.      return fibR(n-1) + fibR(n-2)
```

Figura 3.11 - fibR.pyx: Biblioteca Python com função para geração recursiva de número da sequência de Fibonacci.
Fonte: próprio autor (2016).

O código da Figura 3.11 é gravada em um arquivo com extensão ".pyx" (no exemplo, o código da figura é gravado em um arquivo chamado "fibR.pyx"). Em seguida, cria-se um programa Python necessário para dar suporte à compilação da biblioteca (Figura 3.12 mostra o código do programa "setup.py").

```
1.  from distutils.core import setup
2.  from Cython.Build import cythonize
3.  setup(
4.      ext_modules=cythonize("fibR.pyx"),
5.  )
```

Figura 3.12 - setup.py: programa Python para automatização de compilação com Cython.
Fonte: próprio autor (2016).

Por fim, para realizar a compilação utiliza-se o comando de terminal "python setup.py build_ext --inplace". Ao final da compilação serão gerados os arquivos "fibR.c" e "fibR.o", que são o código em C gerado a partir de "fibR.pyx" e a biblioteca compilada.

O código da Figura 3.13 mostra como pode-se utilizar a função "fibR" da nova biblioteca compilada e, na Figura 3.14, apresenta-se o resultado da análise de desempenho dessa nova aplicação.

```
1.  import fibR
2.  print fibR.fibR(30)
```

Figura 3.13 - Programa python com biblioteca compilada para geração recursiva de número da sequência de Fibonacci.
Fonte: próprio autor (2016).

```
1.  0.231173038483
2.  [0.22117304801940918, 0.25551509857177734, 0.21577692031860352]
3.  0.246431986491
```

Figura 3.14 - Análise de desempenho da biblioteca compilada para geração recursiva de número da sequência de Fibonacci.
Fonte: próprio autor (2016).

Comparando-se os resultados desempenho da Figura 3.14 com os apresentados nas Linhas 9-11 da Figura 3.5 (que representa o desempenho da função recursiva em código Python), podemos observar que, ao utilizar a biblioteca compilada com Cython, houve aumento do desempenho do algoritmo recursivo.

A segunda abordagem, que consiste em criar um *wrapper*, é uma das mais utilizadas (parte do grande sucesso de Python é atribuído à capacidade da linguagem interfacear com bibliotecas compiladas existentes [BANDYOPADHYAY 2009]).

Um *wrapper* consiste de uma interface que permite a chamada de funções de uma biblioteca C pré-compilada a partir de um programa Python [SANNER 1999]. Para tanto, o *wrapper* tem acesso à biblioteca e realiza as devidas conversões entre objetos Python e dados em C. Existem diferentes abordagens e mecanismos para a criação de wrappers, como Pyrex e SWIG [BEHNEL et al. 2011], contudo, neste minicurso é abordado o método mais imediato de criação.

A Figura 3.15 mostra um exemplo do código C de um *wrapper* para uma biblioteca com função para geração recursiva de número da sequência de Fibonacci. No código, a função recursiva de geração de número da sequência de Fibonacci está implementada em C. Nas Linhas 3-8, observa-se o código da função recursiva (função "_fibR").

Já as Linhas 10-17 incluem o código do *wrapper*. Observe que, na Linha 10, a função "fibR_wrapper", que é a interface do *wrapper*, recebe como parâmetros e retorna objetos Python. Na Linha 13, a função "PyArg_ParseTuple" realiza a conversão do objeto Python (passado como parâmetro da função "fibR_wrapper" através de "args") para um dado numérico inteiro (utilizando a máscara "i") e armazena o resultado em "n". Na Linha 16, a função "Py_BuildValue" converte o número inteiro obtido com a execução da função "_fibR(n)" para um objeto Python, retornando-o para o programa Python que invoca a função "fibR" através da interface do módulo (*wrapper*).

```
1.  #include <Python.h>
2.
3.  int _fibR(int n){
4.      if ((n == 1) || (n == 2))
5.          return 1;
6.      else
7.          return _fibR(n-1) + _fibR(n-2);
8.  }
9.
10. static PyObject* fibR_wrapper(PyObject* self, PyObject* args){
11.     int n;
12.
13      if (!PyArg_ParseTuple(args, "i", &n))
14.         return NULL;
15.
16.     return Py_BuildValue("i", _fibR(n));
17. }
18.
19. static PyMethodDef FibMethods[] = {
20.     {"fibR", fibR_wrapper, METH_VARARGS, "Gets a Fibonacci number."},
21.     {NULL, NULL, 0, NULL}
22. };
23.
24. PyMODINIT_FUNC initfibR(void){
25.     (void) Py_InitModule("fibR", FibMethods);
26. }
```

Figura 3.15 - *Wrapper* para função compilada para geração recursiva de número da sequência de Fibonacci.
Fonte: próprio autor (2016).

Ainda na Figura 3.15, o código nas Linhas 19-22 realiza o registro da função “fibR_wrapper” como uma função de módulo Python chamada “fibR”. Por fim, as Linhas 24-26 consistem de um função para inicialização do módulo Python (alguns módulos podem necessitar realizar algum tipo de processamento antes de ser utilizado).

Para a geração do *wrapper*, é necessário criar o programa “setup.py”, da Figura 3.16, o qual automatiza a compilação, convertendo o *wrapper* para um módulo Python. A compilação é realizada com o comando de terminal “python setup.py build”.

```
1.  from distutils.core import setup, Extension
2.
3.  module1 = Extension('fibR', sources = ['fibR.c'])
4.
5.  setup (name = 'PackageName',
6.          version = '1.0',
7.          description = 'Fibonacci package example',
8.          ext_modules = [module1])
```

Figura 3.16 - setup.py: programa para compilação do *wrapper* como um módulo Python.
Fonte: próprio autor (2016).

Ao final da compilação, será gerado o arquivo “fibR.o”, que é o módulo referente ao *wrapper*, e pode ser utilizado com o mesmo programa da Figura 3.13. Os resultados da análise de desempenho do *wrapper* podem ser vistos na Figura 3.17. Comparando-se os novos resultados com os apresentados nas Linhas 9-11 da Figura 3.5 (função recursiva Python para o número de Fibonacci), observa-se que, ao utilizar um *wrapper* para uma biblioteca C pré-compilada, o acréscimo no desempenho torna-se bastante significativo.

```
1.  0.00471711158752
2.  [0.004283905029296875, 0.004303932189941406, 0.004244089126586914]
3.  0.00427595774333
```

Figura 3.17 - Análise de desempenho do *wrapper* para geração recursiva de número da sequência de Fibonacci.
Fonte: próprio autor (2016).

Apesar dos bons resultados obtidos com o uso do *wrapper*, essa abordagem não é definitiva. Estudos, como o apresentado em [ROGHULT 2016], mostram que a escolha da melhor abordagem depende da natureza da aplicação.

3.4. Computação Científica

A linguagem Python, nos últimos anos, tem recebido muita atenção da comunidade científica. Langtangen (2008) enumera uma série de características que têm favorecido a sua aceitação em projetos de pesquisa. São elas:

- **Sintaxe simples e clara**: além de maior aceitação por pesquisadores de todas as áreas, a tarefa de programação torna-se mais produtiva;

- **Integração entre simulação e visualização**: permite visualizar rapidamente e convenientemente resultados de computações;
- **Integração com outras ferramentas**: permite compor um ambiente integrado e completo para computação científica, evitando-se a necessidade de utilizar duas ou mais ferramentas para diferentes tarefas;
- **Integração com bibliotecas**: evita-se a necessidade de reescrita de bibliotecas codificadas em outras linguagens e que foram previamente validadas e/ou otimizadas;
- **Interface/Apresentação**: permite simplificar a interação com a aplicação, a formatação da apresentação de resultados e a criação de programas de demonstração (*demos*), através do uso de interfaces gráficas (*Graphic User Interface* - GUI);
- **Portabilidade**: cientistas podem optar utilizar as plataformas operacionais mais adequadas ao projeto, ou aquelas que lhes trazem maior conforto;
- **Recursos variados**: um conjunto bastante amplo de bibliotecas que trazem funções para diferentes tipos de aplicações, estruturas de dados nativas variadas e de fácil utilização, suporte a diferentes paradigmas de programação, interfaces dinâmicas e flexíveis para funções e outros elementos da linguagem, ambiente de programação interativo, entre outros.

Em [SCIPY 2016], são listadas várias ferramentas de código aberto para computação científica em Python. As ferramentas listadas abrangem várias utilidades, como, por exemplo: visualização de dados, integração com outros ambientes (e.g. R e Matlab), integração com linguagens de programação (e.g. C/C++ e Fortran), otimização, bibliotecas numéricas, armazenamento de dados e bancos de dados, programação paralela e distribuída. Além dessas, há ferramentas específicas para diferentes áreas do conhecimento, como para a astronomia, inteligência artificial e aprendizagem de máquina, estatística, biologia, economia, engenharia elétrica, geociências, modelagem molecular, processamento de sinais.

Dentre todas as ferramentas referenciadas, o site SciPy destaca as seguintes: as bibliotecas Numpy, SciPy, Matplotlib, pandas, SymPy, e o interpretador IPython.

Numpy [NUMPY 2016] é a biblioteca mais fundamental para computação científica em Python. Ela consiste de um módulo compilado (‘numpy’), que inclui uma gama de

funções para manipulação de matrizes multidimensionais, para conversão, classificação e ordenação de dados, bem como para cálculos matemáticos, financeiros e estatísticos. Por ser um módulo compilado, Numpy, geralmente, faz com que aplicações Python obtenham alto desempenho em operações numéricas e/ou manipulação de dados. A Figura 3.18 ilustra a versatilidade das funções de criação de array do módulo Numpy. Na figura, nas instruções das Linhas 2-9, o array:

- ‘a’ consiste de uma matriz 1x3, com os números 2, 3 e 4.
- ‘b’ consiste de uma matriz 2x3, onde a linha 1 contém os números 1, 2 e 3, e a segunda linha contém 4, 5 e 6.
- ‘c’ consiste de uma matriz 3x4, preenchida com números zero.
- ‘d’ consiste de uma matriz 2x3x4, preenchida com números um, de 64-bits.
- ‘e’ consiste de uma matriz 1x6, com os números 0, 1, 2, 3, 4 e 5.
- ‘f’ consiste de uma matriz 1x7, preenchida com números ponto flutuante entre 0 e 2, com intervalos de 0,3. No caso, seriam 0,0, 0,3, 0,6, 0,9, 1,2, 1,5 e 1,8.
- ‘g’ consiste de uma matriz 1x9, preenchida com 9 números ponto flutuante entre os números 0 e 2. No caso, seriam 0,0, 0,25, 0,5, 0,75, 1,0, 1,25, 1,5, 1,75 e 2,0.

```
1.  Import numpy as np
2.
3.  a = np.array( [2,3,4] )
4.  b = np.array( [(1,2,3), (4,5,6)] )
5.  c = np.zeros( (3,4) )
6.  d = np.ones( (2,3,4), dtype=np.int64 )
7.  e = np.arange(6)
8.  f = np.arange( 0, 2, 0.3 )
9.  g = np.linspace( 0, 2, 9 )
```

Figura 3.18 - Criando matrizes com Numpy.
Fonte: próprio autor (2016).

SciPy estende as funcionalidades do pacote NumPy, oferecendo funções de alto nível para processamento e visualização de dados, tornando o interpretador Python um ambiente com funcionalidades semelhantes às do Matlab, do R e do SciLa [SCIPY TUTORIAL 2016]. A biblioteca SciPy é composta de vários submódulos, para diferentes tipos de aplicações científicas. Seguem descrições de alguns deles:

- “scipy.constants” - constantes matemáticas e físicas.
- “scipy.special” - funções matemáticas e físicas.

- “scipy.integrate” - funções para integração numérica.
- “scipy.optimize” - funções de maximização/minimização para funções escalares, mínimos quadrados, sistemas de equações multivariadas, entre outras.
- “scipy.linalg” - funções para álgebra linear. Implementa as bibliotecas ATLAS LAPACK e BLAS.
- “scipy.spatial” - algoritmos para manipulação de estruturas de dados espaciais.
- “scipy.fftpack” - funções para processamento da Transformada de Fourrier.
- “scipy.ndimage” - funções para processamento de imagens multidimensionais.
- “Scipy.stats” - distribuições e funções estatísticas.

Matplotlib é uma biblioteca com funções simples, flexíveis e customizáveis para plotagem de gráficos 2D/3D de alta qualidade. Ela permite gerar uma grande variedade de gráficos, como em barra, linhas, torta/pizza, histogramas e superfície, que podem ser interativos ou gravados em arquivos, em vários formatos, como PNG e PostScript [TOSI 2009]. Em [MATPLOTLIB 2016], pode-se ver exemplos de gráficos.

Python Data Analysis Library, ou pandas, dispõe de um conjunto de funções e estruturas de dados para manipulação eficiente, fácil e expressiva de dados. Ela combina o alto desempenho das funções de manipulação de dados da biblioteca Numpy com os recursos de manipulação de dados presentes em sistemas de bancos de dados e planilhas eletrônicas. pandas inclui, por exemplo, operações sofisticadas de indexação, seleção e agregação [MCKINNEY 2012].

SymPy estende a linguagem python para compor um ambiente integrado com suporte à matemática simbólica. SymPy é um sistema algébrico computacional (*Computer Algebra System* - CAS [DAVENPORT, SIRET and TOURNIER 1988]), e com ele pode-se, por exemplo, simplificar expressões, calcular derivadas, integrais e limites, solucionar equações, entre outras operações da matemática simbólica [SYMPY 2016].

IPython é um interpretador Python alternativo com recursos nativos adicionais. Em [PÉREZ and GRANGER, 2007] há descrições detalhadas de vários desses recursos. Seguem alguns deles:

- Acesso aos estados da seção: armazena em memória e permite manipular todos os comandos digitados, e os respectivos resultados, em uma seção.

- Sistema de controle: disponibiliza um conjunto de comandos (chamados de "*magic commands*") para realizar tarefas diversas, como listar informações e (re)configurar o ambiente do interpretador, manipular variáveis, interagir com e executar comandos do terminal, executar comandos/scripts em outros interpretadores Python (e.g. Python3 e Pypy) e interpretadores de outras linguagens de programação (e.g. Ruby e Perl), importar facilmente as principais bibliotecas científicas, realizar análise de desempenho e *profiling*, entre outras.
- Acesso ao sistema operacional: suporta interação com o sistema operacional, como pesquisar, executar e criar atalhos para comandos e manipular (mover, copiar, remover, renomear, etc) arquivos e diretórios.
- Introspecção e ajuda dinâmica: pode-se, por exemplo, listar os elementos de módulos, auto completar o nome e atributos de objetos, consultar informações internas de objetos (descrição, arquivo em disco, código fonte, entre outras), completar nomes de comandos, arquivos e diretórios do sistema operacional.
- Acesso à execução do programa: permite executar um programa Python em arquivo, trazendo suas instruções para o ambiente em execução, como se o programador as tivesse digitado interativamente. Dessa forma, é possível analisar o desempenho, realizar *profiling*, depuração, plotagem, etc., em tempo de projeto/teste/execução.
- Integração com Matplotlib: disponibiliza um *shell* interativo para plotagem de gráficos, semelhante ao Matlab ou R.
- Computação distribuída e paralela: recursos para desenvolver, testar, depurar, executar e monitorar interativamente aplicações distribuídas e/ou paralelas.

Além dessas, há outras ferramentas que merecem destaque, como a IDE Spyder [SPYDER 2016]. Ela inclui, além de um editor de texto com recursos avançados de edição de código (introspecção, *syntax highlight*, *breakpoints*, consulta à bibliotecas, teclas de atalho, etc.), *variable explorer* e *file explorer*, *object inspector*, integração com IPython, Numpy, Scipy e Matplotlib, análise (estática) de código, entre outros. A Figura 3.19. mostra o ambiente de trabalho da IDE Spyder. Na figura, pode-se observar o editor de código (à esquerda), *object inspector* exibindo a documentação do Numpy (ao centro), *file explorer* listando os arquivos e diretórios da pasta de trabalho (à direita) e seção aberta do interpretador IPython (abaixo).

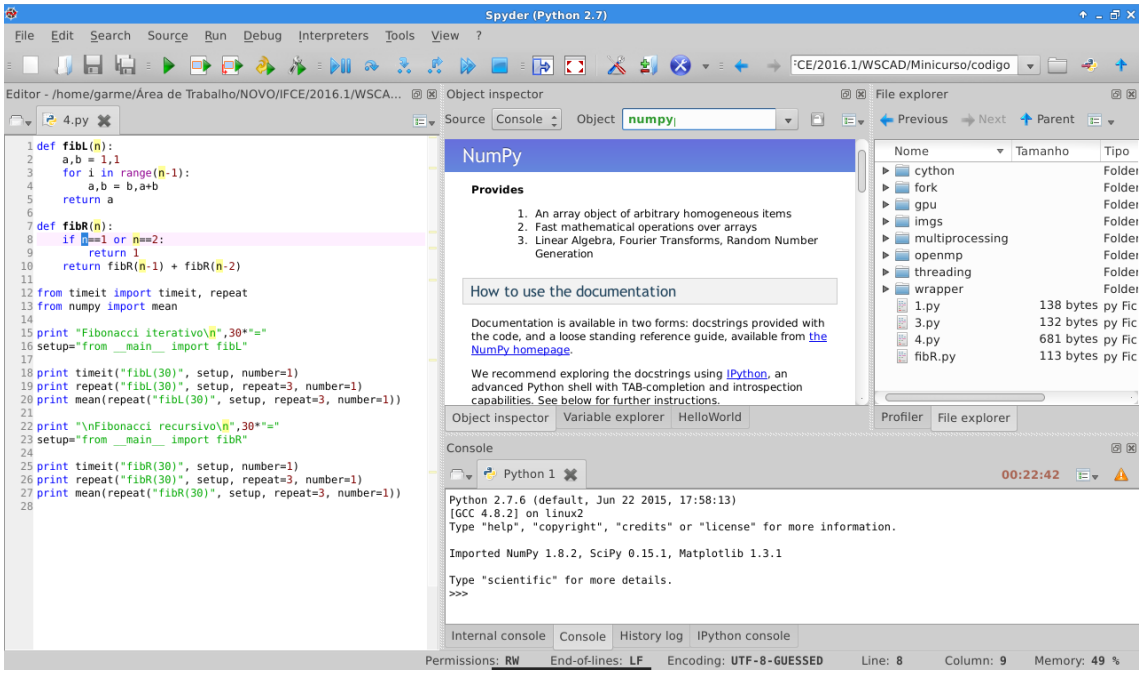


Figura 3.19 - Spyder - Ambiente integrado para desenvolvimento de aplicações científicas.
Fonte: próprio autor (2016).

Outras ferramentas de destaque podem ser encontradas em [PYDATA 2016] e em [NUMERICANDSCIENTIFIC 2016].

3.5. Programação Paralela em Memória Compartilhada

Em um sistema de memória compartilhada, os núcleos de processamento do computador podem ler e escrever na memória principal [PACHECO 2011]. Programação paralela em memória compartilhada consiste em programar múltiplas tarefas, que executarão simultaneamente, em diferentes núcleos de processamento, para solucionar cooperativamente um problema.

Em Python, há diferentes abordagens/mecanismos para suportar programação paralela em memória compartilhada (uma lista resumida pode ser vista em [PARALLEL PROGRAMMING 2016]). As subseções a seguir tratam de quatro modelos, partindo de implementações elementares de processos e *threads* até algumas mais avançadas.

3.5.1. *System call* "fork"

O módulo 'os' disponibiliza uma interface para que aplicações Python possam utilizar os serviços do sistema operacional (*system calls*). As funções do módulo 'os' possuem assinaturas semelhantes às das *system calls*, como é o exemplo da função 'fork', que é utilizada na criação de processos.

A Figura 3.20 mostra um exemplo de uso da *system call* 'fork'. No exemplo, a função 'fork' (Linha 3) é utilizada na criação de um processo filho, que tem sua imagem (em memória) substituída para execução do programa '/bin/ls' (Linha 4). Nessa aplicação, o processo filho será criado para listar o conteúdo do diretório de trabalho.

```
1.  from os import fork, execl
2.
3.  if  (fork() == 0) :
4.          execl( "/bin/ls", "ls", "-l")
5.
6.  print("O processo pai continua normalmente.\n")
```

Figura 3.20 - Criando um processo com a função 'fork'.
Fonte: próprio autor (2016).

Assim como as funções 'fork' e 'execl', no módulo 'os' estão disponíveis muitas outras funções para gerenciamento de processos, como 'getpid', 'system', 'signal', 'kill' e 'wait'. Em [OS 2016] há uma descrição de cada uma das funções disponíveis.

A comunicação inter-processos (*Inter Process Communication* - IPC) pode ser realizada de várias formas. Os quatro exemplos a seguir ilustram o uso dos mecanismos de comunicação *pipe*, *named pipe* (FIFO), fila de mensagens e memória compartilhada.

A Figura 3.21 mostra um exemplo de comunicação inter-processos utilizando *pipe*. No exemplo, o *pipe* é criado na Linha 2, no qual o processo filho escreve a mensagem "Hello" (Linhas 4-7) e o processo pai faz a leitura, imprimindo-a em seguida (Linhas 8-11).

```
1.  from os import pipe, fork, close, write, read
2.  p = pipe()
3.
4.  if(fork() == 0):
5.          close(p[0])
```

```
6.          write(p[1], "Hello")
7.          close(p[1])
8.  else:
9.          close(p[1])
10.         print read(p[0], 5)
11.         close(p[0])
```

Figura 3.21 - IPC utilizando *pipe*.
Fonte: próprio autor (2016).

Na Figura 3.22, é apresentado um exemplo de IPC utilizando uma FIFO, onde ela é criada na Linha 2, na qual o processo filho escreve a mensagem “Hello” (Linhas 4-7), e o processo pai faz a leitura, imprimindo-a em seguida (Linhas 8-11).

```
1.  from os import mkfifo, fork
2.  mkfifo("FILA")
3.
4.  if(fork() == 0):
5.          f = open("FILA", "w")
6.          f.write("Hello")
7.          f.close()
8.  else:
9.          f = open("FILA", "r")
10.         print f.read(5)
11.         f.close()
```

Figura 3.22 - IPC utilizando *named pipe*.
Fonte: próprio autor (2016).

O módulo ‘sysv_ipc’ disponibiliza os objetos “MessageQueue” e “SharedMemory” para implementar IPC por fila de mensagens e memória compartilhada, respectivamente. Os dois exemplos a seguir ilustram os uso desses objetos.

Na Figura 3.23, a Linha 5 mostra como a fila de mensagens é criada pelo processo filho. Em seguida, na Linha 6, o processo filho adiciona a mensagem “Hello” na fila. O processo pai espera o processo filho finalizar (instrução ‘wait’ na Linha 8) e nas Linhas 9-11, ele acessa a fila, lê e imprime a mensagem e, por fim, na Linha 12, remove a fila da memória.

```
1.  from os import fork, wait
2.  from sysv_ipc import MessageQueue, IPC_CREX
3.
4.  if(fork() == 0):
5.          mq = MessageQueue(9999, flags=IPC_CREX | 0666)
6.          mq.send("Hello")
7.  else:
8.          wait()
9.          mq = MessageQueue(9999)
10.         msg, msg_t = mq.receive()
11.         print msg
12.         mq.remove()
```

Figura 3.23 - IPC utilizando fila de mensagens.
Fonte: próprio autor (2016).

Já na Figura 3.24, podemos ver a alocação da memória compartilhada na Linha 5, pelo processo filho, na qual, na Linha 6, ele escreve a mensagem “Hello”. O processo pai espera o processo filho finalizar (Linha 8), acessa a região de memória compartilhada (Linha 9), faz a leitura e impressão na tela da mensagem (Linha 10) e exclui o seguimento de memória compartilhada (Linha 11).

```
1.  from os import fork, wait
2.  from sysv_ipc import SharedMemory, IPC_CREX
3.
4.  if(fork() == 0):
5.          shm = SharedMemory(9999, flags=IPC_CREX, size=5, mode=0666)
6.          shm.write("Hello")
7.  else:
8.          wait()
9.          shm = SharedMemory(9999)
10.         print shm.read()
11.         shm.remove()
```

Figura 3.24 - IPC utilizando memória compartilhada.
Fonte: próprio autor (2016).

A sincronização inter-processos também pode ser realizada de diferentes maneiras em Python. Por exemplo, o módulo ‘sysv_ipc’ disponibiliza um tipo de semáforo através do objeto “Semaphore” e o módulo ‘fcntl’ disponibiliza um *mutex*, também chamado de *file lock*, via função ‘lockf’.

3.5.2. *Threads*

A implementação de *threads* em Python é bastante simples, utilizando o objeto "Thread" do módulo "threading". Na aplicação da figura a seguir, são criadas e iniciadas cinco *threads* que imprimirão na tela seus respectivos números de ordem de criação.

O comportamento das *threads* pode ser visto na função definida nas Linhas 3-5, da Figura 3.25. Nas Linhas (Linhas 8-11), observa-se a criação (Linha 9) e inicialização (Linha 11) das *threads*, dentro do laço "for". Nas Linhas 13-14, o método "join" está sendo utilizado para sincronizar a execução das *threads* do programa (garante que a *main thread* só finalizará após o encerramento de todas as outras *threads*). Mais detalhes do módulo "threading", como os atributos da classe "Thread", outras formas de criação de *threads* e mecanismos de sincronização, podem ser encontrados em [THREADING 2016].

```
1.  from threading import Thread
2.
3.  def worker(num):
4.      print 'Thread: no. %s' % str(num+1)
5.      return
6.
7.  threads = []
8.  for i in range(5):
9.      t = Thread(target=worker, args=(i,))
10.     threads.append(t)
11.     t.start()
12.
13. for i in range(5):
14.             threads[i].join()
```

Figura 3.25 - Criando *threads* em Python.
Fonte: próprio autor (2016).

Uma *thread* Python é, na realidade, mapeada para uma *thread* de sistema (e.g. em sistemas Unix-like, seria mapeada para uma *thread* posix). Quando uma *thread* Python é criada, o interpretador cria e atribui a ela uma estrutura de dados que permite guardar informações sobre a *thread*, como estado, ID e pilha de execução de código Python. Em seguida, o interpretador cria a *thread* de sistema, a qual por sua vez

executará código Python. O interpretador possui uma variável global que aponta para a estrutura de dados da *thread* Python em execução. Com essa variável, o interpretador tem ciência de qual *thread* ele está executando as operações.

Entretanto, o interpretador Python não tem suporte para gerenciar as *threads* de sistema. Operações como definição de prioridades e escalonamento, são realizadas pelo sistema operacional. Assim, para garantir que será executada uma *thread* por vez no interpretador, Python utiliza um tipo de *mutex*, chamado de *Global Interpreter Lock* (GIL). Seu funcionamento é bastante simples: quando uma *thread* está em execução, ela trava (*lock*) o GIL. Para permitir que outras *threads* possam ser executadas, o interpretador realiza periodicamente verificações da existência de outras *threads*. Durante as verificações o GIL é liberado (*unlock*) e um outra *thread* pode entrar em execução. Há situações onde o GIL também pode ser liberado, como em operações de I/O, em chamadas de sistema (*system calls*) ou no uso de módulos implementados em linguagem nativa.

Observa-se que o uso de GIL é controverso, pois o mesmo não permite aumentar o desempenho de aplicações CPU-*bounded* ao se utilizar múltiplas *threads* Python (serão executadas sequencialmente). Neste caso, recomenda-se alguma das seguintes abordagens: 1) utilizar processos para implementação das tarefas paralelas; 2) utilizar interpretadores que não possuem implementação do GIL (como é o caso de Jython e IronPython); e 3) liberar explicitamente o GIL em módulos nativos com Cython [GUELTON, BRUNET and AMINI 2013] [AHMAD, LAKSHMINARASIMHAN and KHAN 2015].

3.5.3. Módulo “Multiprocessing”

Este módulo tem duas características básicas: a primeira refere-se à portabilidade, pois a *systemcall* ‘fork’ não está disponível em alguns sistemas operacionais, e, a segunda, a uma interface de alto nível, simplificando a manipulação de processos (ela é muito semelhante à do módulo “threading”). O módulo ‘multiprocessing’ inclui objetos Python para criação, comunicação e sincronização de processos. Os exemplos a seguir, ilustram o uso de alguns deles.

Na aplicação da Figura 3.26, são criados e iniciados cinco processos (Linhas 8-11) que basicamente imprimirão na tela seus respectivos números de ordem de criação. A função de cada processo pode ser vista nas Linhas 3-5.

```
1.  from multiprocessing import Process
2.
3.  def worker(num):
4.      print 'Processo: no. %s' % str(num+1)
5.      return
6.
7.  processos = []
8.  for i in range(5):
9.      p = Process(target=worker, args=(i,))
10.     processos.append(p)
11.     p.start()
```

Figura 3.26 - Criando processos com o módulo ‘multiprocessing’.
Fonte: próprio autor (2016).

O código da Figura 3.27 traz um exemplo de comunicação e sincronização de processos. O exemplo introduz o objeto “Array”, pertencente ao módulo “multiprocessing”, que é uma estrutura de dados compartilhada em memória e que traz na própria implementação um *mutex* para sincronização de acesso aos dados. Na figura, na Linha 7, o objeto “Array” é criado para armazenar cinco números inteiros (“‘i’, range(5)”) e com suporte de *mutex* ativado (“lock=True”). Na aplicação, cinco processos são criados (Linhas 9-10) para acessar o objeto “Array” e atribuir seu número de ordem de criação à respectiva posição (Linha 4).

```
1.  from multiprocessing import Process, Array
2.
3.  def worker(a, i):
4.      a[i] = i
5.      print "Adicionado o elemento %d" % i
6.
7.  a = Array('i', range(5), lock=True)
8.
9.  for i in range(5):
10.     Process(target=worker, args=(a, i)).start()
```

Figura 3.27 - Comunicação e sincronização de processos com o módulo ‘multiprocessing’.
Fonte: próprio autor (2016).

O próximo exemplo mostra uma maneira de paralelizar o programa de geração recursiva de número da sequência de Fibonacci (o código pode ser visto na Figura 3.28).

```
1.  from multiprocessing import Pool
2.
3.  def fibR(n):
4.      if n==1 or n==2:
5.          return 1
6.      return fibR(n-1) + fibR(n-2)
7.
8.  p = Pool(2)
9.  n = 30
10. print(sum(p.map(fibR, [n-1, n-2])))
```

Figura 3.28 - Geração recursiva de número da sequência de Fibonacci utilizando processos paralelos.
Fonte: próprio autor (2016).

No exemplo da Figura 3.28, utiliza-se um *pool* com dois processos (criado pela instrução da Linha 8), os quais calcularão em paralelo o 29º (n-1) e 28º (n-2) números da sequência. Após calculados, a soma dos números gerados resultará no valor do 30º número da sequência. Para o exemplo, buscou-se utilizar a função recursiva "fibR"(presente nas Linhas 7-10 da Figura 3.4), sem modificá-la. A atribuição da função recursiva a cada um dos processos pode ser vista na Linha 10. Nela, o método "map" do objeto "Pool" mapeia a função "fibR" com os números n - 1 e n - 2 para os processos do *pool*. A análise de desempenho dessa aplicação pode ser vista na Figura 3.29.

Comparando os resultados apresentados na Figura 3.29 com os da Figura 3.5, percebe-se que foi possível aumentar o desempenho da aplicação recursiva ao paralelizá-la com dois processos (*speedup* de quase 1,5x).

```
1.  0.305444955826
2.  [0.30824995040893555, 0.2948570251464844, 0.28676295280456543]
3.  0.294095675151
```

Figura 3.29 - Análise de desempenho da aplicação recursiva para geração de número da sequência de Fibonacci utilizando processos paralelos.
Fonte: próprio autor (2016).

3.5.4. OpenMP

OpenMP consiste de um conjunto de diretivas de compilação, variáveis de ambiente e bibliotecas de funções *runtime*, para descrever paralelismo no código fonte de uma

aplicação [CHANDRA 2001][JIANG 2008]. Um programa com suporte de OpenMP, inicia sua execução com apenas uma *thread* e, ao encontrar alguma diretiva de paralelização, a *thread* inicial criará um grupo de novas *threads* e torna-se mestre delas. Ao final de uma seção paralela, apenas a *thread* mestre continua executando [RAUBER and RÜNGER 2013]. As diretivas OpenMP surgiram a partir de duas necessidades: 1) portabilidade, em compiladores que não suportam OpenMP, as diretivas são ignoradas; e 2) otimização, as diretivas quando reconhecidas pelo compilador, permitem que se faça otimizações no código em nível de compilação [CHANDRA 2001]. Outra característica importante de OpenMP, é, segundo Chapman, Jost and Van Der Pas (2008), a possibilidade de escrever programas paralelos preservando o código sequencial original.

Utilizando Cython é possível paralelizar código Python com suporte de OpenMP. Cython dispõe de uma API que faz interface com OpenMP e, de acordo com Lanaro (2013), dentre as suas construções, “prange” é a mais simples para distribuir as operações contidas em laços entre *threads*.

Basicamente, “prange” deve ser utilizada em substituição à “range”, na definição de laços “for”. Contudo, há um detalhe muito importe que deve ser observado: ao utilizar “prange”, deve-se liberar o corpo do laço do interpretar. Desta forma, as *threads* criadas estarão livres do *Global Interpreter Lock* (GIL) e poderão executar em paralelo (o que não ocorre com o uso do módulo “threading”).

O exemplo a seguir ilustra como utilizar a construção “prange” e como liberar as *threads* do GIL para a aplicação recursiva de geração de número da sequência de Fibonacci (Figura 3.30 mostra o código do módulo Cython).

```
1.  from cython.parallel cimport prange
2.
3.  cdef int fib(int n) nogil:
4.          if n==1 or n==2:
5.                  return 1
6.          return fib(n-1) + fib(n-2)
7.
8.  def fibR(int n):
9.          cdef int i, soma = 0
10.
11.         for i in prange(n - 2 , n, nogil=True):
12.                 soma += fib(i)
```

```
13.
14.         return soma
```

Figura 3.30 - Aplicação recursiva para geração de número da sequência de Fibonacci utilizando threads OpenMP.
Fonte: próprio autor (2016).

As Linhas 3-6 mostram o código da função recursiva. Na Linha 3, a assinatura da função inclui a construção “cdef” para indicar que “fib” trata-se de uma função C pura. Ao utilizar “cdef” deve-se também especificar a tipagem das variáveis utilizadas na função. Essa adaptação é muito importante, pois, como a função será executada pelas *threads*, ela deve ser definida em C puro, para estar livre do interpretador Python (e do GIL).

Assim como na função “fib”, as estruturas de dados tratadas no corpo da iteração paralela também deverão ser definidas utilizando “cdef”. A Linha 9 mostra a definição das variáveis “i” (variável iterativa do laço “for”) e “soma” (recebe o resultado da soma dos número de Fibonacci gerados pelas *threads*).

As Linhas 11-12 mostram o laço “for” paralelo. Na Linha 11, “prange” está configurada para iterar entre dois números (“n - 2, n”) e para liberar as instruções do interpretador (“nogil=True”). Com essa configuração, a instrução da Linha 12 será executada por duas *threads*, sendo que a primeira *thread* executa “fib” com o valor de “i” da primeira iteração, e a segunda *thread* executa com o valor de “i” da segunda iteração. A análise de desempenho da aplicação paralela com suporte de OpenMP pode ser vista na Figura 3.31. Comparando os resultados de desempenho apresentados com os da Figura 3.5, percebe-se que houve aumento de desempenho.

```
1.  0.00875186920166
2.  [0.004308938980102539, 0.004277944564819336, 0.005513906478881836]
3.  0.00495402018229
```

Figura 3.31 - Análise de desempenho da aplicação recursiva para geração de número da sequência de Fibonacci utilizando *threads* OpenMP.
Fonte: próprio autor (2016).

3.6. Programação Paralela em Memória Distribuída

Um sistema de memória distribuída pode ser definido como um conjunto de computadores, cujos processadores acessam localmente sua respectiva memória, ligados por alguma estrutura de interconexão em rede [PACHECO 2011].

Torquati et al. (2014) cita quatro modelos de programação de sistemas de memória distribuída, que são *sockets*, chamada de procedimentos remotos *(Remote Procedure Calls* - RPC), memória virtual compartilhada e troca de mensagens (*message passing*), sendo este último o predominante em aplicações paralelas de alto desempenho. No passado, havia várias bibliotecas para suporte à comunicação via troca de mensagens, contudo havia também problemas de compatibilidade entre elas e de portabilidade entre arquiteturas paralelas [ZELKOWITZ 2000]. Com isso, criou-se o padrão MPI (*Message Passing Interface*) [MPI 2016], que é, atualmente, aceito amplamente pela comunidade de computação de alto desempenho [NIELSEN 2016].

Em Python, o módulo *MPI for Python* (MPI4Py) [DALCIN et al. 2011] inclui suporte de MPI em aplicações Python. MPI4Py na realidade é um *wrapper*, com interface orientada a objetos, para implementações do padrão MPI, como OpenMPI [OPENMPI 2016] e MPICH [MPICH 2016], e, entre os recursos disponíveis, estão diferentes tipos de comunicação ponto a ponto e coletiva.

3.6.1. Comunicação Ponto a Ponto

Comunicação ponto a ponto (*Peer-to-Peer* - P2P) é um mecanismo para troca de dados entre dois processos, sendo um deles o remetente (*sender*) e o outro o destinatário (*receiver*). As comunicações P2P podem ainda ser bloqueantes (os processos envolvidos ficam bloqueados nas funções de troca de dados até que a comunicação finalize) e não bloqueantes (o processo remetente insere os dados em algum *buffer* de comunicação, ficando disponível para realizar outras operações). MPI4Py implementa comunicação ponto a ponto bloqueante e não bloqueante.

A Figura 3.32 mostra um exemplo de comunicação P2P bloqueante. Na instrução da Linha 4, “MPI.COMM_WORLD” cria um objeto que engloba todos os processos criados para a aplicação. Na Linha 5, a variável “rank” recebe o número de ranking do respectivo processo. Das Linhas 7-9, o processo de rank 0 cria um array com 100 números, variando de 0 até 99, e o envia para o processo de rank 1. A função “send” está sendo utilizada para enviar o números do array para o processo de rank 1 (“dest=1”), com mensagem rotulada com número 11 (“tag=11”). Nas Linhas 10-11, o processo de rank 1, ao receber a array enviado pelo processo de rank 0 (“source=0, tag=11”), imprime-o na tela.

Para executar essa aplicação, utiliza-se o comando de terminal “mpirun -np 2 python p2pb.py”, onde “mpirun” é um comando para executar tarefas paralelas da biblioteca OpenMPI, “-np 2” é uma especificação de que serão utilizados dois processos paralelos e “p2pb.py” é o arquivo que inclui o código fonte da Figura 3.32.

```
1.  from mpi4py import MPI
2.  import numpy as np
3.
4.  comm = MPI.COMM_WORLD
5.  rank = comm.Get_rank()
6.
7.  if rank == 0:
8.      data = np.arange(100)
9.      comm.send(data, dest=1, tag=11)
10. elif rank == 1:
11.     print comm.recv(source=0, tag=11)
```

Figura 3.32 - Comunicação P2P bloqueante com MPI4Py.
Fonte: próprio autor (2016).

A mesma aplicação com comunicação não bloqueante pode ser vista na Figura 3.33. Nela, utiliza-se as funções “isend” e “irecv” para comunicação não bloqueante, e a instrução “wait”, na Linha 12, está sendo utilizada para que o processo de rank 1 aguarde a finalização da transferência de dados, antes de imprimir os dados array recebido.

```
1.  from mpi4py import MPI
2.  import numpy as np
3.
4.  comm = MPI.COMM_WORLD
5.  rank = comm.Get_rank()
6.
7.  if rank == 0:
8.      data = np.arange(100)
9.      req = comm.isend(data, dest=1, tag=11)
10. elif rank == 1:
11.     req = comm.irecv(tag=11)
12.     print req.wait()
```

Figura 3.33. Comunicação P2P não bloqueante com MPI4Py.
Fonte: próprio autor (2016).

3.6.2 Comunicação Coletiva

Neste tipo de comunicação, a transmissão de dados é realizada simultaneamente entre vários processos. MPI4Py implementa as funções de comunicação coletiva do padrão MPI, como *broadcast*, *scatter*, *gather*, *reduce* e outras. Ao contrário das funções de comunicação P2P, as instruções coletivas implementam apenas comunicação bloqueante. As Figuras 3.34, 3.35 e 3.36 trazem exemplos de aplicações com comunicação por *broadcast*, *scatter*/*gather* e *reduce*, respectivamente.

Na Figura 3.34, nas Linhas 7-8, o processo de rank 0 cria um array com 100 elementos, variando de 0 a 99, e, na Linha 12, envia-o para todos os outros processos do MPI.COMM_WORLD.

```
1.  from mpi4py import MPI
2.  import numpy as np
3.
4.  comm = MPI.COMM_WORLD
5.  rank = comm.Get_rank()
6.
7.  if rank == 0:
8.      data = np.arange(100)
9.  else:
10.     data = None
11.
12. comm.bcast(data, root=0)
```

Figura 3.34. Comunicação coletiva por *Broadcast* com MPI4Py.
Fonte: próprio autor (2016).

Na Figura 3.35, o processo de rank 0 cria um array com tamanho idêntico ao de número de processos no MPI.COMM_WORLD (Linha 8). O mesmo processo envia uma parte do array a cada um dos processos (inclusive ele mesmo), utilizando a função “scatter”, na Linha 12. Em seguida, cada processo realiza uma computação na sua respectiva parte do array (Linha 13) e envia o resultado para o processo de rank 0, que reagrupa-os através da função “gather” (na Linha 14).

```
1.  from mpi4py import MPI
2.
3.  comm = MPI.COMM_WORLD
4.  rank = comm.Get_rank()
```

```
5.  size = comm.Get_size()
6.
7.  if rank == 0:
8.      data = [x for x in range(size)]
9.  else:
10.     data = None
11.
12. data = comm.scatter(data, root=0)
13. data *= 10
14. data = comm.gather(data, root=0)
15.
16. if rank == 0:
17.     print data
```

Figura 3.35. Comunicações coletivas por *Scatter* e *Gather* com MPI4Py.
Fonte: próprio autor (2016).

Por fim, na aplicação da Figura 3.36, a função “reduce”, na Linha 6, é utilizada para somar (“op=MPI.SUM”) o valor dos rankings (“rank”) do processos do MPI.COMM_WORLD e enviar o resultado para o processo de rank 0 (“root=0”).

```
1.  from mpi4py import MPI
2.
3.  comm = MPI.COMM_WORLD
4.  rank = comm.Get_rank()
5.
6.  soma = comm.reduce(rank, op=MPI.SUM, root=0)
7.
8.  if rank == 0:
9.          print soma
```

Figura 3.36. Comunicação coletiva por *Reduce* com MPI4Py.
Fonte: próprio autor (2016).

Além da operação de soma, implementada por “MPI.SUM” (Linha 6), estão disponíveis ainda as funções: MPI.MAX (máximo), MPI.MIN (mínimo), MPI.PROD (produto), MPI.LAND e MPI.LOR (and e or lógico), MPI.BAND e MPI.BOR (and e or bit-a-bit), MPI.MAXLOC e MPI.MINLOC (valores máximo e mínimo, com rank do respectivo processo). Em [MPI4PY 2016] há a descrição completa da API, com todos os recursos implementados por MPI4Py.

3.7. Programação de GPU's

Uma Unidade de Processamento Gráfico (*Graphics Processing Unit* - GPU) consiste de um computador numérico dedicado com centenas de núcleos processadores, que podem ser programados para realizar operações numéricas simultaneamente em grandes quantidades de dados (paradigma *Single Program, Multiple Data* - SPMD) [SUCHARD 2011]. Estudos, como o apresentado em [OWENS 2007], mostram que, em problemas de processamento intensivo de dados, as GPUs levam grande vantagem sobre o desempenho das CPU's, e que por isso tem despertado grande interesse no meio científico.

Na programação de GPU's, atualmente, utiliza-se as plataformas CUDA [SANDERS and KANDROT 2010] e OpenCL [TSUCHIYAMA et al. 2010]. *Compute Unified Device Architecture*, ou CUDA, é uma biblioteca proprietária exclusiva para programação de placas gráficas da fabricante NVIDIA. Já OpenCL (*Open Computing Language*) consiste de um padrão aberto para programação paralela em diferentes tipos de *devices* (CPU's, GPU's e DSPs), de diferentes fabricantes, como Intel, AMD e a própria NVIDIA [KOMATSU 2010]. A avaliação dessas plataformas foge ao escopo deste minicurso, contudo estudos comparativos podem ser encontrados em [FANG, VARBANESCU and SIPS 2011][DU et al. 2012][SU et al. 2012][MALIK et al. 2012].

PyCUDA e PyOpenCL disponibilizam APIs para suportar CUDA e OpenCL em uma aplicação Python. Essas bibliotecas realizam compilação dinâmica do código que será executado no *device* (*Run-Time Code Generation* - RTCG), trazendo algumas vantagens, como automatização e flexibilidade na geração de código alvo [KLÖCKNER et at. 2012]. Outra característica importante, refere-se ao desempenho. Nessas bibliotecas, a compilação dinâmica gera código C nativo, de forma que, em um programa Python, as seções de código paralelo que executam nos *devices* possuem desempenhos similares aos de programas escritos puramente em C.

A estrutura de código de um programa Python com suporte de PyCUDA/PyOpenCL é bastante simples. Inicialmente, deve-se alocar espaço na memória do *device* para que ele possa armazenar o programa e os dados para processamento. Em seguida, realiza-se a cópia dos dados da memória do computador (*host*) para a memória previamente alocada no *device*, seguida da execução do programa. Por fim, copia-se os dados resultantes do processamento no *device* para a memória do *host*, seguindo com a liberação da memória alocada no primeiro.

No exemplo desta seção, optou-se por utilizar apenas código PyOpenCL, por dispor de sintaxe relativamente mais simples do que PyCUDA, por OpenCL abranger uma maior variabilidade de dispositivos e fabricantes, sendo então mais adequado para programadores iniciantes ou por indisponibilidade de plataformas gráfica da NVIDIA, e por considerar que todos os conceitos são semelhantes em aplicações Python com PyCUDA.

A figura à seguir mostra um exemplo de aplicação de soma vetorial utilizando PyOpenCL.

```
1.  import numpy
2.  import pyopencl as cl
3.
4.  A = numpy.random.rand(1000).astype(numpy.float32)
5.  B = numpy.random.rand(1000).astype(numpy.float32)
6.  C = numpy.zeros(1000, dtype=numpy.float32)
7.
8.  kernel = """__kernel void
9.  soma_vetores(__global float* A, __global float* B, __global float* C){
10.     int i = get_global_id(0);
11.     C[i] = A[i] + B[i];
12. }"""
13.
14. context = cl.create_some_context()
15. queue = cl.CommandQueue(context)
16. program = cl.Program(context, kernel).build()
17.
18. A_dev = cl.Buffer(context, cl.mem_flags.COPY_HOST_PTR, hostbuf = A)
19. B_dev = cl.Buffer(context, cl.mem_flags.COPY_HOST_PTR, hostbuf = B)
20. C_dev = cl.Buffer(context, cl.mem_flags.COPY_HOST_PTR, hostbuf = C)
21.
22. program.soma_vetores(queue, A.shape, None, A_dev, B_dev, C_dev)
23.
24. cl.enqueue_read_buffer(queue, C_dev, C).wait()
25.
26. print(C)
```

Figura 3.37 - Análise de desempenho da aplicação recursiva para geração de número da sequência de Fibonacci utilizando threads OpenMP.
Fonte: próprio autor (2016).

Nas linhas 4-6, são criados os arrays A, B e C no *host*, onde os dois primeiros foram alocados com mil números ponto flutuante de 32 bits, com valores gerados

aleatoriamente, e o terceiro é inicializado com zeros. Os arrays A e B serão somados no *device* e o resultado é será gravado no array C no *host*.

Nas Linhas 8-12, a variável "kernel" recebe, em formato de string, a definição da função C que será executada no *device* (no jargão OpenCL, a função é chamada de *kernel*). Essa função será compilada dinamicamente durante a execução do programa Python. Nas Linhas 8-9, temos a assinatura da função "soma_vetores", com seus parâmetros precedidos pela construção "__global" para sinalizar que os parâmetros estarão alocados na memória global do *device*. Na Linha 10, a instrução "get_global_id(0)" recupera o ID da *thread* na *dimensão 0*. O ID é utilizado para relacionar uma determinada *thread* à operação de soma nas respectivas posições dos arrays (Linha 11). Maiores detalhes sobre a organização de memória e o modelo de programação OpenCL, podem ser encontrados em [TSUCHIYAMA et al. 2010].

A instrução da Linha 14 realiza a criação de um novo contexto, que consiste do conjunto de recursos que serão disponibilizados para que o *kernel* possa executar na GPU. Entre os recursos, estão os *devices*, os objetos de programa (código fonte e código compilado do *kernel*) e de dados. Após a criação do contexto, deve-se criar uma fila para guardar os comandos para execução (*command-queue*). Os comandos na fila podem ser executados na ordem de enfileiramento ou aleatoriamente, e são executados assincronamente em relação ao *host*. A instrução da Linha 15 cria a fila de comandos atrelada ao contexto anteriormente criado e, a instrução da Linha 16, realiza a compilação o *kernel*.

As instruções das Linhas 18-20 alocam, no *device*, memória suficiente para guardar os dados dos array A, B e C. Nessas instruções, depois da alocação, os dados são trazidos de A, B e C e armazenados no *device*, com as respectivas referências de acesso "A_dev", "B_dev" e "C_dev".

A instrução da Linha 22 realiza a execução do *kernel* "soma_vetores". Seus parâmetros incluem (da esquerda para direita): a fila de comandos ("queue"), a geometria que determina como as *threads* serão referenciadas ("A.shape"), e os três arrays envolvidos na soma (os parâmetros da função). Ao final das operações paralelas de soma, os resultados são armazenados em C_dev (na memória global do *device*).

A instrução na Linha 24 realiza a cópia dos dados em C_dev para o array C, residente na memória do *host*. Observe que utilizou-se o método 'wait()', pois ele garante que os dados sejam totalmente copiados, antes de serem utilizados por outra

instrução do programa *host* (como é o caso da instrução na Linha 26, que imprimirá na tela os dados em C).

3.8. Pesquisas e Conferências

Neste minicurso, apresentou-se parte do potencial de Python na computação de alto desempenho. A simplicidade da linguagem, o suporte de bibliotecas de alto desempenho e a integração com bibliotecas pré-compiladas e com programas de terceiros, fazem da linguagem Python uma ferramenta de fácil aprendizado, muito produtiva e que pode trazer bons resultados em diferentes tipos de aplicação.

O ecossistema Python tem sido aplicado em diferentes campos científicos. Entre eles, pode-se citar [MILLMAN and AIVAZIS 2011] [LANGTANGEN 2012] [SCIPY 2016] [NUMERIC AND SCIENTIFIC 2016][CASES 2016]:

- Acessibilidade e inclusão social
- Análise musical
- Aprendizagem de máquina, sistemas de recomendação, visão computacional e algoritmos genéticos
- Automação industrial e Robótica
- Computação em nuvem e virtualização
- Computação gráfica
- *Data mining*, *Bigdata* e *Visualization*
- Eletronic Design Automation
- Educação e educação a distância
- Geociências e astronomia
- Genética e análises biomoleculares
- Processamento de sinais digitais
- Processamento digital de imagens
- Problemas de classificação e de *clustering*
- Radioterapia e ultrassonografia
- Sistemas distribuídos e redes de sensores, e
- Redes e segurança em redes.

Tem sido realizado também muito esforço no desenvolvimento do desempenho da própria linguagem. As pesquisas de área têm focando na criação de novas bibliotecas e frameworks para: implementação de novas técnicas de paralelização, automatização de paralelização de código e programação de dispositivos paralelos; análises eficientes de grandes bases de dados; modelagem e análises matemáticas e estatísticas; otimização e simplificação da integração com outras linguagens de programação, com bibliotecas e com programas de terceiros.

Grande parte dos trabalhos científicos são frequentemente divulgados nas conferências específicas da linguagem. Entre as principais, estão as *PyCon* [PYCON 2016] e *SciPy* [SCIPY CONFERENCES 2016]. *PyCon* são conferências anuais, realizadas em diferentes países pelas comunidades Python locais, com objetivos de atrair novos usuários e de divulgar as principais novidades e tendências da linguagem. Geralmente, sua a programação varia entre apresentações de trabalhos, painéis, mesas redondas, *coding dojo*, minicursos e tutoriais, promovidos por usuários, desenvolvedores da linguagem, pesquisadores e empresas. Entre as *PyCon*, destaca-se a *PyCon US*, a qual além de ser primeira *PyCon*, realizada inicialmente em 2003, é a maior em programação, divulgação e promoção, sendo patrocinada por grandes corporações, como Google, IBM, Intel e Microsoft [PYCONUS 2016]. As *SciPy* possuem formatos semelhantes aos das *PyCon*, contudo sua programação foca em resultados de trabalhos científicos. No Brasil, realiza-se anualmente a *PythonBrasil* [PYTHONBRASIL 2016] e, em 2016, realizou-se, em Santa Catarina, a quarta edição da *SpiPy Latin America* [SCIPYLA 2016]. Destacam-se ainda:

- *EuroPython, Python Conference in Europe* [EUROPYTHON 2016],
- *EuroSciPy, European Conference on Python in Science* [EUROSCIPY 2016], e
- *PyData, Python Data Science Community Conference* [PYDATA 2016].

3.9. Considerações Finais

Pesquisadores têm buscado aumentar o desempenho de suas aplicações, ora para obter os resultados que antes eram impossíveis, devido às limitações da tecnologia, ora para realizar computações mais eficientes.

Programação de aplicações de alto desempenho, entretanto, não é uma tarefa simples e, no atual contexto tecnológico, com o advento dos processadores *multi-core* (e

dos *many-core*), para um maior aproveitamento de todo o potencial dos computadores, deve-se dividir as funções da aplicação em tarefas paralelas para execução simultânea em diferentes núcleos de processamento. Em geral, utiliza-se as linguagens de programação C/C++, consideradas as mais eficientes e que incluem uma variedade de modelos de programação paralela, mas que possuem sintaxes difíceis e confusas, limitando seu aprendizado e domínio.

Surge então a linguagem de programação Python, a qual inclui, além de uma sintaxe simples, definida com viés em produtividade, diferentes recursos para otimização de seu desempenho. Dentre eles, pode-se destacar: mecanismos para análise e diagnóstico de desempenho, suporte de bibliotecas para manipulação de dados e cálculos numéricos eficientes, compilação de código Python para código nativo, integração com bibliotecas compiladas, com programas de terceiros e com diferentes modelos/ambientes de programação paralela.

Na ciência, com o passar do anos, Python vem ganhando notoriedade. Em muitos projetos de pesquisa, em variadas áreas do conhecimento, tem-se optado pelo uso da linguagem, também por sua flexibilidade e suporte, em função dos já consagrados ambientes de análise de dados, como é o caso do Matlab ou R. Além do mais, por trás de Python, há uma grande comunidade (usuários, desenvolvedores da linguagem, empresas e grandes corporações, cientistas e centros de pesquisa) que desenvolve e frequentemente promove todo o ecossistema da linguagem.

Neste minicurso, foram apresentados alguns dos principais recursos desse ecossistema, onde o foco se deu em técnicas de programação de alto de desempenho. Objetivou-se, por consequência, oferecer mais opções de suporte aos projetos de novas aplicações eficientes, através da expansão da visão de potencialidades da linguagem Python, no entanto, naturalmente, sem a intenção de esgotar os assuntos abordados.

Referências

Ahmad, M., Lakshminarasimhan, K. and Khan, O. (2015). Efficient parallelization of path planning workload on single-chip shared-memory multicores. In High Performance Extreme Computing Conference (HPEC), 2015 IEEE (pp. 1-6). IEEE.

Aycock, J. (2003). A brief history of just-in-time. ACM Computing Surveys (CSUR), 35(2), 97-113.

Bandyopadhyay, R. (2009). Compiling dynamic languages via statically typed functional languages. ProQuest.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011). Cython: The best of both worlds. Computing in Science & Engineering, 13(2), 31-39.

Cai, X., Langtangen, H. P. and Moe, H. (2005). On the performance of the Python programming language for serial and parallel scientific computations.Scientific Programming, 13(1), 31-56.

Cases. (2016). Python Success Histories – Science. https://www.python.org/about/success/#scientific. September.

Chandra, R.; Dagum, L., Kohr, D.; Maydan, D.; McDonald, J. and Menon, R. (2001). Parallel programming in OpenMP. Morgan Kaufmann.

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

Chun, W. (2001). Core python programming (Vol. 1). Prentice Hall Professional.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2009). Introduction to algorithms. MIT press.

Dalcin, L., Kler, P., Paz, R. and Cosimo, A. (2011). Parallel Distributed Computing using Python, Advances in Water Resources, 34(9):1124-1139.

Davenport, J. H., Siret, Y. and Tournier, E. (1988). Computer algebra (Vol. 5). London: Academic Press.

Decaluwe, J. (2004). MyHDL: a python-based hardware description language. Linux journal, 2004(127), 5.

Diaz, J., Munoz-Caro, C. and Nino, 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.

Dietz, W., Li, P., Regehr, J. and Adve, V. (2015). Understanding integer overflow in C/C++. ACM Transactions on Software Engineering and Methodology (TOSEM), 25(1), 2.

Doglio, F. (2015). Mastering Python High Performance. Packt Publishing Ltd.

Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., and Dongarra, J. (2012). From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing, 38(8), 391-407.

EuroPython. (2016). Python Conference in Europe. http://www.europython-society.org/europython/. August.

EuroScipy. (2016). European Conference on Python in Science. https://www.euroscipy.org/. August.

Fang, J., Varbanescu, A. L. and Sips, H. (2011). A comprehensive performance comparison of CUDA and OpenCL. In 2011 International Conference on Parallel Processing (pp. 216-225). IEEE.

Ghazi, K. R., Lefèvre, V., Théveny, P. and Zimmermann, P. (2010). Why and how to use arbitrary precision. Computing in Science & Engineering, 12(1-3), 5-5.

Guelton, S., Brunet, P. and Amini, M. (2013, November). Compiling Python modules to native parallel modules using Pythran and OpenMP annotations. In Proc. Workshop on Python for High Performance and Scientific Computing.

Gupta, R. (2003). Making use of Python. John Wiley & Sons.

Hattem, R. V. (2016). Mastering Python. Packt Publ.

Hetland, M. L. (2006). Beginning Python: from novice to professional. Apress.

Holkner, A. and Harland, J. (2009). Evaluating the dynamic behaviour of Python applications. In Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91 (pp. 19-28). Australian Computer Society, Inc.

Hunt, C., & John, B. (2011). Java performance. Prentice Hall Press.

Jiang, Y. (2008). Design and Implementation of Tool-chain Framework to Support OpenMP Single Source Compilation on Cell Platform. ProQuest.

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P. and Fasih, A. (2012). PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation, Parallel Computing, Volume 38, Issue 3, Pages 157-174.

Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, Matthias S. and Nagel, W. E. (2008). The vampir performance analysis tool-set. In Tools for High Performance Computing (pp. 139-155). Springer Berlin Heidelberg.

Knuth, D. E. (1992) Literate Programming. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA.

Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H. and Kobayashi, H. (2010). Evaluating performance and portability of OpenCL programs. In The fifth international workshop on automatic performance tuning (Vol. 66).

Lanaro, G. (2013). Python High Performance Programming. Packt Publishing Ltd.

Langtangen, H. P. (2012) A Primer on Scientific Programming with Python, 3rd edition. Springer.

Larus, J. R. (1990). Abstract execution: A technique for efficiently tracing programs. Software: Practice and Experience, 20(12), 1241-1258.

Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In Code Generation and Optimization, 2004. CGO 2004. International Symposium on (pp. 75-86). IEEE.

Luszczek, P., Gates, M., Kurzak, J., Danalis, A. and Dongarra, J. (2016). Search Space Generation and Pruning System for Autotuners. IEEE International Parallel and Distributed Processing Symposium Workshops, pp. 1545-1554.

Lutz, M. (2013). Learning python. O'Reilly Media, Inc.

Lutz, M. (2006). Programming Python, 3rd. Ed. O'Reilly Media, Inc.

Malik, M., Li, T., Sharif, U., Shahid, R., El-Ghazawi, T., and Newby, G. (2012). Productivity of GPUs under different programming paradigms. Concurrency and computation: practice and experience, 24(2), 179-191.

Martelli, A. (2006). Python in a Nutshell. O'Reilly Media, Inc.

Matplotlib. (2016) Matplotlib: Python Ploting. http://matplotlib.org/. August.

McKinney, W. (2012). Python for data analysis: Data wrangling with Pandas, NumPy, and IPython. " O'Reilly Media, Inc.".

Millman, K. J. and Aivazis, M. (2011). Python for scientists and engineers. Computing in Science & Engineering 13.2: 9-12.

Model, M. L. (2009). Bioinformatics Programming Using Python: Practical Programming for Biological Data. O'Reilly Media, Inc.

Mohror, K., and Karavanic, K. L. (2009). Evaluating similarity-based trace reduction techniques for scalable performance analysis. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (p. 55). ACM.

MPI. (2016). Message Passing Interface Forum. http://www.mpi-forum.org/. August.

MPI4Py. (2016). MPI for Python. https://pythonhosted.org/mpi4py/. August.

MPICH. (2016). High-Performance Portable MPI. https://www.mpich.org/. September.

Murri, R. (2013) Performance of Python runtimes on a non-numeric scientific code. Proceedings of the EuroSciPy2013 Conference.

Nanz, S. and Furia, C. A. (2015). A comparative study of programming languages in Rosetta Code. In Proceedings of the 37th International Conference on Software Engineering-Volume 1 (pp. 778-788). IEEE Press.

Nielsen, F. (2016). Introduction to HPC with MPI for Data Science. Springer.

NumericAndScientific. (2016). Numeric and Scientific Python. https://wiki.python.org/moin/NumericAndScientific. August.

Numpy. (2016). Numerical Python. http://www.numpy.org/. August.

Oliphant, T. E. (2007). Python for scientific computing. Computing in Science & Engineering, 9(3), 10-20.

OpenMPI. (2016). Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/. September.

OS. (2016). os - Miscellaneous operating system interfaces. https://docs.python.org/2/library/os.html. August.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E. and Purcell, T. J. (2007). A Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum. 26:80–113.

Pacheco, P. (2011). An introduction to parallel programming. Elsevier.

ParallelProgramming. (2016) Parallel Processing and Multiprocessing in Python. https://wiki.python.org/moin/ParallelProcessing. August.

Perez, F., Granger, B. E. and Hunter, J. D. (2011). Python: an ecosystem for scientific computing. Computing in Science & Engineering,13(2), 13-21.

Pérez, F., and Granger, B. E. (2007). IPython: a system for interactive scientific computing. Computing in Science & Engineering, 9(3), 21-29.

Phillips, D. (2015). Python 3 Object-oriented Programming. Packt Publishing Ltd.

PyData. (2016). Community for Developers and Users of Open Source Data Tools. http://pydata.org/. August.

PythonBrasil. (2016). Conferência Brasileira da Comunidade Python. http://2016.pythonbrasil.org.br/. August.

PyConUS. (2016). PyCon 2016. . https://us.pycon.org/2016/. August.

PyCon. (2016) PyCon Home at Python.org. https://www.python.org/community/pycon/. August.

PyPI. (2016). Python Package Index. https://pypi.python.org/pypi. August.

Python. (2016). The Official Home of The Python Programming Language. https://www.python.org/. September.

Rane, A. and Browne, J. (2011). Performance optimization of data structures using memory access characterization. In 2011 IEEE International Conference on Cluster Computing (pp. 570-574). IEEE.

Rauber, T. and Rünger, G. (2013). Parallel programming: For multicore and cluster systems. Springer Science & Business Media.

Roghult, A. (2016). Benchmarking Python Interpreters: Measuring Performance of CPython, Cython, Jython and PyPy. Master's Thesis in Computer Science - School of Computer Science and Communication(CSC), Royal Institute of Technology, Stockholm.

Sanders, J., and Kandrot, E. (2010). CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional.

Sanner, M. F. (1999). Python: a programming language for software integration and development. J Mol Graph Model, 17(1), 57-61.

SciPy. (2016). Scientific Computing Tools for Python. http://www.scipy.org/. August.

SciPy Conferences. (2016). SciPy Conferences. http://conference.scipy.org/. August.

SciPyLA. (2016). SciPy Latin America 2016 - Scientific Computing with Python. http://scipyla.org/conf/2016/. August.

SciPy Tutorial. (2016). Scipy Reference Guide. http://docs.scipy.org/doc/scipy-0.18.0/reference/. August.

Spyder. (2016). Scientific PYthon Development EnviRonment. https://pythonhosted.org/spyder/. August.

Su, C. L., Chen, P. Y., Lan, C. C., Huang, L. S., and Wu, K. H. (2012). Overview and comparison of OpenCL and CUDA technology for GPGPU. In Circuits and Systems (APCCAS), 2012 IEEE Asia Pacific Conference on (pp. 448-451). IEEE.

Suchard, M. A., Wang, Q., Chan, C., Frelinger, J., Cron, A., and West, M. (2010). Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures. Journal of Computational and Graphiercal Statistics, 19(2), 419-438.

SymPy. (2016). Python Library for Symbolic Mathematics. http://www.sympy.org/en/index.html. August.

Threading. (2016). Threading - Higher-level Threading Interface. https://docs.python.org/2/library/threading.html. September.

TIOBE Software. (2016). TIOBE Programming Community Index. http://www.tiobe.com/tiobe-index/. August.

Torquati, M., Bertels, K., Karlsson, S. and Pacull, F. (2014). Smart multicore embedded systems. Springer.

Tosi, S. (2009). Matplotlib for Python developers. Packt Publishing Ltd.

Tsuchiyama, R., Nakamura, T., Iizuka, T., and Asahara, A. (2010). The OpenCL Programming Book, Fixstars Corporation.

Zelkowitz, M. (2000). Emphasizing Distributed Systems (Vol. 53). Academic Press.

Zhang, Y. (2015). An Introduction to Python and Computer Programming. Springer Singapore.

Capítulo

4

Introdução à Vetorização em Arquiteturas Paralelas Híbridas

Silvio Stanzani
Raphael Cóbe
Rogério Iope
Igor Freitas

Resumo

Técnicas de vetorização representam o recurso mais básico para explorar as possibilidades oferecidas por modernas arquiteturas paralelas, compostas por combinações de recursos incluindo processadores de múltiplos núcleos, memória em múltiplos níveis, aceleradores e coprocessadores. O objetivo deste minicurso é apresentar algumas das características oferecidas por compiladores C e C++, bem como extensões dessas linguagens e recursos da versão 4 do OpenMP que permitem explorar vetorização automática e vetorização guiada em tais arquiteturas.

4.1. Introdução

A contínua evolução dos microprocessadores nas últimas décadas proporcionou melhorias significativas no desempenho dos computadores, com o desenvolvimento de microarquiteturas cada vez mais complexas, incluindo técnicas de paralelismo em nível de instrução de máquina, aumento dos níveis de memória *cache*, além de consideráveis avanços na frequência do *clock* interno que rege a execução das microinstruções. No entanto, as condições que permitiram tais melhorias começaram a atingir seus limites por volta de 2005, de modo que hoje em dia não se conseguem grandes avanços apenas com aumentos nas frequências de *clock*, ou melhorias na microarquitetura e/ou na hierarquia de *cache*, ou ainda através de incrementos nas taxas de transferências de dados de/para a memória. Para explorar as novas arquiteturas de modo a se obter melhorias significativas de desempenho torna-se cada vez mais necessário aplicar técnicas de paralelismo explícito, pois o aumento no desempenho dos computadores atuais tem se baseado no desenvolvimento de arquiteturas computacionais paralelas.

As modernas arquiteturas de computadores podem ser compostas por diversos processadores, que por sua vez dispõem de múltiplos núcleos de processamento, além de um sistema de memória organizado em múltiplos níveis. Mais recentemente, tem se popularizado o desenvolvimento de sistemas computacionais compostos por coprocessadores e aceleradores que dispõem de muitos núcleos e grande capacidade de processamento vetorial, e atuam em conjunto com os processadores principais [1]. Tais arquiteturas compostas por recursos heterogêneos são definidas também como arquiteturas paralelas híbridas.

Arquiteturas paralelas híbridas disponibilizam múltiplos níveis de paralelismo, tais como paralelismo no nível de dados (vetorização), paralelismo no nível de *threads* e paralelismo no nível de processos, que podem ser usados de modo combinado. A vetorização é o nível mais básico de paralelismo que pode ser explorado, e consiste em aplicar uma mesma operação em um conjunto de pares de operandos; as instruções de máquina que permitem tais operações são chamadas de instruções vetoriais [2]. A utilização de tais recursos, devidamente combinados com técnicas de acesso eficiente a sistemas de memória de múltiplos níveis, é essencial para melhorar o desempenho de aplicações em tais arquiteturas.

Atualmente, boa parte dos compiladores e linguagens de programação oferecem suporte para uso de recursos de processamento vetorial, basicamente de três formas. A primeira é por meio de parâmetros de otimização do compilador, que permitem utilizar instruções vetoriais de modo automático. A segunda é utilizando extensões das linguagens de programação, que servem para instruir o compilador quanto ao uso de tais recursos de modo mais preciso, e também para forçar o uso de tais instruções. Tais extensões são disponibilizadas pela linguagem C, C++ e Fortran, estão presentes na versão 4.0 da especificação OpenMP (Open Multi-Processing) [3], e também estão presentes em outros *frameworks*, tais como, OpenCL [4] e OpenACC [5]. A terceira é utilizando bibliotecas ou funções que fornecem acesso direto a instruções vetoriais, como por exemplo o *Intrinsics* [6].

O objetivo deste minicurso é capacitar os estudantes quanto ao uso das unidades de processamento vetorial e técnicas de acesso eficiente a memória de múltiplos níveis em arquiteturas paralelas híbridas, com ênfase em arquiteturas *multicore* e *manycore* da Intel. Nesse sentido, serão abordadas as extensões disponibilizadas pelas linguagens e pelo padrão OpenMP 4.0, incluindo os parâmetros dos compiladores. A estrutura geral do minicurso é detalhada na Seção 4.1.1.

4.1.1. Estrutura do Minicurso

Esse minicurso está estruturado da seguinte forma: a Seção 0 apresenta o conceito de arquitetura paralela híbrida, descrevendo como funciona o sistema de memória e as unidades de processamento vetorial. A Seção 0 descreve um fluxo para explorar o uso de vetorização usando a ferramenta Intel Advisor. A Seção 0 descreve técnicas para melhorar o desempenho do acesso à memória. A Seção 0 descreve como explorar paralelismo de dados nas unidades de processamento vetorial por meio de parâmetro de compilação e por meio de extensões da linguagem C e OpenMP 4.0. Finalmente, a Seção 0 apresenta as conclusões.

4.2. Arquiteturas Paralelas Híbridas

Sistemas computacionais paralelos modernos são constituídos por uma combinação de recursos que incluem processadores com múltiplos núcleos, subsistemas de memória que podem apresentar múltiplos níveis de acesso, e subsistemas de entrada e saída [7].

Um tipo de arquitetura computacional que tem se popularizado são as arquiteturas paralelas híbridas [1], que são sistemas computacionais paralelos compostos por recursos heterogêneos, que podem ser coprocessadores e/ou aceleradores (gráficos ou de uso geral), que podem acrescentar dezenas ou centenas de elementos de processamento extras, acessíveis ao programador, conforme mostrado na Figura 1.

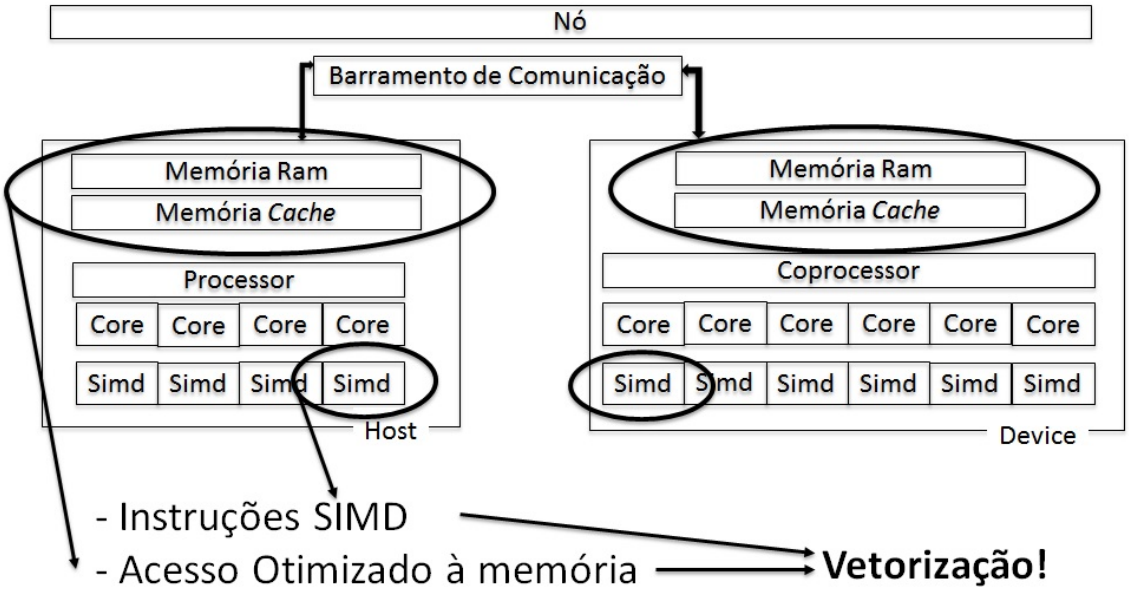


Figura 1. **Arquitetura paralela híbrida.**

No nível do núcleo de processamento dois aspectos podem ser considerados para aumentar o desempenho de aplicações. O primeiro é o sistema de memória em múltiplos níveis, que disponibiliza unidades de memória com alto desempenho próximas ao processador, chamadas de *cache* [8]. O segundo aspecto são as unidades de processamento vetorial, que permitem explorar o paralelismo de dados, também chamado de SIMD (*Single Instruction Multiple Data*), que consiste em executar uma mesma instrução com diversos pares de operandos [2]. O conjunto de técnicas que exploram esses dois aspectos presentes em um núcleo é chamada vetorização, e é essencial para melhorar o desempenho de aplicações

A combinação do uso eficiente de sistemas de memória em múltiplos níveis com o uso de unidades de processamento vetorial é essencial para atingir bons níveis de desempenho de vetorização. Nesse sentido, a Seção 0 descreve o funcionamento do sistema de memória e a Seção 0 descreve as unidades de processamento vetorial.

4.2.1. Sistema de Memória

O sistema de memória da maioria dos sistemas computacionais modernos em geral é organizado em múltiplos níveis. Essa organização tem como objetivo permitir explorar aspectos de localização para melhorar o desempenho das aplicações [7].

O sistema de memória das arquiteturas de múltiplos núcleos (*multicore* e *manycore*) da Intel é organizado de acordo com os seguintes níveis:

- Registradores do processador: é a memória interna do processador, que armazena os dados das instruções que serão executadas;
- *Cache*: armazena fragmentos de dados de programas que serão utilizados para a execução das próximas instruções, além de dados temporários frequentemente utilizados pelos programas em execução;
- Memória principal: armazena todos os dados necessários para a execução dos programas em todas as unidades de processamento;
- Memória secundária: são os dispositivos que armazenam dados e programas que podem estar ou não em execução.

A memória *cache* é uma memória especial que apresenta uma velocidade de acesso maior que a da memória principal. Ela funciona como uma área intermediária de armazenamento, que é utilizada para armazenar conteúdo temporário que será utilizado pelas próximas instruções que entrarão em execução. Esse tipo de memória apresenta um papel essencial no desempenho da execução das aplicações, pois tem o objetivo de minimizar o impacto da transferência de dados entre o processador e a memória principal [8].

A escolha de quais dados são armazenados na memória *cache* é feita pelo sistema operacional (S.O.), e os critérios para realizar tal escolha, na maioria dos sistemas operacionais, são os seguintes:

- Temporal: define que se um determinado item acabou de ser referenciado, ele provavelmente será referenciado novamente em breve; um exemplo disso são dados usados por todas as iterações de um laço;

- Espacial: define que se um determinado item acabou de ser referenciado, itens próximos a ele também serão referenciados em breve; um exemplo é um laço que realiza a leitura de uma matriz.

Uma métrica de desempenho utilizada para medir a eficiência do S.O. quanto ao uso da memória *cache* é chamada *hit ratio.* Tal métrica é baseada em dois eventos: *cache hit* e *cache miss*. Quando uma instrução em execução faz referência a um item e o encontra na memória *cache*, esse evento é definido como um *hit*; quando o item não é encontrado na *cache*, esse evento é definido como *miss*, conforme mostrado na Figura 2. O *hit ratio* é obtido dividindo a quantidade de eventos do tipo *cache hit* pela quantidade total de operações de busca de dados.

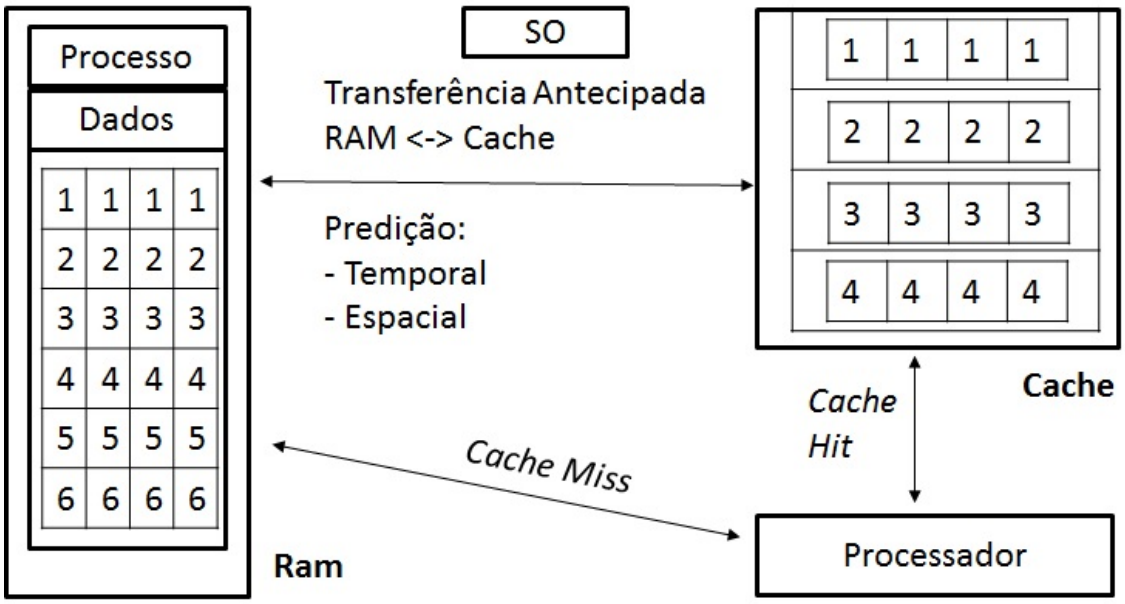


Figura 2. **Sistema de memória em múltiplos níveis.**

Diversos aspectos da codificação influenciam o *hit ratio* do S.O. durante a execução do programa. Um desses aspectos é o *stride* de um laço, que é o padrão de acesso aos dados em memória, o qual pode ser definido usando três padrões:

- O primeiro é chamado *unit-stride, e* indica que os elementos de uma matriz são referenciados um após o outro.
- O segundo é chamado *constant-stride*, e indica que os elementos de *uma matriz*, são referenciados em intervalos regulares.
- O terceiro é chamado aleatório, e indica que os dados são referenciados em intervalos desconhecidos.

A utilização de cada um desses *strides* influencia o *hit ratio* de acesso ao *cache*. A Figura 3 ilustra as transferências realizadas entre memória *cache* e registrador da unidade de processamento utilizando as três formas apresentadas. No caso do *unit-stride*, os dados são encontrados sequencialmente na *cache*. O padrão *constant-stride* possui um desempenho inferior ao *unit-stride*, pois parte dos dados carregados na *cache* não são acessados e portanto a taxa de *cache hit* diminui. O padrão aleatório apresenta o pior desempenho entre eles pois a predição do S.O. apresenta uma taxa baixa de acerto.

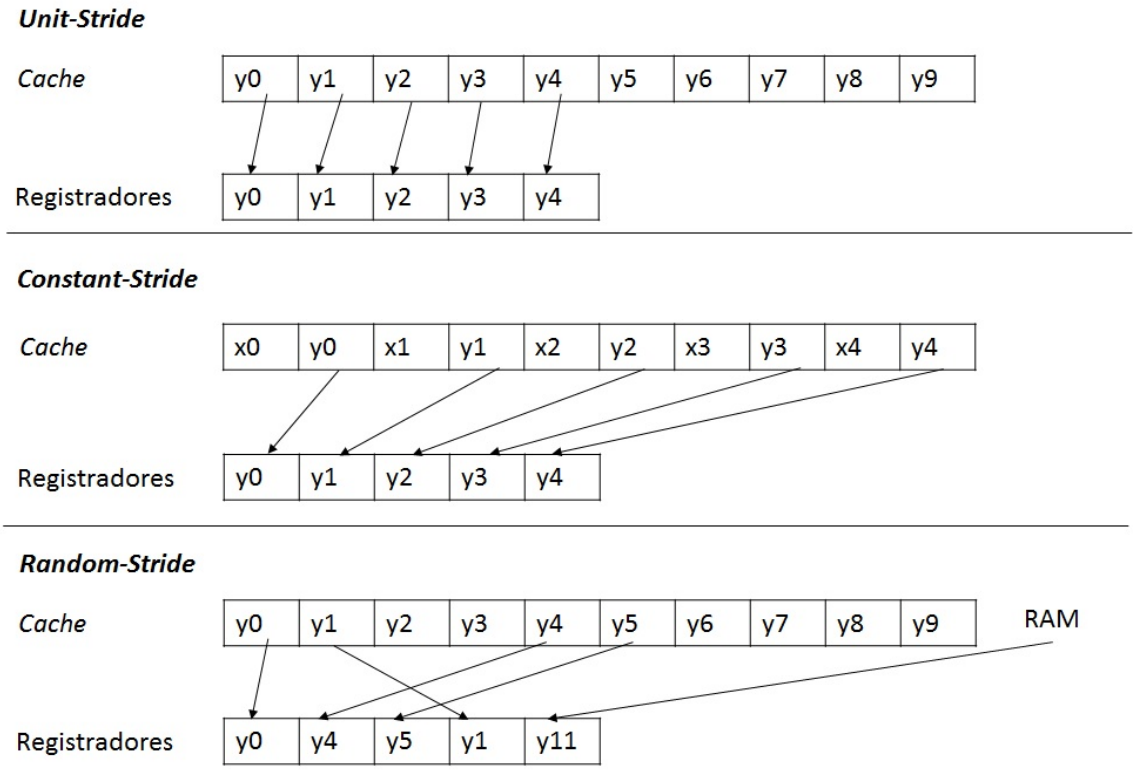


Figura 3. **Transferências de dados entre memória cache e registradores usando diferentes Strides**

A Seção 0 apresenta técnicas para definir estruturas de dados que melhoram o *hit ratio* de *cache* do S.O..

4.2.2. Unidade de Processamento Vetorial

A microarquitetura da maioria dos processadores atuais permite executar dois tipos de instruções: escalares ou vetoriais [7]. Instruções escalares aplicam uma operação em um par de operandos, e instruções vetoriais aplicam uma operação em um conjunto de pares de operandos.

A execução de instruções vetoriais é realizada em unidades de processamento vetorial, que possuem um grande número de registradores para armazenar todos os dados que serão utilizados por uma mesma instrução, conforme mostrado na Figura 4. Nas arquiteturas Intel diversos conjuntos de instruções vetoriais têm sido desenvolvidos, tais como MMX (*Multimedia Extensions*), SSE (*Streaming SIMD Extensions*), AVX (*Advanced Vector Extensions*), AVX-2, AVX512 e IMCI (*Initial Many Core Instructions*).

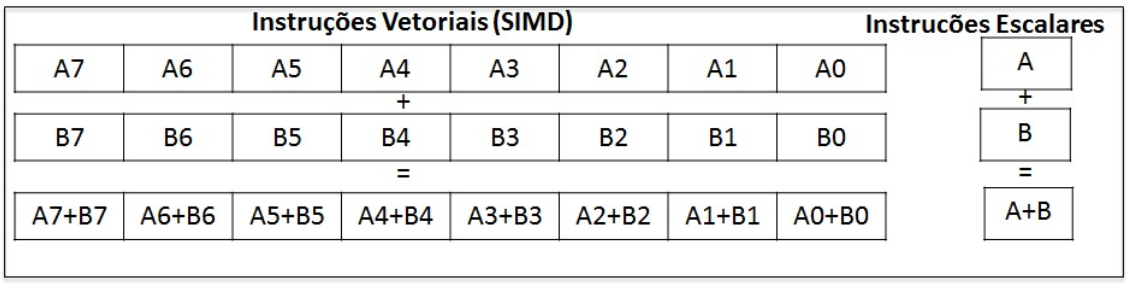


Figura 4. **Execução de instruções vetoriais e instruções escalares.**

As estruturas de código mais apropriadas para serem executadas em unidades de processamento vetorial são os laços, pois definem uma mesma região de código que é executada para um conjunto de valores. Nesse contexto, um requisito para que um laço seja executado utilizando instruções vetoriais é que as iterações não possuam dependências entre si. Três estratégias podem ser adotadas para utilizar instruções vetoriais [9]:

- Vetorização automática: é realizada pelo compilador durante o processo de compilação, e consiste em trocar instruções escalares por instruções vetoriais, sempre que houver garantias de que não haverá alterações no resultado final;
- Vetorização guiada: é realizada por meio de extensões das linguagens de programação, que podem ser colocadas pelo desenvolvedor antes de laços específicos para forçar e/ou parametrizar a vetorização desse laço. Nesses casos, o desenvolvedor é responsável por falhas ou erros no resultado final;
- Vetorização explícita: consiste na utilização de recursos da linguagem de programação ou bibliotecas específicas para desenvolver código que será executado nas unidades de processamento vetorial.

Um exemplo do uso de unidades de processamento vetorial pode ser verificado no laço apresentado no Código 1. Nesse código, a vetorização pode ser aplicada carregando um intervalo de elementos das três matrizes presentes nessa operação, como

por exemplo 4 elementos por vez (A[0..3], B[0..3] e C[0..3]), nos registradores vetoriais e aplicar a operação de adição em todos os elementos.

```
for (i=0; i<n; i++)
  A[i]= B[i]+ C[i];
```

Código 1. **Exemplo de um laço sem dependências entre iterações.**

Alguns dos aspectos que fazem com que a vetorização automática não seja realizada são:

- Dependências assumidas: o compilador avalia para cada laço do código se as iterações são independentes entre si. Nos casos em que o compilador não consegue garantir que as iterações são independentes, o compilador assume que o laço possui dependências;
- Vetorização ineficiente: nos laços em que o compilador consegue garantir que as iterações são independentes, é feita uma análise de estimativa de ganho de desempenho; caso a estimativa de ganho seja baixa o compilador não utiliza instruções vetoriais;
- Laços com número incerto de iterações;

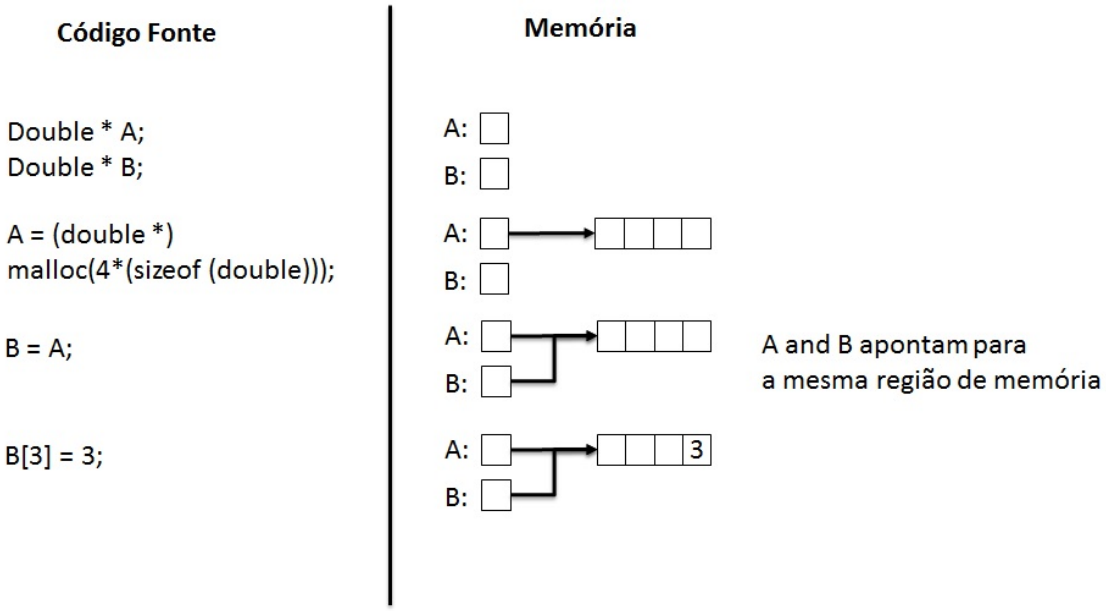


Figura 5. **Duas variáveis apontando para a mesma região de memória.**

- Ponteiros que potencialmente podem acessar uma mesma região de memória (*aliasing*): essa situação é ilustrada na Figura 5, e pode ocorrer quando um laço

está dentro de uma função que manipula ponteiros passados como argumento por exemplo. Nessas situações o compilador não é capaz de garantir que os diferentes ponteiros vão apontar para posições distintas da memória e, portanto, não vetoriza automaticamente.

O Código 2 apresenta dois exemplos de laços que não conseguem ser vetorizados automaticamente. O primeiro laço apresenta uma dependência: a iteração atual atualiza o valor de uma posição da matriz A (**A[i]**), que depende do valor calculado na iteração anterior (**A[i-1]**). No segundo laço os índices dos vetores A e B que serão utilizados na operação são obtidos a partir de valores de outras variáveis; essa referência indireta ao índice da matriz que será atualizado apresenta baixo desempenho quando vetorizado, e em geral o compilador utiliza instruções escalares.

```
for (i=1; i<n; i++)
  A[i]= A[i-1]+ B[i]*randV;
for (i=1; i<n; i++)
  A[B[i]]= B[C[i]]+randV;
```

Código 2. **Exemplos de laços que não são vetorizados automaticamente.**

4.3. Fluxo Iterativo de Vetorização

Uma metodologia para facilitar a otimização de desempenho considerando diferentes estratégias de paralelização pode ser implementada por meio da utilização de ferramentas de perfilamento, que têm como objetivos realizar medições quanto ao tempo de execução de cada linha de código,e quanto à eficiência de uso dos recursos de *hardware*. Tais informações são essenciais para descobrir as oportunidades de paralelismo, que permitem avaliar quais estratégias podem ser adotadas para aproveitar tais oportunidades.

A ferramenta de perfilhamento Intel Advisor provê suporte à vetorização por meio das seguintes análises [10], [11],[12]:

- *Survey target*: medição do tempo de execução e informação sobre vetorização de cada laço, tais como, conjunto de instruções usado e estimativa de ganho;
- *Find trip count*: medição da quantidade de iterações de cada laço;
- *Check dependencies analysis*: verificação quanto à existência de dependências entre iterações de um laço;

- *Check memory access patterns*: análise do padrão de acesso a memória realizado por um laço, utilizando *strides* como parâmetro;

Com base nas análises realizadas pelo Intel Advisor é proposto um fluxo, apresentado na Figura 6, para vetorizar uma aplicação. O objetivo desse fluxo é de modo iterativo identificar as possibilidade de vetorização, aplicar otimizações e avaliar os resultados de ganho de desempenho obtidos.

4.4. Otimização de Acesso à Memória

Um aspecto importante para explorar a vetorização é melhorar o desempenho da transferência de dados entre a memória principal e as unidades de processamento vetorial. Conforme mostrado na Seção 0, nas arquiteturas *multicore* e *manycore* da Intel o recurso que possibilita melhorar o desempenho dessas transferências é a memória *cache*. Nessa seção será apresentado um modelo para definir estruturas de dados que favoreçam o uso eficiente da memória *cache*.

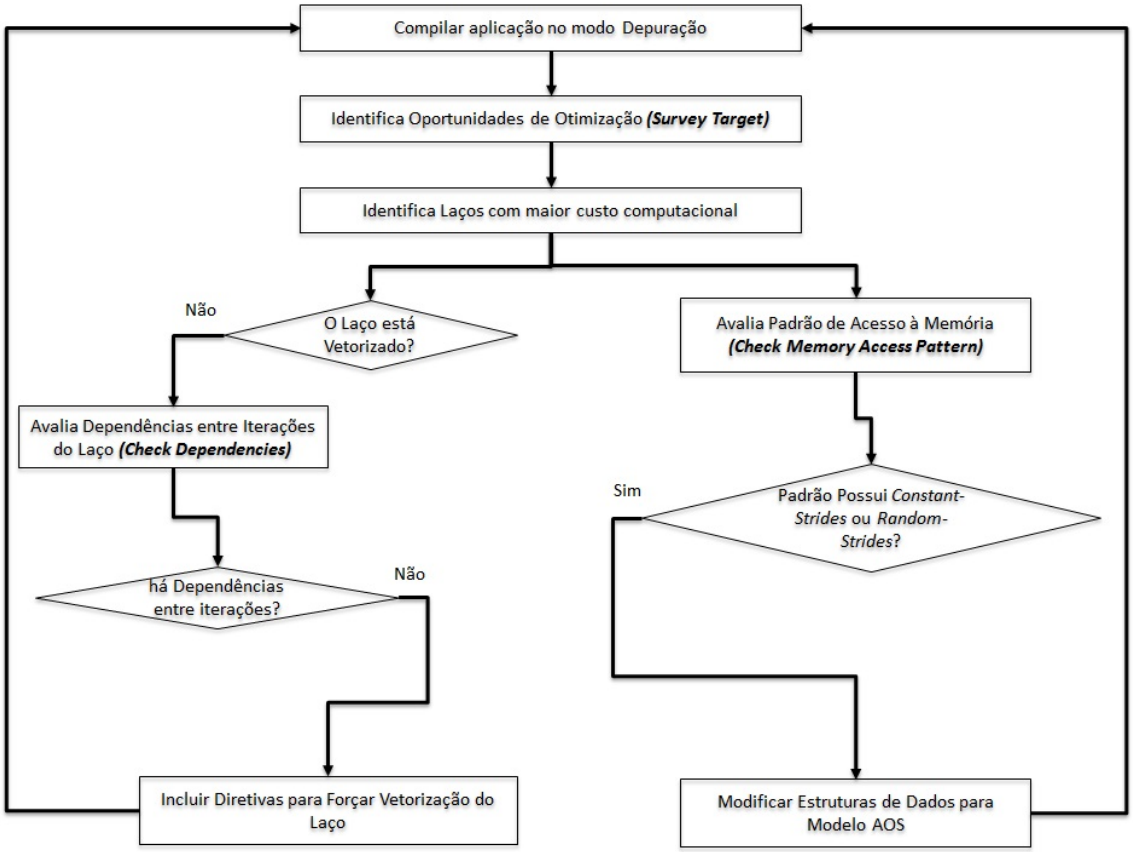


Figura 6. **Fluxo iterativo de vetorização usando Intel Advisor.**

As estruturas de dados em um programa podem ser declaradas de acordo com dois modelos: AOS (*Array of Structures*) ou SOA (*Structure of Arrays*). O modelo AOS é definido como uma estrutura declarada como uma matriz composta por variáveis simples. O modelo SOA é definido como uma única estrutura composta por vetores.

A Tabela 1 apresenta a implementação de uma estrutura para representar coordenadas geográficas em três dimensões, utilizando três variáveis do tipo *float* (x,y e z). O código à esquerda mostra a implementação seguindo o modelo AOS, e o código à direita mostra a implementação seguindo o modelo SOA.

Tabela 1. **Duas implementações de uma mesma estrutura de dados usando modelo SOA e AOS**

struct coordinate {
  float x, y, z;
} crd[N];
	struct coordinate {
	  float x[N], y[N], z[N];
	} crd;

O armazenamento dos dados de um vetor de estruturas declarado usando o modelo AOS é feito colocando uma estrutura completa ao lado da outra. Nesse contexto, um laço que manipula dados dessa estrutura segue o padrão *constant-stride*, pois os dados de x, y e z estão intercalados. No modelo SOA o armazenamento de cada vetor da estrutura é feito em espaços contíguos de memória *cache*, o que favorece que os laços que manipulam dados dessa estrutura sigam o modelo *unit-stride* [13].

O Código 3 a seguir mostra a implementação da estrutura de coordenadas descrita na Tabela 1 seguindo o modelo AOS e SOA, e dois laços que executam o mesmo conjunto de instruções; um deles recebe como entrada a estrutura que representa coordenada implementada de acordo com o modelo AOS, e o outro laço a mesma estrutura implementada de acordo com o modelo SOA.

```
struct coordinate {
  float x, y, z;
} aosobj[60000];

struct coordinate2 {
  float x[60000], y[60000], z[60000];
} soaobj;
…
    for(i=0; i<60000; i++) {
      aosobj[i].x=i*j- randV;
      aosobj[i].y=i+j* randV;
```

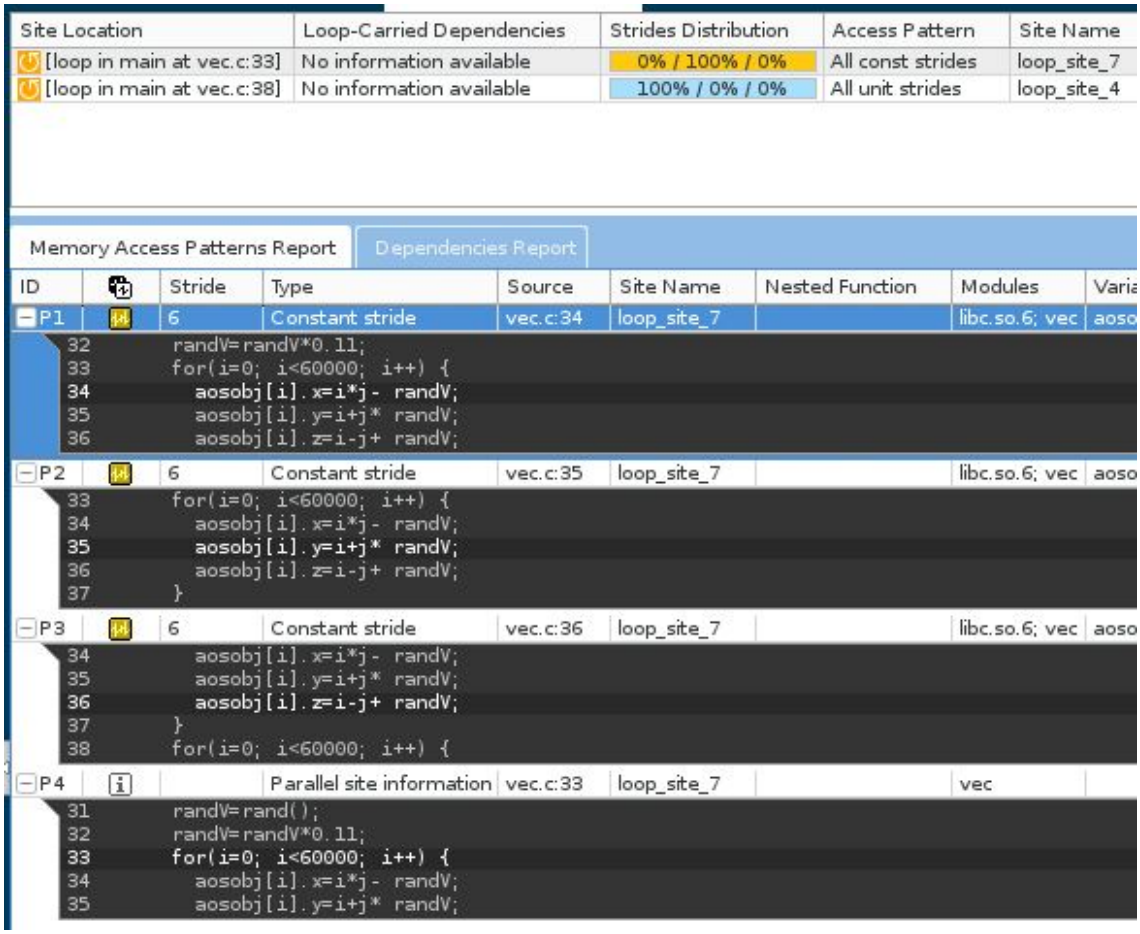
```
    aosobj[i].z=i-j+ randV;
  }
  for(i=0; i<60000; i++) {
    soaobj.x[i]=i*j- randV;
    soaobj.y[i]=i+j* randV;
    soaobj.z[i]=i-j+ randV;
  }
…
```

Código 3. **Fragmento de código que implementa um laço que manipula dados de estruturas AOS e SOA**

Ao avaliar esse código com o *check memory access pattern* foi verificado que o padrão de acesso feito no primeiro laço (AOS) é *constant stride*, enquanto no segundo laço (SOA) o padrão de acesso é *unit-stride* (Figura 8). O impacto no desempenho pode ser verificado na Figura 7: o laço *unit-stride* apresenta menor tempo de execução e maior ganho estimado de desempenho.

Loops	Vector Issues	Self Time	Total Time	Loop Type	Why No Vectorization?	Vectorized Loops: Vec...	Efficiency	Gain Estimate
[loop in __libc_start_main]		0.000s (	18.700s	Scalar				
[loop in main at vec.c:30]	1 Data type conversions present	0.000s (	18.700s	Scalar	inner loop was al...			
[loop in main at vec.c:33]	**2 Data type conversions present**	**6.919s**	**6.919s**	**Vectorized (Body)**		SSE2	~82%	**1.65x**
[loop in main at vec.c:38]	2 High vector register pressure	5.906s	5.906s	Vectorized (Body)		SSE2	~70%	2.80x
[loop in main at vec.c:47]	2 High vector register pressure	5.875s	5.875s	Vectorized Versions		SSE2	~78%	3.14x

Figura 7. **Relatório gerado após a análise survey target.**



Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Site Name
[loop in main at vec.c:33]	No information available	0% / 100% / 0%	All const strides	loop_site_7
[loop in main at vec.c:38]	No information available	100% / 0% / 0%	All unit strides	loop_site_4

Memory Access Patterns Report | Dependencies Report

ID	Stride	Type	Source	Site Name	Nested Function	Modules	Varia
P1	6	Constant stride	vec.c:34	loop_site_7		libc.so.6; vec	aoso

```
32      randV=randV*0.11;
33      for(i=0; i<60000; i++) {
34        aosobj[i].x=i*j- randV;
35        aosobj[i].y=i+j* randV;
36        aosobj[i].z=i-j+ randV;
```

P2	6	Constant stride	vec.c:35	loop_site_7		libc.so.6; vec	aoso

```
33      for(i=0; i<60000; i++) {
34        aosobj[i].x=i*j- randV;
35        aosobj[i].y=i+j* randV;
36        aosobj[i].z=i-j+ randV;
37      }
```

P3	6	Constant stride	vec.c:36	loop_site_7		libc.so.6; vec	aoso

```
34        aosobj[i].x=i*j- randV;
35        aosobj[i].y=i+j* randV;
36        aosobj[i].z=i-j+ randV;
37      }
38      for(i=0; i<60000; i++) {
```

P4		Parallel site information	vec.c:33	loop_site_7		vec	

```
31      randV=rand();
32      randV=randV*0.11;
33      for(i=0; i<60000; i++) {
34        aosobj[i].x=i*j- randV;
35        aosobj[i].y=i+j* randV;
```

Figura 8. **Relatório Mostrando a Distribuição de Strides de cada Laço.**

4.5. Explorando Paralelismo de Dados nas Unidades de Processamento Vetorial

Esta seção descreve alguns recursos para explorar o paralelismo de dados nas unidades de processamento vetorial. A seção 0 descreve como utilizar vetorização automática nos compiladores Intel e GCC (GNU *Compiler Collection*). Na seção 0 serão mostrados recursos para permitir a vetorização por meio da desambiguação de ponteiro. Na seção 0 serão apresentadas as extensões presentes nos compiladores Intel e GCC para forçar vetorização e na seção 0, os recursos da especificação OpenMP 4.0 para prover suporte a vetorização guiada.

4.5.1. Vetorização Automática

A vetorização automática é um dos passos de otimização que são realizados pela maioria dos compiladores C, C++ e Fortran. Esta seção descreve como aplicar essa otimização utilizando compiladores Intel e GCC.

Nos compiladores citados, Intel e GCC, esse passo de otimização é configurado utilizando o parâmetro **-O<nível>**. O parâmetro nível pode variar de 0 a 3 e define quais os tipos de otimizações que devem ser aplicadas.

A Figura 9 mostra como compilar um programa, apresentado no Código 4, ativando a opção de vetorização automática no compilador Intel e GCC usando opção **-O3**.

```
icc autovec.c -O3 -o autovec
gcc autovec.c -O3 -o autovecGCC
```

Figura 9. **Comando para compilação de um progrma em C usando icc e gcc.**

```
#include <time.h>
#include <stdio.h>

int main(){
  const int n=90000;
  int i, j, randV;
  int A[n];
  int B[n];

  for (j=0; j<45000; j++) {
    srand(time(NULL));
```

```
   randV=rand();

   for (i=0; i<n; i++)
     A[i]+=B[i]*randV;
  }
}
```

Código 4. **Exemplo de código que pode ser vetorizado automaticamente.**

É possível avaliar quais regiões de código foram vetorizadas por meio do relatório de otimização, que indica quais regiões de código foram vetorizadas e quais não foram vetorizadas. Para os laços que não foram vetorizados, é indicado o motivo por não terem sido vetorizados.

No GCC é necessário utilizar o parâmetro -ftree-vectorizer-verbose=1 para gerar o relatório de otimização. A Figura 10 mostra como compilar o código gerando o relatório de otimização usando GCC, e a Figura 11 apresenta um fragmento do relatório de otimização gerado a partir da compilação do programa descrito no Código 4.

```
gcc autovec.c -O3 -ftree-vectorizer-verbose=1 -o autovecGCC
```

Figura 10. **Comando para compilação usando GCC gerando relatório de otimização.**

```
Analyzing loop at autovec.c:10

autovec.c:10: note: not vectorized: loop contains function calls or data references that cannot be analyzed
autovec.c:10: note: bad data references.
Analyzing loop at autovec.c:14

autovec.c:14: note: Unknown misalignment, is_packed = 0
autovec.c:14: note: Unknown misalignment, is_packed = 0
autovec.c:14: note: virtual phi. skip.

Vectorizing loop at autovec.c:14

autovec.c:14: note: virtual phi. skip.
autovec.c:4: note: vectorized 1 loops in function.
```

Figura 11. **Fragmento do relatório de vetorização gerado pelo GCC.**

No compilador Intel é necessário utilizar o parâmetro -vec-report5 para gerar o relatório de otimização. A Figura 12 mostra como compilar o código gerando o relatório de otimização usando icc, e a Figura 13 apresenta um fragmento do relatório de otimização gerado a partir da compilação do programa descrito no Código 4.

```
icc autovec.c -o autovec -O3 -vec-report5
cat autovec.optrpt
```

Figura 12. **Comando para compilação usando ICC gerando relatório de otimização.**

```
LOOP BEGIN at autovec.c(10,5)
  remark #15542: loop was not vectorized: inner loop was already vectorized

  LOOP BEGIN at autovec.c(14,9)
    remark #15388: vectorization support: reference B has aligned access [ autovec.c(15,18) ]
    remark #15388: vectorization support: reference A has aligned access [ autovec.c(15,13) ]
    remark #15305: vectorization support: vector length 4
    remark #15399: vectorization support: unroll factor set to 4
    remark #15309: vectorization support: normalized vectorization overhead 2.667
    remark #15300: LOOP WAS VECTORIZED
    remark #15449: unmasked aligned unit stride stores: 2
    remark #15475: --- begin vector loop cost summary ---
    remark #15476: scalar loop cost: 4
    remark #15477: vector loop cost: 1.500
    remark #15478: estimated potential speedup: 2.660
    remark #15488: --- end vector loop cost summary ---
  LOOP END
LOOP END
```

Figura 13. **Fragmento do relatório de otimização gerado pelo compilador Intel (icc).**

4.5.2. Desambiguação de Ponteiro

Um dos fatores que impede que o compilador realize a vetorização automática é a possibilidade de que ponteiros passados como parâmetros de função apontem para a mesma região de memória. Um exemplo dessa ambiguidade pode ser observado no Código 5; nesse código a função **add_floats** recebe cinco parâmetros como ponteiro. Nesse contexto o risco de ambiguidade ocorre, pois a função **add_floats** pode ser chamada de outros arquivos que são compilados separadamente; nesses casos o compilador identifica que há um risco de ambiguidade.

```
#include "func.h"

void add_floats(double * a, double * b, double * c, double * d, double * e, int n){
  int i,j;

  for (j=1; j<n-1; j++){
    for (i=1; i<n-1; i++){
      a[i] =e[j-1] + (c[i] * d[j+1]) ;
    }
```

```
  }
}
```

Código 5. **Exemplo de função com possível ambiguidade de ponteiro.**

Quando o desenvolvedor consegue garantir que não há risco de ambiguidades, é possível instruir o compilador a ignorar ambiguidades de ponteiros usando dois mecanismos: **fargument-noalias** e **restrict**.

O argumento **fargument-noalias** indica que nenhum ponteiro passado como parâmetro de nenhuma função possui ambiguidade. Um exemplo do uso desse argumento é mostrado na Figura 14.

```
icc func.c -c -o func.o -O3 -vec-report5 -fargument-noalias
```

Figura 14. **Compilação usando parâmetro fargument-noalias.**

O argumento **restrict** também pode ser utilizado para indicar ao compilador que não há ambiguidades, porém essa indicação deve ser feita para cada variável de cada função. Nesse sentido, é necessário passar o argumento -restrict na compilação, e para cada variável incluir a palavra reservada restrict entre * e o nome da variável. Um exemplo do uso desse parâmetro é mostrado no Código 6. A compilação deve incluir o parâmetro -restrict.

```
…
void add_floats(double * restrict  a, double * restrict  b, double * restrict  c, double * restrict  d, double * restrict  e, int n){
...
```

Código 6. **Exemplo de eliminação de ambiguidades usando parâmetro restrict.**

4.5.3. Extensões de Compilador para Vetorização

A maioria dos compiladores C, C++ e Fortran disponibiliza extensões para prover suporte à vetorização, que também são denominadas diretivas de vetorização. Tais diretivas podem ser utilizadas para instruir o compilador a forçar e parametrizar a vetorização de um laço. Nessa seção serão descritas três diretivas: **ivdep**, **simd** e **declspec(vector)**.

A diretiva **ivdep** tem como objetivo instruir o compilador a ignorar dependências de vetor. O Código 7 mostra um exemplo de uso dessa diretiva.

Sintaxe:

```
#pragma ivdep
Declaração de laço
```

```
#pragma ivdep
for (i = 0; i < N; i++)
   c[i] = a[i] + b[i];
```

Código 7. **Exemplo de uso da diretiva ivdep.**

A diretiva **simd** é similar à diretiva **ivdep**, porém implementa métodos mais agressivos de otimização e possui argumentos para controlar a vetorização. Um exemplo do uso dessa diretiva é apresentado no Código 8.

Sintaxe:

```
#pragma simd [Cláusulas]
Declaração de laço
```

Algumas cláusulas da diretiva **simd** são descritas a seguir:

- vectorlength(*n1*[,…]) : Assume que a vetorização é segura para um certo tamanho de vetor;
- vectorlengthfor(*dtype*) : Assume que a vetorização é segura para um certo tipo de dados;
- linear (variável[:passo-linear],...) : define que a variável é incrementada a cada passo do laço, e que o valor do incremento atribuído à variável a cada passo é igual ao valor passado no parâmetro passo-linear;
- [no]assert: gera mensagens de avisos ou de erros quando a vetorização falha.

```
#pragma simd linear(m:1)
for (i=0; i<20; i++) {
  a[i] = a[i]+m;
  m++;
}
```

Código 8. **Exemplo de uso da diretiva SIMD.**

A diretiva **__declspec(vector)** tem como objetivo instruir o compilador a utilizar instruções vetoriais para compilar uma função, que é chamada a partir do corpo de um laço. É recomendado utilizar essa diretiva em funções com poucas linhas de código. O Código 9 mostra um exemplo de uso dessa diretiva.

Sintaxe:

```
__declspec(vector)
Definição ou declaração de função
```

```
__declspec(vector)
void v_add2(float c, float a, float b)
{
   c = a + b;
}

void v_addMain(float *c, float *a, float *b, int N)
{
  int i;
  for (i = 0; i < N; i++)
    v_add2(c[i], b[i], a[i]);
}
```

Código 9. **Exemplo de uso de uma diretiva para prover suporte à vetorização de função.**

4.5.4. Diretivas do OpenMP 4 para Vetorização

A especificação do OpenMP a partir da versão 4.0 passou a disponibilizar as seguintes diretivas para prover suporte à vetorização: **omp simd** e **omp declare simd**.

A diretiva **omp simd** define que o laço marcado deve ser compilado usando instruções vetoriais, e funciona de modo similar a diretiva **pragma simd** apresentada na Seção 0.

Sintaxe:

```
#pragma omp simd[clause[[,] clause],…]]
Laço
```

A seguir são descritas as cláusulas que podem ser usadas em conjunto com as diretiva **omp simd**:

- safelen (tamanho) : o parâmetro tamanho define o número máximo de iterações do laço que podem ser executadas concorrentemente sem quebrar uma dependência entre as iterações;
- aligned(variável[:alinhamento]) : define que a variável foi previamente alinhada em memória, e o parâmetro alinhamento define o tamanho do bloco de alinhamento;

- collapse (n) : especifica o número de laços aninhados (parâmetro **n**) que serão agrupados em um único laço.

Além dessas, podem ser utilizadas também as cláusulas descritas na Seção 0. Um exemplo do uso dessa diretiva é mostrado no Código 10.

```
#pragma omp simd
for (i = 0; i < N; i++)
   c[i] = a[i] + b[i];
```

Código 10. **Exemplo de uso da diretiva omp simd.**

A diretiva **omp declare simd** possui o mesmo objetivo da diretiva **__declspec(vector)** descrita na Seção 5.3, porém permite utilizar cláusulas para controlar o processo de vetorização.

Sintaxe:

```
#pragma omp declare simd [clause[[,] clause],…]]
Definição ou declaração de função
```

A seguir são descritas algumas cláusulas que podem ser usadas em conjunto com a diretiva **omp declare simd**:

- uniform(variável): define que o valor da variável se mantém constante para todas as chamadas concorrentes da função;
- simdlen(tamanho): o parâmetro tamanho define a quantidade máxima de elementos de cada argumento da função, que podem ser carregados por intruções SIMD;
- inbranch: define que a função pode ser chamada a partir de estruturas condicionais no laço;
- notinbranch: define que a função nunca é chamada a partir de estruturas condicionais no laço.

Além dessas, podem ser utilizadas também as cláusulas descritas na diretiva **omp simd**: aligned, simdlen, linear, reduction e uniform.

O Código 11 mostra como utilizar essa diretiva em um código que realizar interpolação de matrizes [14, p. 22].

```
#pragma omp declare simd
int FindPosition(double x) {
   return (int)(log(exp(x*steps)));
}
#pragma omp declare simd uniform (vals)
double Interpolate(double x, const point* vals)
{
   int ind = FindPosition(x);
   ...
   return res;
}
int main ( int argc , char argv [] )
{
   . . .
  for ( i=0; i <ARRAY_SIZE;++ i ) {
      dst[i] = Interpolate( src[i], vals ) ;
   }
   . . .
}
```

Código 11. **Exemplo de vetorização de função usando OpenMP 4.0.**

4.5.5. Avaliação de Dependências entre Iterações de um Laço

Um pré-requisito essencial para utilizar diretivas que forçam a vetorização de laços, é garantir que os laços não apresentem dependências entre as iterações. Uma forma simples de identificar se um laço possui tais dependências é usando a análise *check dependencies* do Intel Advisor. Um exemplo do uso dessa análise para suporte à vetorização será mostrado usando como exemplo o programa descrito no Código 12, que apresenta a versão serial de um código que realiza a multiplicação entre duas matrizes bidimensionais (A e B) e armazena o resultado em uma terceira matriz C.

```
void multiply(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE
c[][NUM], TYPE t[][NUM])
{l
  int i,j,k;
  for(i=0; i<msize; i++) {
    for(k=0; k<msize; k++) {
      for(j=0; j<msize; j++) {
                                    c[i][j] = c[i][j] + a[i][k] * b[k][j];
      }
    }
  }
}
```

Código 12. **Código Sequencial para Multiplicação de Matrizes.**

O resultado da análise *survey target* dos laços descritos no Código 12 indica que os três laços estão sendo executados de modo escalar. Nesse sentido, o laço mais interno foi selecionado para ser avaliado pelo *check dependencies analysis*. A análise confirma que não há dependências entre as iterações; isso indica que é seguro vetorizar esse laço.

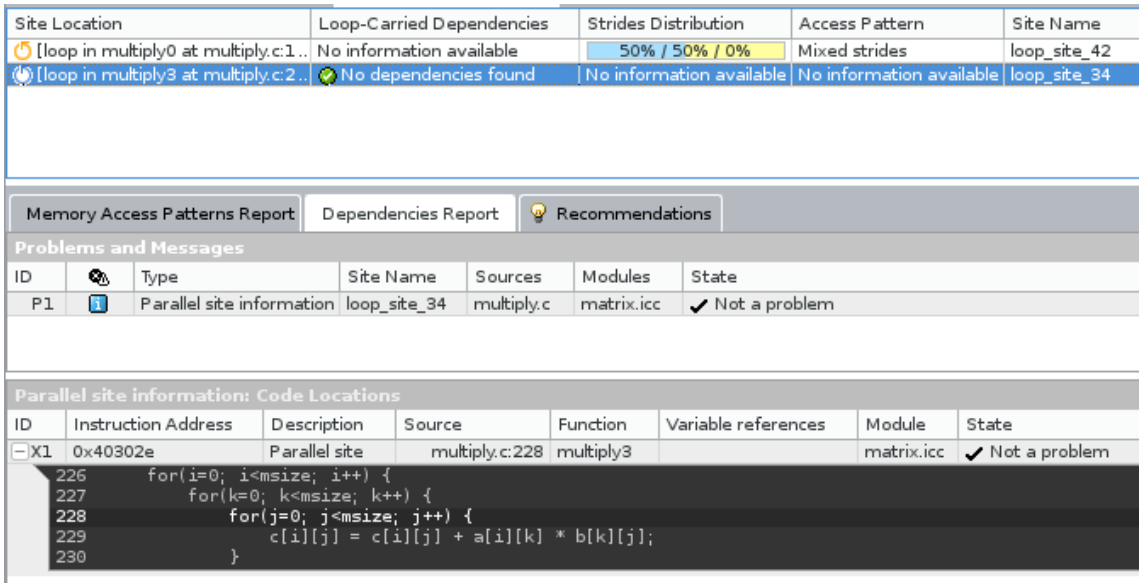


Figura 15. **Resultado da Análise de Dependências entre Iterações de Laços.**

A diretiva **pragma simd** foi colocada no laço mais interno conforme mostrado no Código 13, e a análise *survey target* foi executada novamente. O resultado indica que o laço foi vetorizado pelo compilador.

```
void multiply(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE
c[][NUM], TYPE t[][NUM])
{
 int i,j,k;
 for(i=0; i<msize; i++) {
  for(k=0; k<msize; k++) {
   #pragma simd
   for(j=0; j<msize; j++) {
    c[i][j] = c[i][j] + a[i][k] * b[k][j];
   }
  }
 }
}
```

Código 13. **Código vetorizado para Multiplicação de Matrizes.**

4.6. Consideração Finais

Arquiteturas paralelas híbridas disponibilizam múltiplos níveis de paralelismo que podem ser explorados de modo combinado. Os recursos de processamento vetorial representam o nível mais básico de paralelismo. As técnicas de vetorização representam o conjunto de mecanismos disponibilizados pelos compiladores, extensões das linguagens e até mesmo bibliotecas especializadas como o Intel Intrisincs.

Nesse minicurso foram apresentados, de forma introdutória – com exemplos de códigos fontes e aplicações reais -, recursos básicos para explorar a vetorização automática oferecida pela maioria dos compiladores C, C+ e Fortran, bem como extensões da linguagem C e funcionalidades do OpenMP para explorar vetorização guiada.

Referências

[1] A. Heinecke, M. Klemm, and H.-J. Bungartz, “From GPGPU to Many-Core: Nvidia Fermi and Intel Many Integrated Core Architecture,” *Comput. Sci. Eng.*, vol. 14, no. 2, pp. 78–83, 2012.

[2] G. E. Blelloch, *Vector Models for Data-parallel Computing*. Cambridge, MA, USA: MIT Press, 1990.

[3] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 4.0*. 2013.

[4] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems,” *IEEE Test*, vol. 12, no. 3, pp. 66–73, May 2010.

[5] S. Wienke, P. Springer, C. Terboven, and D. an Mey, “OpenACC: First Experiences with Real-world Applications,” in *Proceedings of the 18th International Conference on Parallel Processing*, Berlin, Heidelberg, 2012, pp. 859–870.

[6] “Intel Intrinsics Guide.” [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide [Accessed 16-Sep-2016].

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

[8] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, 1 edition. Boca Raton, FL: CRC Press, 2010.

[9] A. Vladimirov, R. Asai, and V. Karpusenko, *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*, 2nd edition. Colfax International, 2015.

[10] "Intel® Advisor | Intel® Software." [Online]. Available: https://software.intel.com/en-us/intel-advisor-xe [Accessed 16-Sep-2016].

[11] "Get a Helping Hand from the Vectorization Advisor | Intel® Software." [Online]. Available: https://software.intel.com/articles/get-a-helping-hand-from-the-vectorization-advisor [Accessed 16-Sep-2016].

[12] "Guided Code Vectorization with Intel® Advisor XE - Colfax Research." [Online]. Available: http://colfaxresearch.com/guided-code-vectorization-with-intel-advisor-xe/ [Accessed 16-Sep-2016].

[13] "Memory Layout Transformations | Intel® Software." [Online]. Available: https://software.intel.com/en-us/articles/memory-layout-transformations [Accessed 13-Sep-2016].

[14] G. M. Raskulinec and E. Fiksman, "Chapter 22 - SIMD Functions Via OpenMP," in *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*, vol. 2, J. Reinders and J. Jeffers, Eds. Boston, MA, USA: Morgan Kaufmann, 2015, pp. 421–440.

Capítulo 5

Computação Heterogênea com GPUs e FPGAs

Ricardo Ferreira[2], José Augusto Miranda Nacif
Departamento de Informática
Universidade Federal de Viçosa

Resumo

As GPUs e as FPGAs são soluções complementares para computação de alto desempenho com eficiência energética. Este trabalho irá enfatizar as tendências, metodologias, plataformas e ferramentas para as duas arquiteturas alvo. As GPUs podem ser programadas com os modelos CUDA e OpenCL. Outra alternativa para GPUs é fazer uso das bibliotecas e do modelo OpenACC. As tendências em Deep Learning e Redes Complexas bem como a evolução das arquiteturas das GPUs também serão apresentadas. Recentemente, novas plataformas com FPGAs para computação de alto desempenho foram propostas. Os resultados preliminares das novas plataformas de FPGA e as tendências para alto desempenho serão discutidas. Finalmente, uma comparação entre FPGA e GPU enfatizando as características, vantagens e desvantagens das duas abordagens será apresentada.

Abstract

GPUs and FPGAs are promising architectures for energy efficient high performance computing. This work aims to present the most recent developments regarding the two target architectures. First, we present a brief review of CUDA and OpenCL, which are frameworks designed for GPUs. Next, we discuss libraries and OpenACC, since they are also two efficient ways to achieve high performance by using GPUs. Recently, deep learning methods have been successfully applied to a variety problems, and we show the GPU deep learning approaches. Even with a low clock frequency compared to

[2] Auxílio Financeiro: Fapemig Universal, Intel Labs, Intel Brasil, Nvidia Education Center, Capes, Cnpq, Gapso-Accenture

GPUs, FPGAs can achieve high performance by exploiting spatial and temporal parallelism. Preliminary results on novel FPGA accelerators are reported. Finally, we point out main advantages and weakness for both accelerator architectures.

5.1. Introdução

Este trabalho apresenta duas soluções para programação paralela com alto desempenho com eficiência energética: GPUs e FPGAs. As GPUs, com apenas uma década de história, já geraram mais de 60 mil artigos científicos publicados com aplicações de alto desempenho. O modelo de programação segue o paradigma SIMT (*single instruction multiple threads*). As grandes inovações das GPUs são: a capacidade de gerenciar milhões de *threads*, a eficiência energética, a disponibilidade de várias opções de memória com alto desempenho e a regularidade da arquitetura com uma estrutura complementar às CPUs. Primeiramente, se faz necessário compreender a arquitetura da GPU e o seu modelo de programação como uma extensão de C/C++, que facilitou a sua rápida popularização. A difusão das GPUs como placas de vídeos, possibilitou a conquista do mercado, levando a uma redução de custo e uma alta disponibilidade para os programadores. As aplicações com o perfil mais adequado para paralelização em GPUs foram rapidamente codificadas como algoritmos para processamento de vídeos, imagens, vetores e matrizes. Atualmente é necessário um conhecimento mais aprofundado da arquitetura das GPUs para mapeamento das aplicações. Algumas vezes, este mapeamento não é trivial. Para auxiliar na modelagem, várias bibliotecas com código otimizado e uma ampla gama de funcionalidades estão disponíveis. Outra opção é o OpenACC, que é esforço conjunto das empresas Cray, Caps, Nvidia e PGI, que permite utilizar simples anotações para paralelizar um código C/C++ sem conhecimentos aprofundados de GPU. Além dessas abordagens, os fenômenos IoT (*Internet of Things*) e *big data*, criaram novas oportunidades de implementação com outros modelos em alto nível para uso das GPUs, como por exemplo, o uso de redes neurais com muitas camadas (*Deep Learning*). Plataformas para aprendizado de máquina baseadas em *Deep Learning* [Bahrampour, 2015] como o Caffe [Jia, 2014] disponível em [Caffe, 2016] e o Torch [Torch, 2016] vem ampliando o número de aplicações usando GPUs sem a necessidade de codificação em linguagens convencionais de programação.

Os FPGAs se apresentam como uma outra alternativa com eficiência energética e desempenho, excelente relação preço/desempenho com um hardware regular que pode oferecer alta densidade de computação paralela, tanto em termos espaciais como também em termos de paralelismo temporal (*pipeline*). Em comparação com as CPUs para *big data*, como por exemplo a ferramenta de Bing da Microsoft, a solução em FPGA [McMillan, 2014] pode ser até 40 vezes mais rápida. Outro fato recente, foi a aquisição da empresa Altera (maior fabricante de FPGAs) pela Intel para o desenvolvimento de plataformas de alto desempenho com uma expectativa que um terço dos servidores de nuvem em 2020 serão baseados em FPGAs [Iordache, 2016]. Comparado com CPUs multi-core superescalares, os FPGAs proporcionam 2 vezes mais desempenho com redução de energia [Cong, 2016].

5.2. GPUs

As *graphics processing units* (GPUs) foram projetadas para proporcionar um hardware eficiente para processamento de vídeo. Sua arquitetura básica oferece centenas de unidades de processamento, memória com uma alta vazão e *pipeline* para processamento vetorial. Essas características motivaram os pesquisadores a modelar outros problemas além das aplicações para processamento gráfico. No final de 2006, as plataformas CUDA e OpenCL surgiram para possibilitar um modelo de programação paralela genérico para uso das GPUs como unidades de processamento paralelo de finalidade geral. Os supercomputadores mais rápidos na lista do Top500 incorporam aceleradores e a expectativa que os aceleradores estejam presentes em todas as máquinas. Este fenômeno gera uma demanda por ambientes e modelos para programação onde OpenCL e CUDA são os padrões mais utilizados [Kim,2015]. Além das GPUs, ferramentas para OpenCL visam gerar código para outras plataformas como DSPs, Intel Xeon Phi e FPGAs.

Para GPUS, outras alternativas também foram propostas como o uso de bibliotecas, o modelo OpenACC e mais recentemente as plataformas para *Deep Learning*. Inicialmente, esta seção irá detalhar as plataformas CUDA e OpenCL bem como o modelo de programação e as arquiteturas das GPUs. Posteriormente, as seções 5.3 e 5.4 irão abordar as bibliotecas, o modelo OpenACC e as plataformas de *Deep Learning*.

5.2.1. CUDA

CUDA é uma plataforma de computação paralela e um modelo de programação introduzido pela empresa *Nvidia* no final de 2006 para possibilitar o desenvolvimento de aplicações de finalidade geral de uma forma simples programando as GPUs com uma extensão de C. Este texto apresenta as linhas gerais, para uma apresentação detalhada da plataforma CUDA o leitor pode consultar os livros [Sanders, 2010] e [Cheng, 2014] que ilustram de forma didática apresentando vários exemplos de código.

A abordagem proposta pela plataforma CUDA é baseada em uma extensão das linguagens de programação C/C++. Com chamada de funções e palavras chaves, a extensão permite ao programador C/C++ expressar um paralelismo massivo com milhares de threads que serão diretamente mapeados pelo compilador. CUDA também oferece suporte para outras linguagens como Python [PyCuda, 2016], Fortran [CUDA Fortran, 2016], dentre outras. As principais primitivas de CUDA fazem a alocação de memória da GPU com os métodos C*udaMalloc* e outros, a transferência de dados entre a CPU e a GPU com o método *CudaMemcpy* e a execução paralela de milhares de *threads* com a chamada da função seguida pela expressão <<< *b*, *t* >>>, que irá disparar b*t cópias da função onde b é o número de blocos com t *threads* cada. Outra primitiva muito utilizada é a palavra chave *shared* que é reservada para alocação da memória compartilhada da GPU que pode funcionar como uma cache gerenciada pelo próprio programador.

5.2.2. OpenCL

A extensão *Open Computing Language* (OpenCL) é também uma plataforma para programação paralela em sistemas heterogêneos compostos por CPUs, GPUs, DSPs, FPGAs e outros aceleradores. OpenCL é uma extensão de C (baseado em C99) para programação e desenvolvimento de interface (APIs). OpenCL oferece um padrão independente do fabricante do acelerador para programação paralela a nível de tarefas e dados. OpenCL é mantido pelo consórcio *Khronos* com apoio de várias empresas como por exemplo Altera, AMD, Apple, ARM Holdings, Creative Technology, IBM, Imagination Technologies, Intel, Nvidia, Qualcomm, Samsung e Xilinx.

A primeira versão foi lançada em Agosto de 2009 e em março de 2015 foi lançada a versão 2.1. Apesar de ser semelhante a CUDA, OpenCL é bem mais genérico em termos de dispositivos alvos. Conceitualmente, as principais semelhanças são: chamada de função com múltiplas cópias, organização dos *threads* em blocos, alocação e transferência de dados entre a CPU e o dispositivo acelerador. OpenCL garante portabilidade, uma vez que o código é compilado em tempo de execução para o dispositivo alvo. A versão 2.1 já possibilita que os fabricantes de dispositivos aceleradores forneçam bibliotecas proprietárias com portabilidade. O leitor pode consultar o livro [Banger, 2013] para um introdução à plataforma OpenCL ilustrada de forma didática com auxílio de vários exemplos de código.

5.2.3. Comparação entre CUDA e OpenCL

A Tabela 5.1 apresenta as principais semelhanças e diferenças entre a terminologia e as extensões da linguagem C (qualificadores) de CUDA e OpenCL. Ambos têm um modelo de programação semelhante no qual a chamada de função é acompanhada de dois parâmetros básicos: número de blocos e *threads*. Em CUDA, os *threads* (ou *work-item* em OpenCL) são organizados em blocos (*work-group* em OpenCL). Dentro do bloco, os *threads* podem se comunicar através da memória compartilhada (*shared* ou *local*). Cada *thread* tem um identificador único que é a tupla (*t,b*) onde *t* é identificador do *thread* e *b* é o identificador do *bloco*. Em OpenCL existe a função get_global_id() que retorna um identificador único (ID) para cada instância de *thread (work-item)*. Em CUDA, o mais usual é o programador gerar o ID com a equação *ID = b * TamanhoBloco + t*. Esta mesma equação é usada pela função get_global_id() do OpenCL. Um princípio básico de programação é usar o ID do *thread* como indexador para que cada *thread* faça a sua parte da tarefa. Por exemplo, em uma soma paralela de vetores, cada *thread* pode usar seu ID para acessar um elemento diferente do vetor e todos trabalharem em paralelo com elementos diferentes.

Tabela 1.1 - Semelhanças e diferenças entre CUDA e OpenCL

Tópicos	**CUDA**	**OpenCL**
Termos	*Thread*e*Bloco*	*Work-item* e *Work-group*
	Global memory e *Constant memory*	*Global memory* e *Constant*

		memory
	Shared memory e *local memory*	*Local memory* e *Private memory*
Qualificadores	*__global__*função	*__kernel*função
	__device__ variável global dispositivo	__global variável global dispositivo
	__shared__ variável memória compartilhada	__local variável memória compartilhada
Indexação	threadIdx	get_local_id()
	blockIdx	get_group_id()
	--	get_global_id()

Fonte: Elaborado pelos autores

A Figura 5.1a ilustra um exemplo de programação no modelo CUDA. O código ilustrado destaca as partes que são executadas na CPU e as partes que serão executadas na GPU, além da transferência explícita de dados entre a CPU e GPU. Em OpenCL, o código é maior por ser mais genérico. O código OpenCL na CPU envolve a busca do contexto que determina quais são os dispositivos que estão disponíveis, a geração de código para o dispositivo. Posteriormente, os modelos são semelhantes com os passos: alocação de memória, transferência de dados, execução paralela de *kernel* e liberação do espaço alocado em memória. A figura 5.1b ilustra apenas o trecho de código para o kernel em OpenCL, no qual cada *thread* obtém seu identificador global através da função get_global_id() e executa a operação de forma paralela sobre cada componente do vetor.

Durante a execução, cada sub-grupo de 32 *threads* executa ao mesmo tempo, denominado pelo termo *warp*, em CUDA. Porém, os *warps* podem executar em qualquer ordem. Por exemplo, suponha que sejam disparados 16 blocos com 64 *threads* cada. A ordem que os blocos ou os dois *warps* dentro do bloco irão executar não é definida. CUDA provê mecanismos de sincronização a nível do bloco (o método *__syncthreads*) e a nível de todos os blocos (o método *cudaDeviceSynchronize*) para garantir a ordem de execução. Importante salientar, que quando um bloco é escalonado para execução, o bloco será alocado em uma unidade de execução e permanece na unidade até finalizar sua execução. Portanto, a sincronização dentro do bloco é executada pela GPU, já a sincronização de todos os blocos só pode ser realizada com uma chamada externa da CPU. OpenCL também possui funções equivalentes.

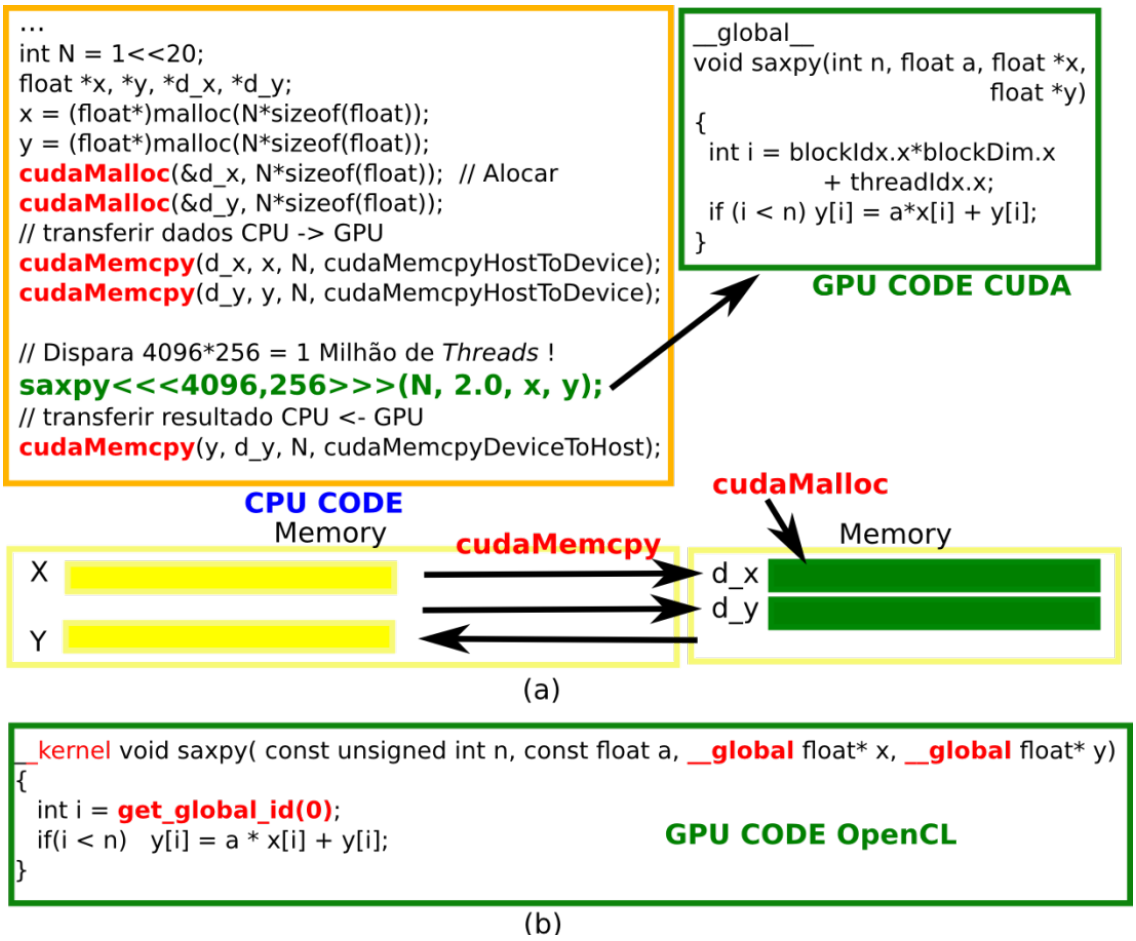


Figura 5.1 - (a) Exemplo de código CUDA para a função SAXPY Y= AX+Y em CUDA para vetores com 1 milhão de elementos disparando 1 milhão de threads. (b) Código do Kernel SAXPY em OpenCL.
Fonte: Elaborado pelos autores

5.2.4. Arquitetura

A arquitetura básica da GPU é composta por centenas de núcleos simples que podem executar uma operação com inteiros, com a memória, com ponto flutuante ou uma função especial. Cada núcleo é referenciado pelo termo *Stream Processor* (SP) ou *Processing Element* (PE) na nomenclatura CUDA e OpenCL, respectivamente. Os núcleos são agrupados em *Stream Multiprocessor* (SM) ou *Compute Unit*, seguindo a nomenclatura CUDA e OpenCL. Nas arquiteturas CUDA, o SM ou SMX, foi evoluindo ao longo das gerações. Na primeira geração eram 8 SPs por SM, depois 32 SPs na geração Fermi em 2010, depois 192 na geração Kepler, posteriormente reduzindo para 128 na geração Maxwell e em 2016 reduzindo para 64 na geração Pascal. Além de agrupar os SPs, cada SM tem recursos que são compartilhados pelos SPs como:

memória cache L1, memória compartilhada, banco de registradores e escalonador de execução. Cada SM pode receber um ou mais blocos para executar. Quando um bloco entra em execução, ele ficará residente no SM até finalizar sua execução. Cada geração tem um número máximo de blocos que podem ser alocados simultaneamente na mesma SM. Cada geração é caracterizada por sua *capability*. Além do número máximo de blocos por SM, a *capability* determina a quantidade máxima de registradores por *thread*, o tamanho das memórias L1 e compartilhada, o número máximo de *warps* que podem estar em execução, dentre outros aspectos.

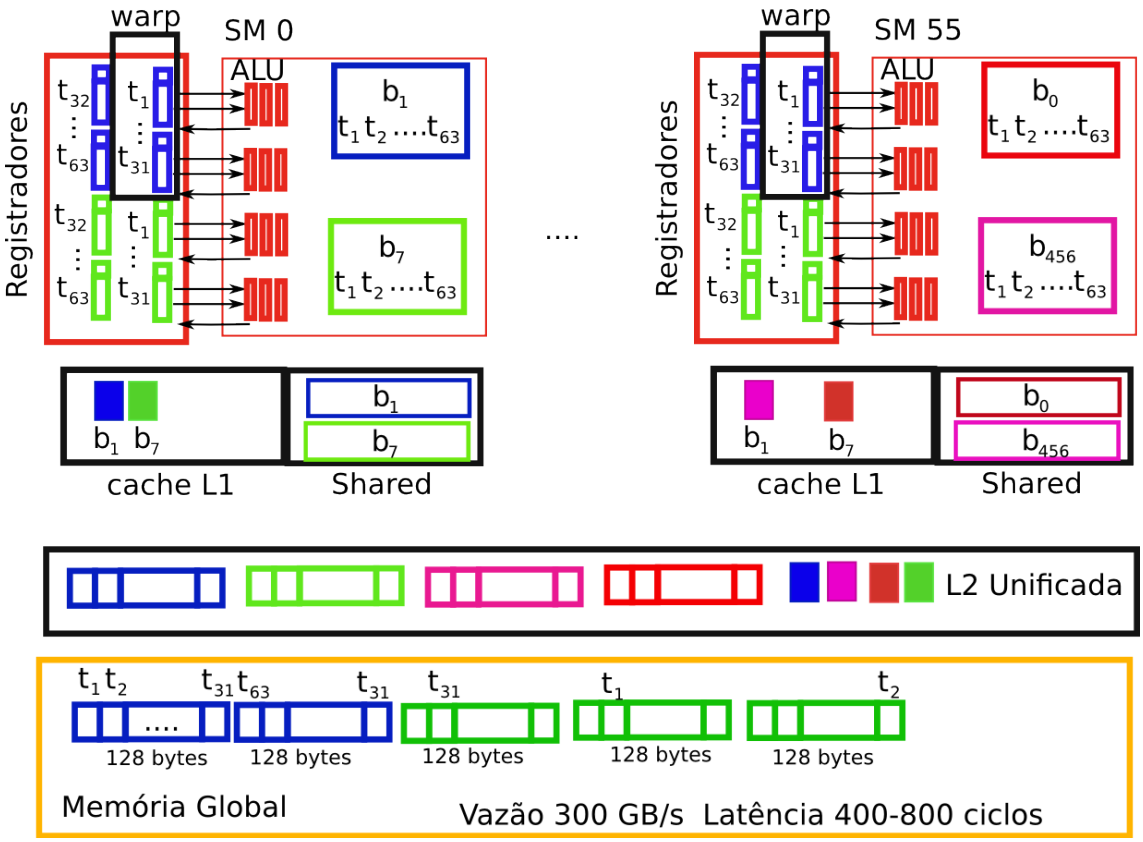


Figura 5.2 - Arquitetura da GPU Nvidia. Exemplo ilustrando uma alocação de blocos na SM com vários Warp sem execução, compartilhamento dos recursos de memória (L1, registradores, compartilhada ou *shared*, L2 e memória global).
Fonte: Elaborado pelos autores

A Figura 5.2 apresenta de forma simplificada a arquitetura de uma GPU. A memória global da GPU tem uma vazão elevada de até 300 GB/s. Para obter o máximo de vazão, os *threads* devem disparar várias requisições, sendo que os *threads* do mesmo *warp* devem requisitar dados consecutivos em pacotes de 128 bytes. Suponha que a memória global tenha uma latência de 1.000 ns e pode chegar a uma vazão de 320 GB/s. Para

obter esta vazão é necessário ter 320 GB/s x 1.000ns=320KB em pedidos em andamento. Suponha uma GTX1080 com 20 SMs. Cada SM deve solicitar 320/20=16KB. Se cada SM estiver executando 2.048 threads, teremos 16k/2K=8 bytes por *thread*. Ou seja, cada *thread* deve requisitar no mínimo dois inteiros. Se o *thread* requisitar mais dados desde que utilize um padrão de acesso agrupado a nível do *warp*, o desempenho máximo pode ser mais facilmente obtido como demonstrado em [Volkov, 2010]. Suponha os *threads* 0 a 31 do bloco B1, ilustrados em azul na figura 5.2. Podemos observar que o bloco B1 está alocado no SM0 e irá permanecer até o fim de sua execução. Cada *thread* tem seu próprio conjunto de registradores. Quando o *warp* entra em execução, os registradores fornecem imediatamente os valores para os cores ou SPs. Os SPs executam em *pipeline*, demorando de 10 a 20 ciclos para gerar o resultado. Se as instruções tem dependência de dados, o *warp* sai da execução e outro *warp* entra. A troca de contexto pode ser realizada em um ciclo de relógio. Um dos diferenciais da GPU é a presença de um grande número de registradores. As variáveis locais dos *threads* permanecem vivas enquanto o bloco está alocado na SM. A capacidade de armazenar dados nos registradores é igual ou maior que a capacidade da cache L1. A arquitetura do banco de registradores têm grande impacto no desempenho das GPUs [Mittal, 2016c].

Suponha que sejam disparados 512 blocos. A Figura 5.2 destaca apenas 4 blocos em execução, os blocos B1 e B7 na SM0 e os blocos B0 e B456 na SM55. Os modelos CUDA e OpenCL não garantem a ordem de execução dos blocos nem a alocação dos mesmos nas SMs ou *Compute Unit*. Ao alocar um bloco, seus *threads* ganham seus registradores exclusivos e sua área na memória compartilhada (*shared*). Somente os *threads* do mesmo bloco podem acessar sua porção da *shared*.

Diferentemente do acesso a memória *shared*, o acesso a *cache* não é exclusivo ao thread ou ao bloco. A *cache* L1 pode conter dados que sirvam aos blocos alocados no SM como mostra a Figura 5.2. Um dado da memória global pode até ser replicado em mais de uma *cache* L1. A *cache* L2 é unificada para todas as SMs, ou seja, um dado requisitado por um bloco pode já estar na L2 devido a um acesso prévio de outro bloco anterior. Outro ponto que é muito destacado são os acessos agrupados (*coalesced*) na memória global. Os dados são agrupados em pacotes de 128 bytes (32 palavras de 32 bits). Qualquer acesso, busca um ou mais pacotes de 128 bytes. Quando um *warp* faz acesso a um pacote, cada *thread* busca um dado dentro do pacote como ilustrado na

figura 5.2 para o *warp* 0 de B1, em azul. Mesmo que os acessos sejam permutados (por exemplo o reverso) como para o *warp* 1 de B1, o acesso é perfeito. Porém, para o *warp* 0 de B7, em verde, onde T31, T1 e T2 acessam pacotes distintos, serão necessários três acessos. No pior caso podem ser até 32 acessos de 128 bytes cada. Algumas GPUs permitem o acesso em pacotes de 32 bytes que reduz o impacto nos acessos irregulares. O melhor é organizar suas estruturas para gerar acesso agrupado dentro do *warp* ou fazer uso da memória compartilhada.

A memória compartilhada ou *shared* têm várias aplicações e propriedades. Primeiro, cada bloco tem seu espaço reservado e privado na *shared*. O espaço é compartilhado pelos threads do bloco e pode ser usado para eles se comunicarem e cooperarem. A *shared* é organizada em 32 bancos. Se cada *thread* do *warp* fizer acesso a um banco distinto, não haverá conflito. Se 2 ou mais *threads* fazem acesso ao mesmo banco mas na mesma posição também não tem conflito. Ou seja, a *shared* possibilita ao programador organizar os dados para garantir um acesso eficiente (separando por bancos). Exemplos clássicos de uso eficiente da *shared* são a transposição e multiplicação de matrizes e histogramas [Cheng, 2014]. Ao usar a *shared*, o programador pode reusar o dado e evitar buscas na memória global. Além disso, a vazão da *shared* é bem superior à vazão da memória global e a latência é menor. Ademais, as memórias da GPU podem ser exploradas como recurso didático [Ferreira, 2013].

São muitos recursos em hardware com finalidades e propriedades específicas. Além da memória global em pacotes, uso massivo de registradores, *shared* em bancos distribuídos e conteúdo gerenciado pelo programador, execução em *warps* e a *cache* L1, temos a disponibilidade instruções cálculo com precisão menor e maior desempenho (compromisso desempenho/precisão), operação *shuffle* para troca de valores entre os *threads* do *warp*, cache de constantes, paralelismo a nível de tarefa com *streams* e multiprocessamento, operações atômicas, dentre outras. Para facilitar a compreensão do impacto de cada um dos recursos no desempenho da sua aplicação, as ferramentas Nvpp e *nvprof* da Nvidia permitem a visualização de métricas (transações com a memória, falhas de *cache*, ocupação da GPU, número de instruções por classe que foram executadas, etc) durante a execução. A métrica mais importante é o desempenho final, mas é preciso entender como foi obtido e se pode ser melhorado. Modelos podem ser utilizados para facilitar a análise [Bombieri, 2016]. O *nvprof* é uma ferramenta de linha de comando, ao executar um aplicativo mostra o número de chamadas por função e o

tempo com precisão de μs. Ao executar o *nvprof* com o parâmetro *--metrics all*, todas as métricas serão exibidas e o programador terá muita informação para fazer sua análise. O *Nvvp* é uma ferramenta gráfica que usa as métricas e já disponibiliza algumas análises dos pontos do código que podem ser melhorados. Outros ambientes estão sendo desenvolvidos para auxiliar o programador nas opções de implementação paralelas, como o Pro++ [Bombieri, 2015], que coleta as informações das ferramentas de *profile* das GPUs para analisar o impacto de uma ou mais primitivas no desempenho final. Estudos recentes propõem também um conjunto de benchmarks para avaliação das arquiteturas de GPU [Bombieri, 2016] em termos de desempenho, potência e energia. Com o auxílio das métricas e destes estudos, o programador pode guiar o desenvolvimento da sua aplicação otimizando o acesso aos vários tipos de memória e as classes de instruções da GPU para maximizar a eficiência energética e o desempenho. Existem também ferramentas automáticas que dinamicamente otimizam a execução do código, como por exemplo a otimização do acesso à memória global proposta em [Fanzia, 2015] que reporta ganhos de 2 a 35 vezes em desempenho. Outro ponto importante é garantir a corretude do código. Recentemente, ferramentas de verificação formal para GPUs foram propostas [Pereira, 2016].

5.3. Bibliotecas e OpenACC

Ajustes finos para otimizar código em CUDA ou OpenCL exigem um conhecimento detalhado do modelo de programação e do hardware da GPU. Para abstrair destes detalhes das arquiteturas e das construções paralelas com as extensões CUDA e OpenCL, o programador pode fazer uso de duas abordagens: bibliotecas ou OpenACC. Para diversas aplicações, existem várias bibliotecas para GPU que oferecem portabilidade e alto desempenho. Por exemplo, a Nvidia oferece a biblioteca *Thrust* [Bell, 2011], [Thrust, 2016] que é baseada na biblioteca padrão STL C++. Com diversos *templates*, o programador pode, por exemplo, criar um vetor, transferir para a GPU e executar uma função de ordenação com poucos comandos, conforme ilustrado na Figura 5.3a. A Figura 5.3b ilustra o código para a função SAXPY, por ser simples, pode ser implementada diretamente na chamada. Contrastando com o código CUDA da Figura 5.1, as bibliotecas oferecem a portabilidade e o alto nível nas chamadas, sem perder eficiência na execução.

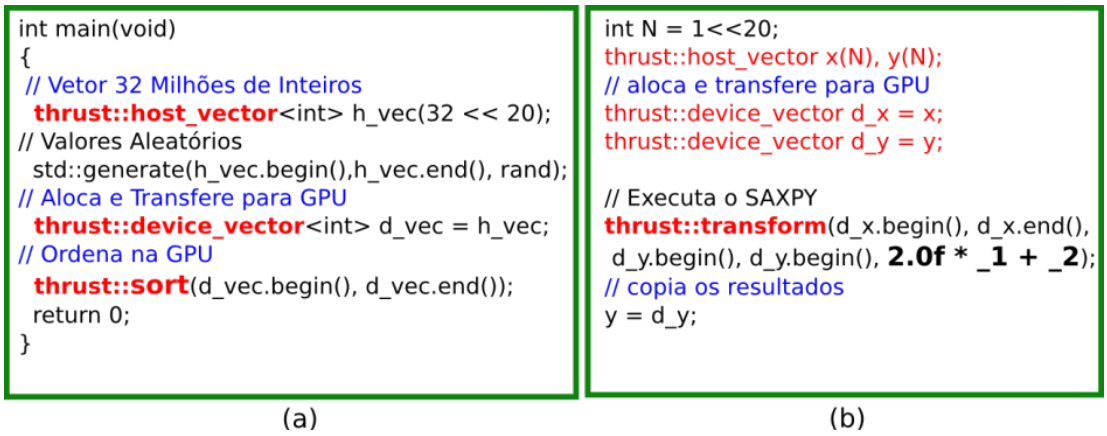


Figura 5.3 - Biblioteca Thrust (a) Ordenação (b) Função SAXPY
Fonte: Elaborado pelos autores

Outro exemplo é a biblioteca CUBLAS para álgebra linear como o desempenho de 2,2 TeraFlops para multiplicação de matrizes em uma GPU TitanX. O leitor pode se referenciar ao livro [Cheng, 2014] ou a página de bibliotecas da Nvidia [Libraries, 2016] para maiores detalhes. A programação com bibliotecas envolve um investimento inicial na documentação. Porém, se suas aplicações alvo podem ser escritas com as primitivas da biblioteca, ou se você já é usuário de bibliotecas semelhantes como a STL ou a BLAS, a transição para *Thrust* e CUBLAS é direta. Podemos destacar ainda outras bibliotecas: NPP para processamento de imagens, FFmpeg para processamento de áudio e vídeo, CURAND para números aleatórios e OpenCV para vídeos e imagens. Na seção 5.4 iremos detalhar mais dois exemplos de bibliotecas: a CuDNN para redes neurais e a NVGraphs para grafos com redes complexas. As bibliotecas podem trabalhar em cooperação com o código CUDA. Se o programador tem uma aplicação que precisa realizar ordenação ou redução, pode fazer uso da biblioteca *Thrust* e depois trabalhar os resultados com as funções específicas de sua aplicação implementadas em CUDA. Outra opção é incluir o código da biblioteca na sua aplicação C ou C++ para fazer uso da GPU sem a necessidade de ter conhecimentos de CUDA ou OpenCL.

Uma segunda opção de abstração para programação alto nível com desempenho é a utilização do OpenACC, que é uma padronização através de diretivas de compilação com *pragmas* para computação paralela desenvolvida pelas empresas Cray, CAPS, Nvidia e PGI. O padrão foi projetado para simplificar a programação paralela em ambientes heterogêneos com múltiplas plataformas como CPUs e GPUs, dentre outras. Diretivas OpenACC e OpenMP podem ser usadas conjuntamente.

```
void saxpy(int n, float a, float * restrict x, float * restrict y) {
#pragma acc kernels
for (int i = 0; i < n; ++i)     // o comando FOR será mapeado na GPU
     y[i] = a*x[i] + y[i];
}
....
saxpy(1<<20, 2.0, x, y); // Vetor 1 milhão para função Y = a*X + Y
```

Figura 5.4 - Código para exemplificar o uso de uma diretiva OpenACC para expressar paralelismo em alto nível.
Fonte: Elaborado pelos autores

A Figura 5.4 ilustra uma trecho de código C com uma diretiva OpenACC. A função *saxpy* é chamada para calcular um vetor do 1 milhão de elementos. No código da função, a diretiva *acc kernels* irá mapear o trecho do comando *for* para a GPU de forma transparente. Adicionando diretivas ao longo do código, compilando e avaliando o desempenho, o desenvolvedor pode, em poucas horas de trabalho, acelerar sua aplicação sem conhecimentos de CUDA ou OpenCL. No exemplo da Figura 5.4, todos os elementos do vetor podem ser calculados em paralelo e o compilador detecta este tipo de padrão com facilidade. Outras ferramentas estão sendo desenvolvidas para auxiliar a programação com OpenACC e OpenMP. Por exemplo, o projeto DawnCC [DawnCC, 2016] detecta automaticamente código paralelizável em programas C/C++, alterando o código fonte para ser compilado com OpenACC ou OpenMP, otimizando de forma transparente a aplicação para executar em GPU ou arquiteturas SIMD.

5.4. Deep Learning e Big Data

Com o fenômeno do *Big Data*, as GPUs se popularizaram ainda mais nos *data centers* devido à sua eficiência energética em comparação com as CPUs. Para analisar o grande volume de dados, várias técnicas de aprendizado de máquina vêm sendo pesquisadas e aplicadas. Uma das abordagens é o *deep learning* baseada em um modelo em alto nível usando um grafo com muitas camadas de processamento, ou seja, uma grafo “profundo”. Recentemente, a NVIDIA vem disponibilizando várias ferramentas e bibliotecas de alto desempenho com *deep learning* para serem usadas nos *data centers*, nos computadores pessoais e nos sistemas embarcados. Uma delas é o DIGITS (anacrônico para *Deep LearnIng GPU Training Systems*) [Digits, 2016], que permite

aos pesquisadores usarem as DNNs (*Deep Neural Networks*) através de uma interface gráfica usando um navegador de Internet. O DIGITS é voltado para classificação de imagens e detecção de objetos. A ideia é oferecer um ambiente interativo para treinar e projetar uma rede para classificação de imagens sem a necessidade de programação e depuração. Outra opção é usar as ferramentas de *Deep Learning* da NVIDIA em conjunto com outros ambientes como Caffe da Universidade de Berkeley [Caffe, 2016], o CNTK, o TensorFlow, o Theano ou o Torch [Torch, 2016]. Por exemplo, o Caffe é um ambiente que suporta as linguagens C,C++, Python, MATLAB ou uma interface simples com comando de linha. Toda a modelagem do *deep learning* é realizada em alto nível. A GPU é usada de forma transparente para acelerar a execução. O modelo é portátil e pode ser executado em uma máquina com ou sem GPU. A GPU é vantajosa em cenário com grandes bases de dados. Para informações adicionais com exemplos e tutoriais, o leitor pode se referir a documentação oficial do Caffe disponível em [Caffe, 2016].

Outro recurso visando a computação com *Big Data* é o suporte a pacotes para manipulação de grafos com grande volume de dados, as redes complexas. Uma rede complexa pode modelar uma base de dados de artigos científicos, sistemas biológicos, redes sociais, redes de roteadores (infraestrutura da internet), redes de interligações entre aplicações WEB (*twitter, youtube, etc.)* dentre outros problemas. A NVIDIA disponibiliza a biblioteca *NVgraphs* [Nvgraphs, 2016] na versão 8 da plataforma CUDA. Oficialmente, a documentação ainda não está disponível, mas uma versão beta da plataforma pode ser avaliada e apresenta alguns exemplos de operações como o caminho mais curto, *pagerank* (3 vezes mais rápido em comparação Xeon E5 com 24 núcleos) e as primitivas *semi-ring* SPMV que podem ser aplicadas em algoritmos para percursos em grafos. A ideia é oferecer suporte para implementação paralela de forma transparente para algoritmos de análise de dados estruturados como redes complexas.

5.5. FPGAs

Os FPGAs (*Field Programmable Gate Array*) são circuitos reconfiguráveis após a fabricação [Hauck, 2010]. Os primeiros FPGAs surgiram na década de 1980, como uma evolução dos PLA (Programmable Logic Array). Um FPGA é um arranjo bi-dimensional de blocos lógicos reprogramáveis interligados por um conjunto de

conexões que também são reprogramáveis. Cada bloco lógico em geral implementa uma função lógica simples com 3 a 6 variáveis. Com a evolução dos FPGA, os blocos lógicos se tornaram mais complexos e usualmente são denominados pelo termo CLB (*Configurable Logic Block*). As interconexões são em geral implementadas por vários barramentos em linhas e colunas como ilustrado na Figura 5.5. Nos cruzamentos são posicionados os roteadores (SW - *Switches*). Os FPGAs têm pinos de entrada e saída reconfiguráveis (IOBs) para se comunicarem externamente. Além disso, os FPGAs atuais tem blocos heterogêneos com granularidade maior como blocos de memórias RAM com capacidade de 2 a 20 Kbits, em geral com largura de 16 a 36 bits que são intercalados com a estrutura regular de CLBs. Os blocos de CLB podem também ser usados como módulos de memória de poucos bits. Existem também módulos de DSP que são ALUs que já tem implementadas funções lógicas e aritmética (soma, subtração e multiplicação) com 18-36 bits. Alguns FPGAs também possuem processadores implementados dentro do chip com módulos de memória *cache*. Por exemplo, o Zynq-7000 da Xilinx possui integrado no chip do FPGA: ARM CortexA9 dual core, CLBs equivalentes a 430K até 1,6M portas lógicas, 240KB a 3MB em módulos BRAM, 80 a 2.020 módulos DSPs.

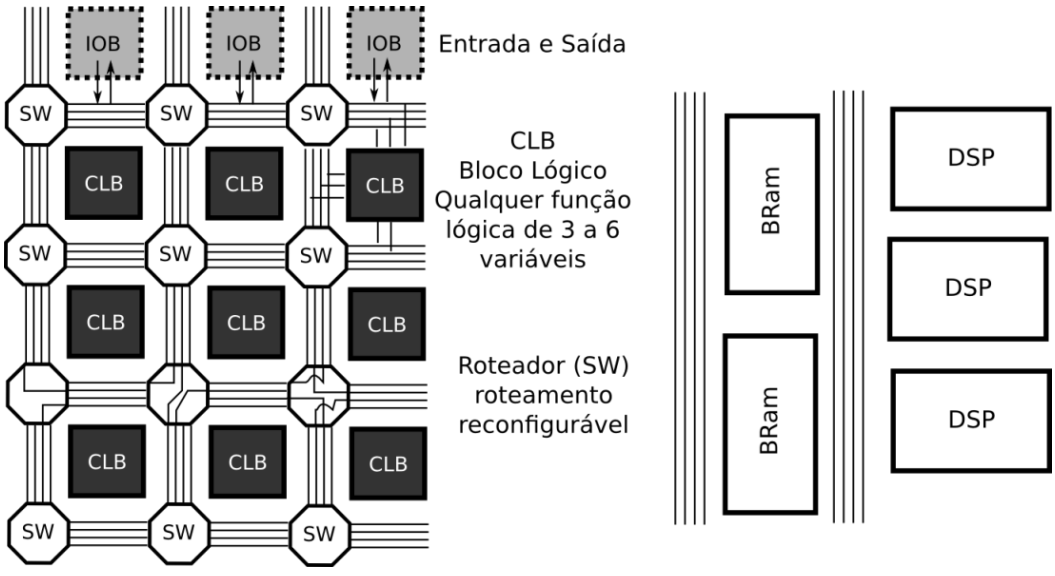


Figura 5.5 - Uma matriz bi-dimensional de um FPGA com blocos lógicos (CLB), roteadores (SW), entradas e saídas (IOB).
Fonte: Elaborado pelos autores

A estrutura regular torna o FPGA escalável. Sua estrutura reconfigurável o torna flexível e versátil para implementar as funções diretamente em hardware.

5.5.1. OpenCL

A Altera e a Xilinx disponibilizam ambientes (Software Development Kits - SDKs) para programação com OpenCL em FPGAs. Existem também outros ambientes para programação em linguagem C com OpenCL que podem ser usados para gerar códigos para FPGAs Altera ou Xilinx, como o [Pico, 2016]. O objetivo é oferecer uma abordagem em alto nível para o mapeamento do código em FPGA sem a necessidade de conhecimentos aprofundados de hardware. Um segundo ponto é popularizar a solução com FPGA na comunidade de programadores de GPU e OpenMP. Por exemplo, a implementação OpenCL da função SAXPY é a mesma em FPGA ou GPU. O ambiente da Altera [OpenCL Altera, 2016] disponibiliza vários exemplos de códigos em OpenCL com tutoriais didáticos. Um exemplo é a implementação do algoritmo GZIP com o desempenho de 3 Gb/s em uma FPGA 28-nm Stratix-V A7, uma aceleração de 10 vezes em relação a versão CPU 32-nm Intel Core i5 650, 3.2 GHz [Abdelfattah, 2014]. Além do desempenho, o FPGA se mostrou 12 vezes mais eficiente em consumo energético, considerando o exemplo de compactação [Abdelfattah, 2014]. A Figura 5.6 ilustra o código C e o hardware para o casamento de padrões paralelo com uma implementação com paralelismo temporal e espacial em hardware. O string alvo é comparado com 4 strings candidatos em paralelo, onde o match é armazenado no vetor length e o melhor candidato é escolhido após a redução. Outro exemplo é o ambiente Malie [Wang, 2016], direcionado para a abordagem com MapReduce com programação em OpenCL para FPGAs, onde os resultados experimentais mostraram em média que uma FPGA é 3,9 vezes mais eficiente energeticamente em comparação com a mesma descrição OpenCL mapeada em CPU e em GPU. No experimento, foram usadas as plataformas: um FPGA Altera Stratix V GX com 622K elementos lógicos, 2.560 blocos de memória de 20K e 256 blocos DSP, uma CPU 2.40GHz Intel Xeon E5645 (12 cores) e uma GPU AMD FirePro V7800 com 18 multiprocessadores com 128 cores cada.

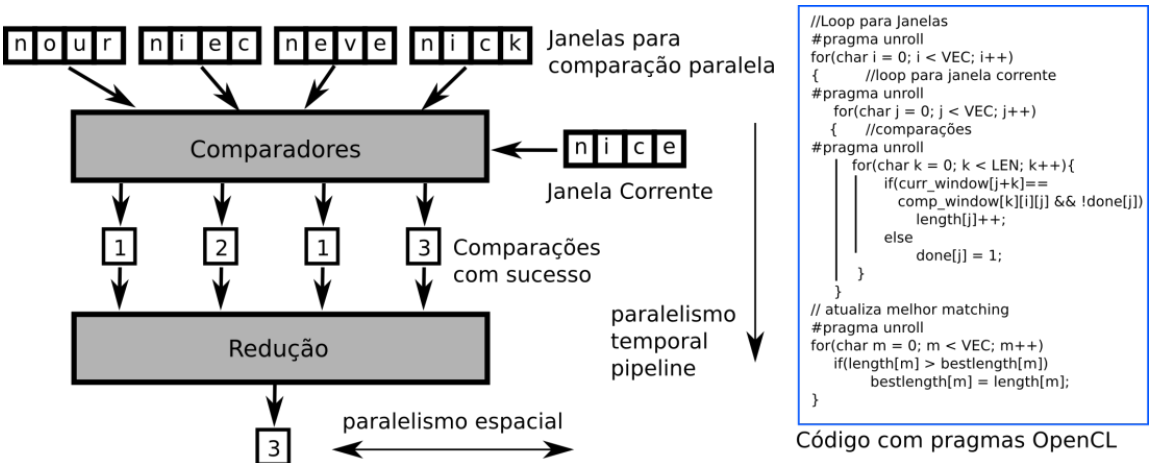


Figura 5.6 - Exemplo de Hardware gerado para Código em OpenCL em FPGA com paralelismo espacial e temporal para casamento de padrões. Figura
Fonte: Adaptada de [Abdelfattah, 2014]

5.5.2. Linguagens para desenvolvimento de aplicações em FPGAs

Aplicações desenvolvidas em FPGAs são normalmente implementadas utilizando linguagens de descrição de hardware como Verilog [Readler, 2011] ou VHDL [Pedroni, 2010]. Estas linguagens estão muito distantes de linguagens de programação e funcionam em um nível muito baixo de abstração utilizando elementos lógicos como, portas digitais, registradores e multiplexadores. Estas descrições podem também ser automaticamente derivadas a partir de linguagens de programação de alto nível similares a C, Java ou OpenCL [Bacon, 2013], [Cardoso, 2009]. Entretanto, para alcançar os melhores níveis de desempenho, os desenvolvedores de aplicações em FPGA devem ter um conhecimento profundo do mapeamento entre um programa de alto nível e sua tradução de baixo nível, incluindo diretivas de otimização no código [Grigoras, 2013]. Dessa forma, a programação eficiente para FPGAs requer treinamento específico e experiência, o que efetivamente cria uma barreira de entrada para novos desenvolvedores. Uma alternativa é a utilização de bibliotecas altamente otimizadas e especializadas desenvolvidas por fabricantes de FPGAs. Apesar de muitas dessas bibliotecas serem específicas de domínio elas também podem fornecer funções genéricas como algoritmos de análise de dados, processamento multimídia e aprendizado de máquina [Iordache, 2016].

5.6. FPGA em Plataformas de Alto Desempenho

Recentemente, várias plataformas de desenvolvimento integrando CPU e FPGA vêm sendo apresentadas como, por exemplo, a plataforma Intel-Altera Harp [Olivier, 2011] e a placa Alpha Data [Alpha, 2016]. As plataformas oferecem uma conexão de alto desempenho entre a CPU e o FPGA via barramento PCI [Alpha, 2016], [Putnam, 2014] ou QPI da Intel [Olivier, 2011]. Um estudo recente apresenta uma avaliação do desempenho destas plataformas [Choi, 2016], avaliando o desempenho das duas arquiteturas e seus mecanismos de comunicação com a CPU. A plataforma Intel-Altera HARP comunica através do barramento QPI da Intel compartilhando a memória cache do processador de forma coerente. A plataforma Alpha comunica através do barramento PCI e transfere os dados para um memória no dispositivo, semelhante ao modelo das GPUs. Além do modelo de programação, a conexão CPU-FPGA envolve novos desafios e questões a serem respondidas e avaliadas nestas plataformas como a frequência de relógio no FPGA, a latência e a vazão na comunicação entre a CPU e FPGA. Estas plataformas são recentes e existe pouca documentação disponível. A avaliação disponibilizada em [Choi, 2016], mostra que a plataforma HARP [Olivier, 2011] possibilita uma comunicação a 7 GB/s entre a CPU e a GPU. Já a plataforma Alpha [Alpha, 2016] que comunica usando o barramento PCI, tem uma taxa de transferência de apenas 1,6 GB/s, entretanto, uma vez copiados os dados para o acelerador FPGA, a leitura na sua memória global pode ser realizada a 9 GB/s. Em [Cong, 2016], os experimentos mostram que um acelerador FPGA acoplado a um processador de alto desempenho como um Xeon é 2 vezes mais rápido que 8 FPGAs acoplados a processadores Low-power (ARM), mesmo que os 8 FPGAs potencialmente tenham 2 vezes mais recursos, ou seja, o caminho é a computação heterogênea com soluções complementares (Superescalares, FPGAs e GPUs). Em [Zhang, 2016], algoritmos de ordenação foram implementados na plataforma Intel/Altera HARP. Os experimentos mostraram um ganho de 2,9 vezes e 1,9 vezes quando comparados a uma implementação em CPU e uma implementação em FPGA. Além disso, os resultados mostraram um aumento em 2,3 vezes na vazão comparado com o estado da arte dos algoritmos de ordenação em FPGA. Em [Passe, 2016], uma abordagem dinâmica para programação do HARP foi proposta. Ou seja, as plataformas heterogêneas com CPU e FPGA trabalhando cooperativamente, são uma alternativa promissora para os data

centers. Em [Putnam, 2014], os experimentos mostram que os FPGAS são robustos para uso em *data centers* e que a maioria dos serviços podem ser mapeados de forma eficiente nos FPGAs usando interconexões com baixa latência entre múltiplas FPGAs. Vale ressaltar que um cuidado especial deve ser tomado ao reiniciar os sistemas, devido à memória de configuração do FPGA. A adição das placas de FPGA só aumentou em 10% o consumo total de energia do sistema e em 30% o custo do hardware, com um grande potencial de aceleração e adaptação [Putnam, 2014].

5.7. FPGA, Deep Learning e Big Data

Como alternativa às GPUs para implementação de *deep learning* tem-se os FPGAs. Ao contrário de GPUs, a principal característica dos FPGAs é a capacidade de reconfiguração de hardware. Além disso, como já mencionado, os FPGAs normalmente apresentam melhor desempenho por Watt para algoritmos importantes em *deep learning* como, por exemplo, processamento por janela deslizante [Fowers, 2012]. Entretanto, para a programação destes dispositivos, é necessário conhecimento específico de hardware, o que muitas vezes dificulta a utilização de FPGAs por cientistas de dados. Recentemente, algumas ferramentas de desenvolvimento de FPGAs têm adotado modelos de programação baseados em software. Conforme descrito na seção 5.5.1, um exemplo é o OpenCL [Czajkowski, 2012] que tem tornado a utilização de FPGAs mais atrativa e acessível para cientistas de dados.

As primeiras propostas de redes neurais em FPGAs surgiram no início dos anos 90 [Cox, 1992]. Entretanto, implementações de redes neurais convolucionais (*Convolutional Neural Networks* – CNNs) em FPGAs apareceram apenas alguns anos depois. Em função das tecnologias utilizadas nas FPGAs da época, estas implementações de CNNs apresentavam limitações de precisão aritmética, velocidade e tamanho [Cloutier, 1996]. Duas décadas depois, as FPGAs atuais apresentam melhorias em densidade de elementos lógicos e unidades computacionais disponíveis em hardware tornando viável, assim a implementação de CNNs mais eficientes em FPGA.

Um estudo de caso interessante de implementação de CNN em FPGA pode ser encontrado em [Ovtcharov, 2015]. Esta aplicação foi desenvolvida no Catapult, que é uma plataforma experimental desenvolvida pela Microsoft para integrar FPGAs em *data centers*. Os autores deste trabalho relatam uma vazão de 134 imagens por segundo

no conjunto de dados ImageNet 1K [Krizhevsky, 2012], o que representa um ganho de aproximadamente 3x em relação à melhor implementação anterior. A FPGA utilizada pelos autores é uma Stratix V D5 e apresenta um consumo de apenas 25 W. No caso de se utilizar um modelo de FPGA mais moderno (Arria 10 GX1150) é esperado que a vazão alcance o valor de 233 imagens por segundo com o mesmo consumo de energia. Como fator de comparação, é importante mencionar que uma implementação do mesmo sistema em GPUs de alto desempenho (Caffe + CuDNN) alcança uma vazão de 500 a 824 imagens por segundo, mas com um consumo de 235 W.

A rede neural recorrente (*Recurrent Neural Network* - RNN) também tem sido implementada com sucesso em FPGAs. As RNNs podem aprender à partir de diferentes sequências de dados no tempo. Esta característica é alcançada à partir da adição de realimentação nas redes neurais tradicionais. Dessa forma, saídas anteriores são levadas em consideração na predição das próximas saídas. RNNs têm se mostrado promissoras em diversas aplicações como reconhecimento de voz [Graves, 2013], tradução automática [Sutskever, 2014] e análise de cenas [Byeon, 2015]. Uma implementação em FPGA de uma RNN do tipo *Long-Short Term Memory* (LSTM) é descrita em [Chang, 2015]. Neste trabalho os autores implementam uma RNN de 2 camadas com 128 unidades em hardware no FPGA Xilinx Zynq 7020. A implementação da RNN foi testada utilizando um modelo de linguagem a nível de caracteres, obtendo resultados 21x mais rápidos que a mesma implementação sendo executada em uma CPU ARM na mesma FPGA.

5.8. GPUs e FPGAs

As GPUs e os FPGAs são arquiteturas complementares que oferecem novas possibilidades para superar os desafios de alto desempenho com eficiência energética. Uma discussão detalhada sobre o estado da arte na avaliação da eficiência energética das GPUs e seu impacto nos supercomputadores foi apresentada em [Mittal, 2015]. Um aspecto importante é não avaliar apenas o consumo. Por exemplo, uma GPU GeForce GTX 590 (40nm) consome 365W em comparação com um processador Xeon E7-8870 (32nm) que consome 150W. Como a GPU executa mais rápido, a energia consumida (produto tempo e potência) da GPU será menor. O estudo apresentado por [Mittal,

2015] mostra que em várias avaliações para aplicações mapeadas em GPUs e FPGAs, as FPGAs são mais eficientes em termos de energia que as GPUs, ambas são melhores que as CPUs, além disso, em termos de desempenho, as GPUs são melhores. Apesar do estudo [Mittal, 2015] ser recente, são avaliados trabalhos até 2014, e as GPU evoluíram significativamente em termos de energia com inovações tecnológicas nas gerações Kepler, Maxwell e Pascal que não foram avaliadas nestes estudos.

As GPUs não são adequadas para códigos com muitos fluxos de controle (sequências com comandos condicionais). Algumas ferramentas foram desenvolvidas para melhorar o desempenho das divergências de controle estaticamente ou dinamicamente [Liang, 2016]. As FPGAs podem implementar fluxo de dados com controle divergente, porém existem ferramentas para mapeamento automático deste perfil de aplicação. Outra limitação das GPUs é a execução de tarefas diferentes de forma concorrente e a baixa vazão na transferência de dados entre a CPU e a GPU limitada pelo barramento PCI. Com a introdução das múltiplas filas de execução na arquitetura Kepler da Nvidia tornou-se possível a execução concorrente de diferentes *kernels*. Alguns cuidados devem ser levados em consideração para evitar serializações das tarefas durante a sua execução. A abordagem apresentada em [Luley, 2016] propõe um ambiente para gerenciar e escalonar de forma eficiente e transparente para o programador diversos *kernels.* Além disso, o ambiente escalona as transferências de dados para maximizar o desempenho e ao mesmo tempo reduzir o consumo de energia. Ademais, o ambiente proposto implementa uma classe para monitorar a potência que é conectada a NVIDIA Management Library (NVML), que pode acessar a GPU e ler o sensor de potência da placa.

Os maiores desafios nos FPGAs ainda são a programação e a pouca disponibilidade de aceleradores com bibliotecas para uso em *Big Data*. A maioria dos aceleradores é projetada no nível RTL (*Register Transfer Level*). Recentemente, novos avanços em síntese de alto nível estão disponibilizando ferramentas, mas ainda se faz necessário conhecimentos de hardware [Cong, 2016b]. Outro aspecto é a falta de gerenciadores *runtime* para suporte aos aceleradores FPGAs nos *datacenters*. Recentemente, um ambiente para programação e gerenciamento em alto nível a partir código C foi proposto em [Cong, 2016b]. Em OpenCL, o compilador gera o código para o acelerador alvo, otimizando memória, reuso de dados, fluxo de dados, etc. A programação é baseada em três *pragmas*: *task*, *parallel* e *pipeline*. O pragma *task* especifica a região

que será mapeada no FPGA. Os pragmas *parallel* and *pipeline* podem ser usados pelo programador para controlar quais laços do código serão mapeados em kernels paralelos (espacial) ou em pipeline (temporal) no FPGA.

5.9. Considerações Finais

Apesar de existirem mais de 60 mil artigos científicos publicados que reportam aceleração com o uso de GPUs, a maioria é para casos bem específicos e existem poucos artigos [hu, 2016], [Mittal, 2015], [Navarro, 2014] com uma descrição mais ampla da arquitetura e suas características que evoluíram ao longo da última década. Para FPGAs também existem milhares de artigos com ganhos em aplicações específicas publicados nos últimos 25 anos, porém poucas ferramentas de alto nível com finalidade geral. Em ambas as plataformas, a abordagem de deep learning vem sendo investigada para mapeamento de problemas mais irregulares com grande volume de dados. Em ambas as soluções, para ajustes finos, o programador deve conhecer bem a arquitetura e entender as métricas de avaliação, seja para uma programação avançada em CUDA/OpenCL para GPU ou em OpenCL ou um plataforma aceleradora como o HARP ou Catapult para os FPGAs. Em termos de energia, os FPGAs são eficientes ao implementar a computação espacial eliminando os gastos necessários nos modelos de CPU e GPU devido à busca de dados e instruções na memória. Como a tecnologia continua evoluindo seguindo a lei de Moore, os FPGAs são uma solução promissora por serem regulares e escaláveis para o desenvolvimento de aceleradores com eficiência energética. Atualmente, em termos de facilidade de programação e configuração, as GPUs ainda são superiores e como são mais eficientes em termos de energia quando comparadas as CPUs, estes aceleradores já fazem parte da arquitetura básica de alto desempenho dos sistemas embarcados, computadores pessoais e supercomputadores.

Referências

Abdelfattah, M. S., Hagiescu, A., & Singh, D. (2014). “Gzip on a chip: High performance lossless data compression on fpgas using opencl”. In Proceedings of the International Workshop on OpenCL 2013 & 2014 (p. 4). ACM.

Alpha (2016) - “Alpha Data - A high performance reconfigurable board based on the Xilinx Virtex-7”. http://www.alpha-data.com/pdfs/adm-pcie-7v3.pdf

Bacon, D.F., Rabbah, R., Shukla, S., “FPGA programming for the masses,” ACM Queue, vol. 11, no. 2, 2013.

Bahrampour, S., Ramakrishnan, N., Schott, L., & Shah, M. (2015). Comparative Study of Caffe, Neon, Theano, and Torch for Deep Learning.

Banger, R., Bhattacharyya, K. (2013), “OpenCL Programming by Example” ISBN 139781849692342, Packt

Bell, N., & Hoberock, J. (2011). Thrust: A productivity-oriented library for CUDA. GPU computing gems Jade edition, 2, 359-371.

Bombieri, N., Busato, F., & Fummi, F. (2015). An enhanced profiling framework for the analysis and development of parallel primitives for gpus. In IEEE Interrnational Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)

Bombieri, N., Busato, F., Fummi, F., & Scala, M. (2016). MIPP: A microbenchmark suite for performance, power, and energy consumption characterization of GPU architectures. In 2016 11th IEEE Symposium on Industrial Embedded Systems

Bombieri, N., Busato, F., & Fummi, F. (2016, March). A fine-grained performance model for GPU architectures. In 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1267-1272)

Byeon, Wonmin, Marcus Liwicki, and Thomas M. Breuel. "Scene analysis by mid-level attribute learning using 2D LSTM networks and an application to web-image tagging." Pattern Recognition Letters 63 (2015): 23-29.

Caffe, (2016), “Caffe: A Deep Learning Framework” - Berkeley Vision and Learning Center - http://caffe.berkeleyvision.org/

Cardoso, J. M., Diniz, P. C., Compilation Techniques for Reconfigurable Architectures. Springer, 2009.

Chang, Andre Xian Ming, Berin Martini, and Eugenio Culurciello. "Recurrent Neural Networks Hardware Implementation on FPGA." arXiv preprint arXiv:1511.05552 (2015).

Cheng, J., Grossman, M., & McKercher, T. (2014). Professional Cuda C Programming. John Wiley & Sons.

Cloutier, Jocelyn, et al. "Vip: An fpga-based processor for image processing and neural networks." Fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems (MicroNeuro'96), Lausanne, Switzerland. 1996.

Cong, J., Huang, M., Wu, D., & Yu, C. H. (2016). Invited Talk-Heterogeneous datacenters: options and opportunities. In Proceedings of the 53rd Annual Design Automation Conference.

Cong, J., Huang, M., Pan, P., Wu, D., & Zhang, P. (2016b). Software Infrastructure for Enabling FPGA-Based Accelerations in Data Centers: Invited Paper. In Proceedings of the 2016 International Symposium on Low Power Electronics and Design.

Cox, Charles E., and W. Ekkehard Blanz. "GANGLION-a fast field-programmable gate array implementation of a connectionist classifier." IEEE Journal of Solid-State Circuits 27.3 (1992): 288-299.

CUDA Fortran, (2016), PGI CUDA Fortran Compiler - http://www.pgroup.com/resources/cudafortran.htm

Czajkowski, Tomasz S., et al. "From OpenCL to high-performance hardware on FPGAs." 22nd International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2012.

DawnCC, (2016) "DawnCC: a Source-to-Source Automatic Parallelizer of C and C++ Programs", DCC, UFMG - http://cuda.dcc.ufmg.br/dawn/

Digits, (2016), "The NVIDIA Deep Learning GPU Training System" https://developer.nvidia.com/digits

Fauzia, N., Pouchet, L. N., & Sadayappan, P. (2015). Characterizing and enhancing global memory data coalescing on GPUs. In Proceedings of the 13th Annual

IEEE/ACM International Symposium on Code Generation and Optimization (pp. 12-22).

Ferreira, R., Fontes, G. (2013) “Ensino de Organizações de Memória em Arquiteturas Paralelas usando Placas Gráficas Aceleradoras”, International Journal of Computer Architecture Education (IJCAE), V2(1).

Fowers J., Brown G., Cooke P., and Sitt G.. “A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications”. In Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pages 47-56. ACM, 2012.

Graves, Alex, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks." 2013 IEEE international conference on acoustics, speech and signal processing. IEEE, 2013.

Grigoras, P, Niu, X, Coutinho, J. G., Luk, W., Bower, J., Pell, O., “Aspect driven compilation for dataflow designs,” in Proc. ASAP, 2013.

Hauck, S., & DeHon, A. (2010). Reconfigurable computing: the theory and practice of FPGA-based computation (Vol. 1). Morgan Kaufmann.

Hu, L., Che, X., & Zheng, S. Q. (2016). A Closer Look at GPGPU. ACM Computing Surveys (CSUR), 48(4), 60.

Iordache, A., Pierre, G., Sanders, P., Coutinho, J. G. D. F., & Stillwell, M. (2016, December). High Performance in the Cloud with FPGA Groups. In 9th IEEE/ACM International Conference on Utility and Cloud Computing

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., & Darrell, T. (2014). “Caffe: Convolutional architecture for fast feature embedding”. In Proceedings of the 22nd ACM international conference on Multimedia

Kim, J., Dao, T. T., Jung, J., Joo, J., & Lee, J. (2015). Bridging OpenCL and CUDA: a comparative analysis and translation. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

Liang, Y., Satria, M. T., Rupnow, K., & Chen, D. (2016). An Accurate GPU Performance Model for Effective Control Flow Divergence Optimization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 35(7), 1165-1178.

Libraries. (2016) “GPU-Accelerated Libraries: highly-optimized algorithms and functions”, http://developer.nvidia.com/gpu-accelerated-libraries.

Luley, R. S., & Qiu, Q. (2016, May). Effective Utilization of CUDA Hyper-Q for Improved Power and Performance Efficiency. In 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 1160-1169).

McMillan, R. (2014) “Microsoft supercharges Bing search with programmablechips,” Wired, http://www.wired.com/2014/06/microsoft-fpga/.

Mittal, S., & Vetter, J. S. (2015). A survey of methods for analyzing and improving GPU energy efficiency. ACM Computing Surveys (CSUR), 47(2), 19.

Mittal, S., & Vetter, J. S. (2015). A survey of CPU-GPU heterogeneous computing techniques. ACM Computing Surveys (CSUR), 47(4), 69.

Mittal, S. (2016) “A Survey of Techniques for Architecting and Managing GPU Register File”. IEEE Transactions on Paralleland Distributed Systems. DOI: 10.1109/TPDS.2016.2546249.

Navarro, C. A., Hitschfeld-Kahler, N., & Mateu, L. (2014). A survey on parallel computing and its applications in data-parallel problems using GPU architectures. Communications in Computational Physics, 15(02), 285-329.

Nvgraphs, (2016), “NVIDIA Graph Analytics library (nvGRAPH)” https://developer.nvidia.com/nvgraph.

Oliver, N., Sharma, R. R., Chang, S., Chitlur, B., Garcia, E., Grecco, J., & Mitchel, H. (2011, November). A Reconfigurable Computing System Based on a Cache-Coherent Fabric. In ReConFig (pp. 80-85).

Ovtcharov, Kalin, et al. "Accelerating deep convolutional neural networks using specialized hardware." Microsoft Research Whitepaper 2 (2015).

Passe, F., Vasconcelos, V.C.R, Silva, L. B., Nacif, J., Ferreira, R. (2016) “Virtual Reconfigurable Functional Units on Shared- Memory Processor -FPGA Systems”, SFORUM - Chip in the Mountains.

Pedroni, Volnei A. Circuit design and simulation with VHDL. MIT Press, 2010.

Pereira, P., Albuquerque, H., Marques, H., Silva, I., Carvalho, C., Cordeiro, L., Santos,V., Ferreira.R. (2016). Verifying CUDA programs using SMT-based context-bounded model checking. In Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16).

Pico, (2016), “PicoComputing OpenCL - Removing the Barriers to HPC Design” - http://picocomputing.com/opencl/https://developer.nvidia.com/gpu-accelerated-libraries.

Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., & Haselman, M. (2014). A reconfigurable fabric for accelerating large-scale datacenter services. In ACM/IEEE International Symposium on Computer Architecture (ISCA).

PyCUDA, (2016), “PyCUDA lets you access Nvidia's CUDA parallel computation API from Python” - https://developer.nvidia.com/pycuda.

Readler, Blaine C. Verilog by Example: A Concise Introduction for FPGA Design. Full Arc Press, 2011.

Sanders, J., & Kandrot, E. (2010). CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional.

Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. "Sequence to sequence learning with neural networks." Advances in neural information processing systems. 2014.

Thrust (2016) “Thrust: a parallel algorithms library” https://thrust.github.io.

Torch (2016) “Torch: scientific computing framework” - http://torch.ch/Z.

Wang; S. Zhang; B. He; W. Zhang, (2016)"Melia: A MapReduce Framework on OpenCL-based FPGAs," in IEEE Transactions on Parallel and Distributed Systems, vol.PP, no. 99.

Zhang, C., Chen, R. and Prasanna, V. (2016) High Throughput Large Scale Sorting on a CPU-FPGA Heterogeneous Platform” Computer Engineering Technical Report Number CENG-2015-10 - University of Southern California.

Capítulo 6

Implantação e Análise de Desempenho de um Cluster com Processadores ARM e Plataforma Raspberry Pi

Felipe dos Anjos Lima[1], Edward David Moreno[1], Wanderson Roger Azevedo Dias[2]

[1]Laboratório de Computação de Alto Desempenho (LCAD)
Programa de Pós-Graduação em Ciência da Computação (PROCC)
Universidade Federal de Sergipe (UFS)
{felipes1474, edwdavid}@gmail.com

[2]Laboratório de Arquiteturas Computacionais e Processamento de Alto Desempenho (LACPAD)
Coordenadoria de Informática (COINF)
Instituto Federal de Sergipe (IFS)
wradias@gmail.com

Resumo

Com o desenvolvimento da Computação de Alto Desempenho (HPC), grandes volumes de dados passaram a ser processados de forma rápida, permitindo assim, que avanços significativos fossem alcançados em varias áreas do conhecimento. Para isso, sempre se observou a área de HPC tendo uma infraestrutura complexa. Por outro lado, nos últimos anos, se observa que a capacidade de processamento dos processadores usados em sistemas embarcados, seguindo arquitetura ARM, vem aumentando de forma significativa. Além disso, os custos de aquisição e o consumo de energia dos processadores ARM são menores, quando comparados aos processadores de outras plataformas. Neste âmbito, cria-se a possibilidade de ter HPC usando plataformas menores e mais econômicas e com um custo de manutenção mais acessível. Assim, este capitulo de livro apresenta a análise de desempenho de um cluster embarcado de baixo custo composto por processadores da arquitetura ARM e plataforma Raspberry Pi.

Além da análise do impacto de usar as bibliotecas MPICH-2 e OpenMPI, executando os programas dos benchmarks HPCC e HPL. Neste capitulo apresentamos resultados de desempenho e consumo de energia do cluster com esses programas, mostrando que é possível usar clusters de plataformas embarcadas de baixo custo e com satisfatórios speedups e consumo de energia.

Abstract

With the recent advancements in High Performance Computing (HPC), it is possible to rapidly process high volumes of data, allowing accomplishments in several areas of knowledge. Although the HPC area has been observed as an area of complex infrastructure, in the last years, its been observed that the processing power of processors used in embedded systems, using the ARM architecture, has been increasing significantly. Furthermore, the acquisition costs and energy consumption are lower, when compared to processors of other platforms, thus allowing for the possibility of having HPC with smaller and more economical platforms, with lower maintenance cost and more accessible. Thus, this chapter presents the performance analysis of a low cost embedded cluster composed of processors using ARM architecture and the Raspberry Pi platform. In addition, we analyze the impact of using MPICH-2 and OpenMPI libraries, running benchmark programs HPCC and HPL. In this chapter, we present results of performance and energy consumption of this cluster with these programs, proving that it is possible to use clusters of low cost embedded platforms with satisfactory speedups and energy consumption.

6.1. Introdução

Com o desenvolvimento tecnológico, o poder de processamento dos computadores aumentou muito nos últimos anos. No entanto, a quantidade de informações processadas também aumentou. Por isso, a busca por processadores e sistemas cada vez mais velozes e que atendam as novas demandas continua sendo prioridade. Segundo Rauber e Rünger (2013), várias áreas das ciências naturais e da engenharia estão cada vez mais necessitando de poder computacional, pois são áreas que necessitam realizar simulações de problemas científicos que manipulam grandes quantidades de dados, demandando

assim grande esforço computacional, visto que um baixo desempenho dos sistemas pode levar a uma restrição das simulações e uma imprecisão nos resultados obtidos.

Atualmente, a tarefa de manter o poder de computação dos processadores aumentando continuamente, seguindo a Lei de Moore (MOORE, 1987), tem sido um grande desafio para os engenheiros e cientistas da computação. Os principais motivos são as dificuldades presentes no desenvolvimento de tecnologias que permitam reduzir o tamanho dos transistores, bem como a quantidade de calor dissipada por esses componentes (BLUME, 2013).

Para resolver esses problemas, os *designers* de processadores passaram a projetar circuitos que pudessem processar dados em paralelo, ou seja, ao invés de tentarem aumentar o poder de processamento de um único núcleo do processador, incluindo mais transistores, os *designers* passaram a considerar a criação de processadores com muitos núcleos em um único circuito integrado. Tais circuitos são conhecidos como processadores *multicore*. Ao tirar proveito do paralelismo com processadores compostos por vários núcleos, foi possível elevar o poder de processamento a taxas mais altas do que as que vinham sendo alcançadas quando apenas um núcleo era utilizado.

No entanto, essa não é a única forma possível de explorar paralelismo, pois é possível construir sistemas que realizam processamento paralelo por meio da associação de dois ou mais computadores, conhecidos como nós, conectados a uma rede local LAN (*Local Area Network*), tais sistemas são conhecidos como *clusters*. Todos os computadores em um *cluster* são vistos como um sistema único e coerente que realiza o gerenciamento de memória necessário para manter a consistência da computação realizada.

Para tirar proveito dos recursos disponibilizados pelos sistemas que suportam o processamento paralelo, existem atualmente várias APIs (*Application Programming Interfaces*) e bibliotecas que auxiliam o desenvolvimento de aplicações paralelas. Como exemplos, podemos citar as bibliotecas MPI (*Message Passing Interface*) e OpenMP (*Open Multi-Processing*). A biblioteca MPI possui um conjunto de funções que permitem que sistemas paralelos com memória distribuída troquem mensagens entre si e assim possam compartilhar dados e instruções de controle. Já a biblioteca OpenMP é utilizada para prover o processamento paralelo entre os núcleos de um processador com memória compartilhada, ou seja, várias *threads* são alocadas e executadas

simultaneamente pelos núcleos do processador. Neste capitulo, utilizamos diferentes implementações da biblioteca MPI, assim foi possível medir e analisar o desempenho do *cluster* composto por processadores *quadcore* da plataforma ARM (*Advanced RISC Machine*).

Nos últimos anos, os processadores da plataforma ARM começaram a dominar o mercado dos sistemas embarcados. Apesar de termos a arquitetura x86 presente em boa parte dos computadores pessoais e dos *notebooks*, por serem mais complexos e possuírem estágios de processamento mais longos, os processadores x86 não são uma solução viável para sistemas embarcados. Já os processadores ARM, por serem baseados na arquitetura RISC (*Reduced Instruction Set Computing*), possuem um conjunto de instruções mais simplificado, o que torna o processador menos complexo.

O desenvolvimento das micro-arquiteturas de hardware também permitiu que sistemas computacionais embarcados fossem criados. No entanto, nem todos os tipos de processadores possuem um bom desempenho em tais sistemas. E isso se deve entre outros fatores ao alto consumo de energia, impedindo que o sistema embarcado possa operar de forma independente por mais tempo. Por isso, criar sistemas que operam com pouca quantidade de energia é fundamental para a viabilização de soluções computacionais embarcadas. Além disso, sistemas que consomem muita energia podem não ser implantados, pois os altos custos envolvidos na manutenção do sistema podem inviabilizar a sua utilização (ZOMAYA, 2012).

O objetivo deste capitulo de livro é apresentar a implantação e análise de desempenho de um *cluster* embarcado com processadores ARM e plataforma Raspberry Pi, a fim de verificar o comportamento do mesmo durante a execução de *benchmarks* paralelos em diferentes ambientes de programação. Assim, o capitulo está dividido da seguinte maneira:

- **Seção 6.2** – Conceitos Básicos, tais como: Computação de Alto Desempenho; Tipos de Sistemas Paralelos, Sistemas Multicores, Clusters, Algoritmos Paralelos, Níveis de Paralelismo, Benchmarks Paralelos, Biblioteca MPI e Análise Estatística dos Dados.;
- **Seção 6.3** – Implementação do *Cluster* Embarcado com Processadores ARM, bem como os testes e análises de desempenho levando em consideração os *benchmarks* HPL e HPCC as bibliotecas OpenMPI e MPICH-2;
- **Seção 6.4** – Considerações Finais.

6.2. Conceitos Básicos

Nesta seção apresentamos uma descrição dos conceitos básicos para este capitulo, bem como os materiais e métodos que foram necessários para a extração e análise estatística dos dados. Os experimentos levaram em consideração o desempenho do *cluster* embarcado em diferentes ambientes de programação. Para isso, foram utilizadas as implementações OpenMPI e MPICH-2 da biblioteca MPI. Com o auxílio dos *benchmarks* HPCC e HPL, foi possível analisar a capacidade de processamento do *cluster*, bem como aspectos referentes à comunicação entre os processadores pela rede. Além disso, também foram realizados testes que mediram o consumo de energia do *cluster* durante a execução dos *benchmarks* em diferentes ambientes de programação.

6.2.1. Computação de Alto Desempenho

Com o desenvolvimento científico e tecnológico, foi possível aumentar a capacidade de processamento dos computadores. No entanto, a utilização de sistemas computacionais com apenas um processador passou a não suprir a demanda por processamento necessária em diversas áreas do conhecimento. É nesse contexto que surge a Computação de Alto Desempenho ou HPC (*High-performance computing*). Assim, foi possível criar supercomputadores e *clusters* capazes de processar grandes volumes de dados em um curto período de tempo. Como exemplos de aplicações que demandam grande poder de processamento, podemos citar sistemas que realizam análises meteorológicas, sistemas que realizam análises de grandes cadeias de nucleotídeos presentes no DNA humano, e sistemas que buscam dados na internet (PACHECO, 2012). Atualmente, a lista TOP500 apresenta os 500 supercomputadores mais poderosos do mundo. O critério de avaliação utilizado para a inclusão de um supercomputador na lista é o desempenho obtido durante a execução do *benchmark* HPL.

6.2.2. Tipos de Sistemas Paralelos

Sistemas computacionais podem ser caracterizados de acordo com vários critérios. Uma das classificações mais conhecidas é a proposta por Flynn em 1972. Segundo Flynn, levando-se em consideração o número de fluxos de dados e de fluxos de controle que

um sistema pode gerenciar simultaneamente, um sistema computacional pode ser classificado segundo uma das categorias a seguir:

- **Sistemas SISD (*Single Instruction, Single Data*)**: são sistemas que possuem apenas um processador, e por isso, possuem apenas um fluxo de dados e um fluxo de execução de instruções;
- **Sistemas SIMD (*Single Instruction, Multiple Data*)**: são sistemas que possuem um conjunto de unidades funcionais que operam simultaneamente e são gerenciadas por um único controlador. Assim, uma mesma instrução pode ser executada sobre dados diferentes;
- **Sistemas MIMD (*Multiple Instruction, Multiple Data*)**: são sistemas que possuem múltiplos fluxos de instruções e múltiplos fluxos de controle. Fazem parte dessa categoria os processadores *multicore*, os computadores com múltiplos processadores e os *clusters*.

6.2.3. Sistemas Multicores

Segundo Pacheco (2012), à medida que o tamanho dos transistores diminui, o poder de processamento do circuito integrado aumenta. No entanto, o calor dissipado por esses circuitos também aumenta, sendo um problema para o desempenho do sistema. A solução encontrada para contornar esse problema foi colocar em um único circuito integrado várias unidades de processamento. Tais circuitos ficaram conhecidos como circuitos *multicore* (GEBALI, 2011).

Apesar de o termo *multicore* ser comumente usado para designar sistemas que possuem vários núcleos de processamento em um único chip, ele também é usado para designar sistemas onde os processadores estão em chips diferentes (HAOQIANG, 2011). A Figura 6.1 apresenta o esquema de um processador com duas unidades de processamento.

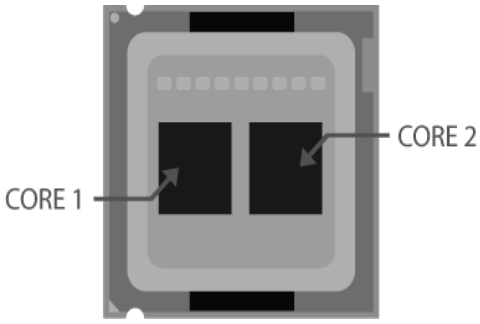


Figura 6.1 - Processador Multicore
Fonte: GEBALI, 2011

Para que possam operar corretamente, sistemas *multicore* necessitam de um gerenciamento de memória eficiente, para que seja mantida a consistência dos dados processados pelos vários núcleos.

Em sistemas de memória compartilhada, com múltiplos processadores, a conexão dos processadores com a memória pode ser feita de duas formas (PACHECO, 2012): UMA (*Uniform Memory Access*) e NUMA (*Nouniform Memory Access*).

No modelo de conexão UMA, os processadores são ligados diretamente à memória principal. Dessa forma, o tempo de acesso a todos os endereços da memória é o mesmo para todos os processadores (PACHECO, 2012). A Figura 6.2 apresenta o esquema de funcionamento do modelo UMA.

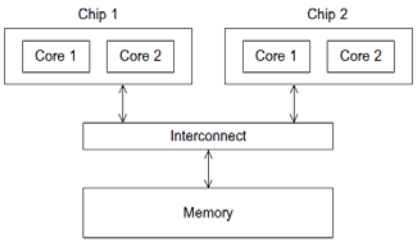


Figura 6.2 - Modelo de conexão UMA
Fonte: PACHECO, 2012

Já no modelo NUMA, cada processador tem acesso a um bloco de memória principal própria (GEBALI, 2011). Caso um processador deseje acessar um bloco de memória de outro processador, será necessário utilizar um hardware adicional, pois o acesso não poderá ser feito diretamente. A Figura 6.3 apresenta o esquema de funcionamento do modelo NUMA.

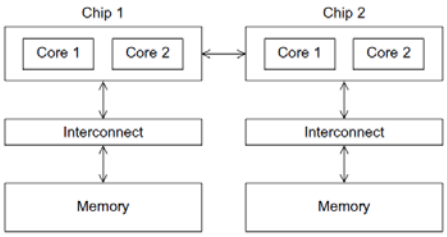


Figura 6.3 - Modelo de conexão NUMA
Fonte: PACHECO, 2012

6.2.4. Clusters

Um *cluster* é uma coleção de dois ou mais computadores usados para executar uma dada tarefa (HAOQIANG, 2011). Normalmente, os nós do *cluster* são conectados por meio de uma rede local que deve permitir a comunicação eficiente entre os mesmos. A Figura 6.4 apresenta o esquema de funcionamento de um *cluster*.

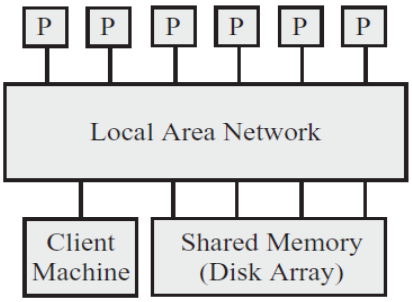


Figura 6.4 - *Cluster* com múltiplos nós
Fonte: PACHECO, 2012

Para manter a consistência durante a execução das tarefas, o *cluster* utiliza o modelo de memória compartilhada NUMA, pois cada processador possui acesso a uma memória própria (PACHECO, 2012). Alguns autores chamam de sistemas de memória distribuída, os sistemas que usam o modelo de compartilhamento de memória NUMA.

6.2.5. Algoritmos Paralelos

Para que seja possível tirar proveito dos recursos disponibilizados pelas arquiteturas paralelas, é necessário que os algoritmos estejam preparados para operar neste tipo de arquitetura, a fim de melhorar a sua velocidade de processamento.

Atualmente, existem muitos algoritmos que não reconhecem os recursos disponibilizados por arquiteturas de hardware paralelas. Sistemas que realizam processamento de imagens, análises de dados sobre o clima, computação científica, pesquisas energéticas e que manipulam grandes quantidades de dados, podem ter um aumento considerável no desempenho caso tirem proveito dos recursos providos por arquiteturas de hardware paralelas.

Para que um algoritmo paralelo possa ser executado corretamente pelo sistema paralelo, ele deve realizar três tarefas básicas:

- **Mapeamento**: é a distribuição das tarefas para os diferentes processadores;
- **Compartilhamento**: compartilha a execução das tarefas conforme a dependência de dados;
- **Identificação**: identifica os dados que trafegam entre os processadores.

6.2.6. Níveis de Paralelismo

Existem vários tipos de paralelismo envolvido em um sistema computacional. Para que o algoritmo possa executar de forma paralela, é necessário que o hardware dê o suporte necessário. A seguir apresentamos os principais tipos de paralelismo (PACHECO, 2012).

- **Paralelismo em nível de *bit***: os *bits* de uma instrução são processados em paralelo;
- **Paralelismo em nível de instrução**: através de um recurso conhecido como *pipeline*, os processadores podem processar instruções em paralelo, pois possuem múltiplas unidades funcionais;
- **Paralelismo em nível de dados**: os dados armazenados podem ser acessados em paralelo e em seguida terem seus resultados combinados;
- **Paralelismo em nível de tarefas**: usa um fluxo separado de controle para cada tarefa que será executada de forma independente.

6.2.7. Benchmarks Paralelos

Para medir o desempenho do *cluster*, foram utilizados *benchmarks* que permitiram analisar vários aspectos do *cluster* proposto. Em Ciência da Computação, um *benchmark* é um conjunto de programas que realizam grande quantidade de operações matemáticas, a fim de extrair métricas de desempenho do sistema. Todos os testes podem ser customizados, por meio da passagem de parâmetros, no momento da execução do *benchmark*.

6.2.7.1. Benchmark HPL - O *benchmark* HPL (*High-Performance Linpack*) é uma versão aprimorada do *benchmark* Linpack, que permite medir o desempenho de sistemas paralelos e *clusters*. Para isso, o *benchmark* realiza operações que demandam grande esforço computacional. Durante sua execução, o HPL resolve um sistema de equações lineares denso do tipo A.x = b, onde A é uma matriz densa gerada aleatoriamente de dimensão N x N, e x e b são vetores de tamanho N. O valor da variável N pode ser alterado quando necessário, a fim de permitir uma maior precisão dos resultados.

O HPL também permite a definição de vários cenários diferentes. Por meio de um arquivo de configuração, é possível modificar os parâmetros que serão levados em consideração pelo *benchmark* durante a sua execução. A Tabela 6.1 apresenta os principais parâmetros do arquivo de configuração do HPL.

Tabela 6.1 - Parâmetros do *benchmark* HPL

Parâmetros	Significado
N	Tamanho do problema
NB	Tamanho do bloco
P	Número de linhas de processos
Q	Número de colunas de processos

Fonte: Elaborada pelos autores

6.2.7.2. Benchmark HPCC - O *benchmark* HPCC é um conjunto formado por outros sete *benchmarks*: HPL, *STREAM*, *RandomAccess*, PTRANS, FFTE, DGEMM e *b_eff Latency*/*Bandwidth*. A descrição de cada *benchmark* é apresentada a seguir:

- **HPL**: realiza o cálculo de um sistema linear denso e como saída apresenta a capacidade de processamento em Gflops/s e o tempo gasto para resolver o problema. A seção anterior apresentou os detalhes do HPL;
- **STREAM**: mede a largura de banda da memória disponível;
- **RandomAccess**: mede a taxa de atualizações aleatórias da memória;
- **PTRANS**: mede a taxa de transferência de grandes *arrays* de dados em sistemas de memória multiprocessados;
- **FFTE**: mede a taxa de execução de pontos flutuantes durante a realização da Transformada de *Fourier*;
- **DGEMM**: mede a taxa de execução de pontos flutuantes durante a multiplicação de matrizes de números reais;
- **b_eff Bandwidth**: mede a largura de banda envolvida na comunicação entre os nós do sistema paralelo.

6.2.8. Biblioteca MPI

MPI (*Message Passing Interface*) é uma biblioteca de "*Message-Passing*", desenvolvida para ser padrão, inicialmente, em ambientes de memória distribuída, em "*Message-Passing*" e em computação paralela (PACHECO, 2012). Todo paralelismo é explícito, ou seja, o programador é responsável por identificar o paralelismo e implementar um algoritmo utilizando construções com o MPI (ALEXANDER, 2010).

Neste modelo de programação paralela, um mesmo código fonte é executado em cada processador. Assim, as variáveis declaradas são relativas a cada processador. O MPI é o responsável por realizar o gerenciamento de memória entre os núcleos do sistema. A Figura 6.5 apresenta o modelo de memória usado em *clusters*.

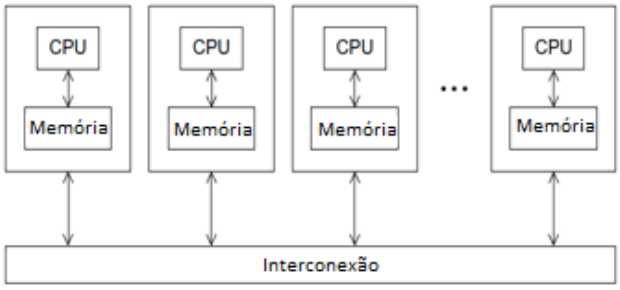


Figura 6.5 - Sistema de Memória Distribuída
Fonte: PACHECO, 2012

Os processos que estão sendo executados pelos nós do *cluster* se comunicam através do envio de mensagens. Para isso, o MPI disponibiliza um conjunto de funções que estão descritas a seguir:

- **MPI_Init**: responsável pelas configurações iniciais do sistema. É a primeira função a ser executada pelo sistema;
- **MPI_Finalize**: libera todos os recursos utilizados pelo MPI. É a última função a ser executada pelo sistema;
- **MPI_Send**: carrega as informações que serão trocadas entre os processos;
- **MPI_Recv**: é usado para receber as mensagens que foram enviadas por outros processos.

Todas as funções apresentadas acima podem ser chamadas a partir de programas escritos nas linguagens C, C++, Python e Fortran.

6.2.9. Análise Estatística dos Dados

Após a realização dos testes com os *benckmarks* propostos, foi feita a análise estatística dos dados obtidos. Para isso, utilizamos a distribuição *t* de *Student*, que pode ser aplicada quando o número de elementos em uma amostra é menor ou igual a 30 (FREUND, 2206). Assim, com a distribuição *t* de *Student* foi possível realizar a análise probabilística que permitiu calcular o intervalo de confiança para a média dos valores presentes nas amostras coletadas durante os testes. Devido ao tempo gasto durante a realização dos testes, optamos por realizar trinta execuções por teste, alterando os *benchmarks* e as implementações da biblioteca MPI.

O cálculo do intervalo de confiança para a média de amostras com N elementos, onde N é menor ou igual a trinta, compreende os seguintes passos (LARSON, 2010):

- Cálculo da média da amostra;
- Cálculo do desvio padrão da amostra;
- Definição do intervalo de confiança desejado;
- Definição do grau de liberdade. Geralmente, é igual a N-1;
- Cálculo da margem de erro no intervalo de confiança;
- Cálculo dos extremos esquerdo e direito do intervalo de confiança.

6.3. Cluster Embarcado com Processadores ARM

Nesta seção, apresentamos o *cluster* embarcado proposto neste trabalho, enfatizando as características da plataforma Raspberry Pi, do sistema operacional Raspbian e do processador ARM.

O *cluster* montado foi composto por quatro nós (plataformas Raspberry Pi do modelo B+), sendo que as comunicações realizadas entre eles são através dos módulos de rede que cada Raspberry Pi possui. O processador contido na plataforma é da família ARM. A Figura 6.6 apresenta uma Raspberry Pi do modelo B+.

Figura 6.6 - Raspberry Pi, modelo B+
Fonte: Fundação Raspberry Pi

Uma das vantagens da utilização da plataforma Raspberry Pi é o baixo custo, pois ela possui todos os componentes que um computador de custo mais elevado possui. Dessa forma, é possível instalar um sistema operacional que irá gerenciar os componentes da placa.

Após a montagem do *cluster*, as tarefas enviadas pela rede são paralelizadas para que sejam executadas por todos os nós do *cluster*. A Figura 6.7 apresenta o *cluster* após a montagem.

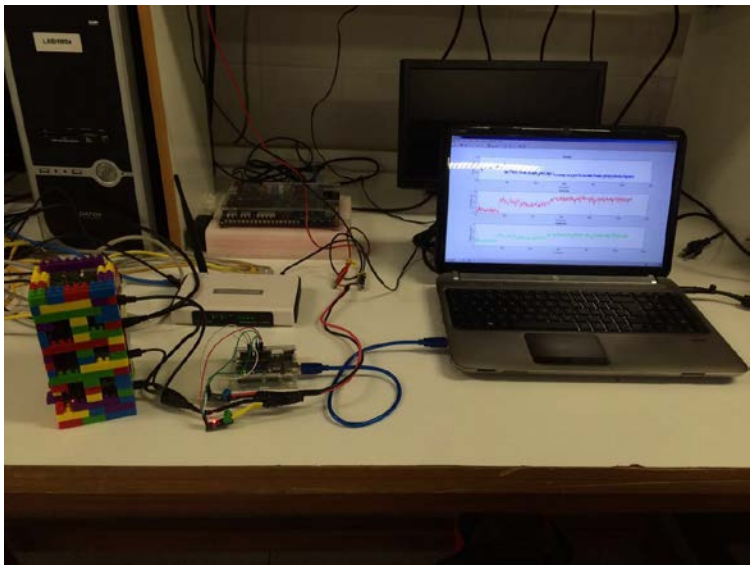


Figura 6.6 - *Cluster* Embarcado
Fonte: Elaborada pelos autores

Para análise de desempenho do *cluster* desenvolvido, foram criados alguns cenários de testes, conforme apresentados na Tabela 6.2.

Tabela 6.2 - Cenários de testes do *cluster* embarcado

Cenário 1	*Cluster* com o *benchmark* HPL e OpenMPI
Cenário 2	*Cluster* com o *benchmark* HPL e MPICH-2
Cenário 3	*Cluster* com o *benchmark* HPCC e OpenMPI e MPICH-2

Fonte: Elaborada pelos autores

6.3.1. Cenário 1 - Desempenho do *Cluster* com HPL e OpenMPI

O *benchmark* HPL resolve um sistema linear com N variáveis e ao final exibe a capacidade de processamento em Gflops/s e o tempo gasto para resolver o sistema linear. Para uma melhor análise do desempenho do *cluster* embarcado, os experimentos levaram em consideração a resolução de sistemas lineares com diferentes valores de N, onde N é o número de variáveis do sistema linear.

Além disso, para verificar a eficiência energética do *cluster* embarcado em diferentes ambientes de programação, neste cenário, também foram realizados testes que mediram o consumo de energia durante a execução do *benchmark* HPL.

Inicialmente, foram realizados testes que mediram a capacidade de processamento do *cluster* à medida que os valores de N e o número de nós aumentavam.

Assim, os testes levaram em consideração a resolução de sistemas lineares com 5000 e 10000 variáveis.

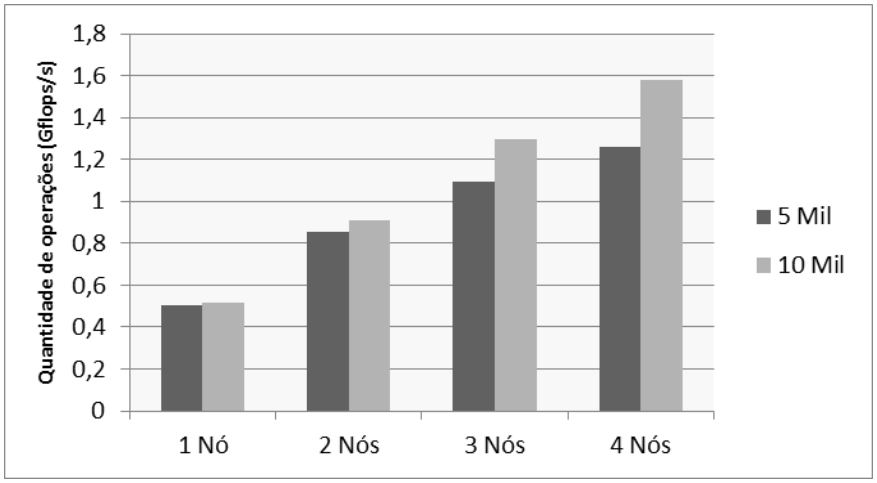


Figura 6.7 - Desempenho do *cluster* com HPL e biblioteca OpenMPI

Fonte: Elaborada pelos autores

Tabela 6.3 - Intervalos de confiança de 95% para N = 5000

Número de Nós	Média	Intervalo de Confiança
1	0,5062	(0,5060 – 0,5064)
2	0,8541	(0,8528 – 0,8554)
3	1,0916	(1,0895 – 1,0937)
4	1,2583	(1,2572 – 1,2594)

Fonte: Elaborada pelos autores

Tabela 6.4 - Intervalos de confiança de 95% para N = 10000

Número de Nós	Média	Intervalo de Confiança
1	0,5185	(0,5183 – 0,5187)
2	0,9093	(0,9072 – 0,9114)
3	1,2991	(1,2976 – 1,3006)
4	1,5767	(1,5750 – 1,5784)

Fonte: Elaborada pelos autores

A análise do gráfico apresentado na Figura 6.7 mostra que um aumento no número de nós do *cluster* embarcado provocou um aumento no desempenho do mesmo.

Para N = 5000, podemos perceber que o *cluster* composto por 4 processadores obteve um desempenho em Gflops/s 59,78% melhor, quando comparado ao *cluster* com apenas um processador. Já para N = 10000, o *cluster* composto por 4 processadores obteve um desempenho em Gflops/s 67% melhor, quando comparado ao *cluster* com apenas 1 nó. As Tabelas 6.3 e 6.4 apresentam os valores médios sob um intervalo de confiança de 95%.

6.3.2. Cenário 1 - Tempo de Execução do *Cluster* com HPL e OpenMPI

Além de medir a capacidade de processamento do *cluster* em Gflops/s, o *benchmark* HPL também mediu o tempo gasto para realizar o cálculo do sistema linear. A Figura 6.8 apresenta os tempos de execução do HPL para a solução de um sistema linear com N = 5000 variáveis.

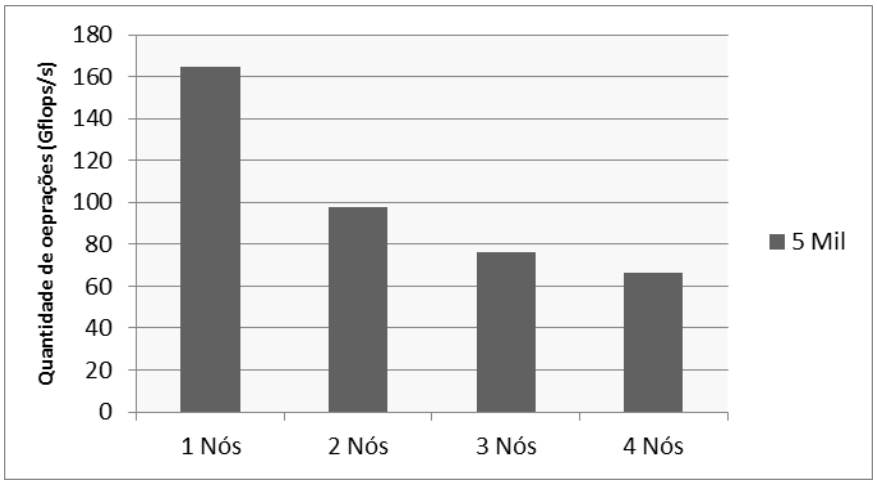


Figura 6.8 - Tempos de Execução do HPL para N = 5000
Fonte: Elaborada pelos autores

Tabela 6.5 - Intervalos de confiança de 95% para N = 5000

Número de Nós	Média	Intervalo de Confiança
1	164,66	(164,64 – 164,68)
2	97,61	(97,58 – 97,64)
3	76,37	(76,36 – 76,38)
4	66,25	(66,23 – 66,27)

Fonte: Elaborada pelos autores

A análise dos dados apresentados na Figura 6.8 mostra que a adição de novos processadores ao *cluster* provocou uma redução no tempo de execução do *benchmark* HPL. Além disso, podemos perceber que o *cluster* composto por quatro processadores obteve um desempenho 60% melhor, em relação ao *cluster* composto por um único processador. Ao analisar os *speedups* apresentados no gráfico da Figura 6.9, podemos perceber que o *cluster* com quatro processadores obteve um desempenho 2.5 vezes melhor quando comparado ao *cluster* com apenas um processador. Apesar de a adição de novos nós ter provocado um aumento no desempenho do *cluster*, pode-se perceber que para a resolução de um sistema linear com 5000 variáveis não houve um aumento considerável no desempenho do *cluster*, pois em um sistema paralelo, além do tempo gasto para executar instruções, existe também o tempo gasto para realizar a comunicação entre os nós.

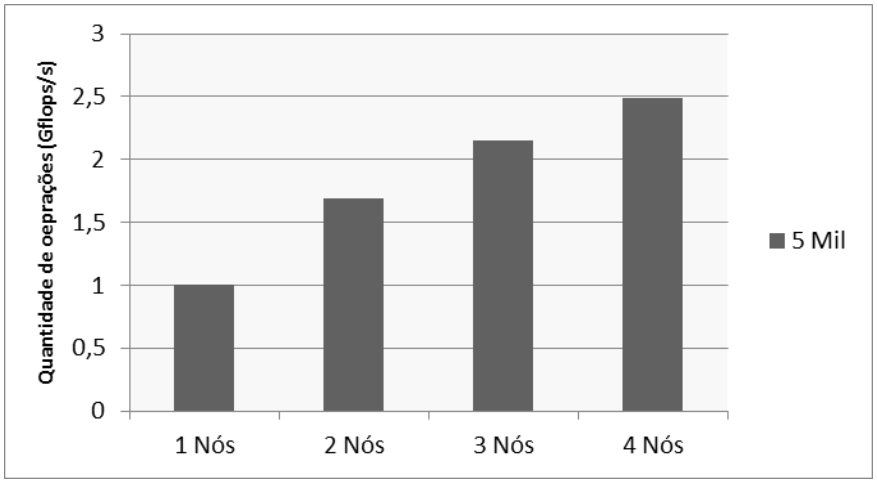


Figura 6.9 - *Speedup* do HPL para N = 5000
Fonte: Elaborada pelos autores

Para verificar o comportamento do *cluster* após o aumento da carga de trabalho, também foram realizados testes que levaram em consideração a resolução de sistemas lineares com dez mil variáveis. O gráfico da Figura 6.10 apresenta os tempos de execução do *cluster* embarcado.

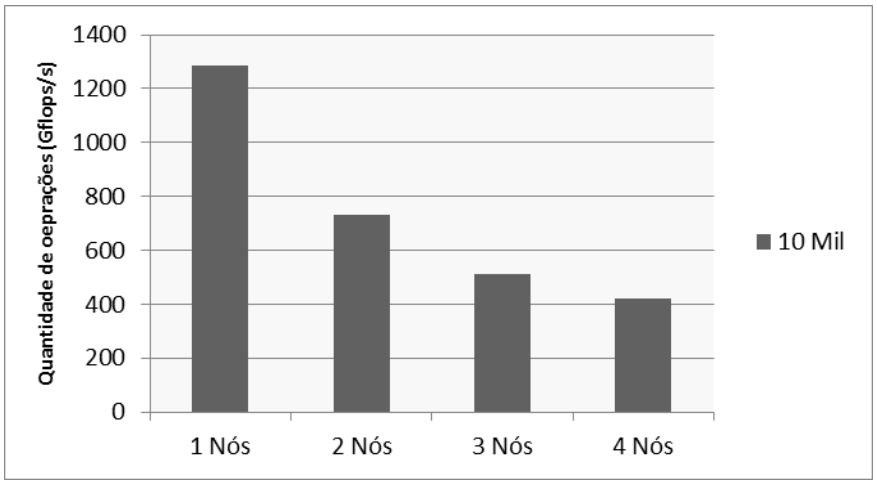


Figura 6.10 - Tempos de Execução do HPL para N = 10000
Fonte: Elaborada pelos autores

Tabela 6.6 - Intervalos de confiança de 95% para N = 10000

Número de Nós	Média	Intervalo de Confiança
1	1285,92	(1285,90 – 1285,94)
2	733,24	(733,20 – 733,28)
3	513,27	(513,25 – 513,29)
4	422,91	(422,89 – 422,93)

Fonte: Elaborada pelos autores

A análise do gráfico apresentado na Figura 6.10 mostra que o *cluster* composto por quatro processadores obteve um desempenho 67% melhor, quando comparado à solução sequencial com apenas um processador. Em relação ao *speedup* alcançado, pode-se perceber que o aumento do tamanho do problema, ou seja, o aumento do número de variáveis do sistema linear provocou um aumento no desempenho do *cluster*. Isso se deve ao fato de os processadores passarem a realizar mais operações sobre os dados, fazendo com que o tempo gasto durante o processamento superasse o tempo gasto durante a comunicação entre os nós. O gráfico da Figura 6.11 mostra que o *cluster* composto por quatro processadores obteve um desempenho três vezes melhor, em relação à solução sequencial com apenas um processador.

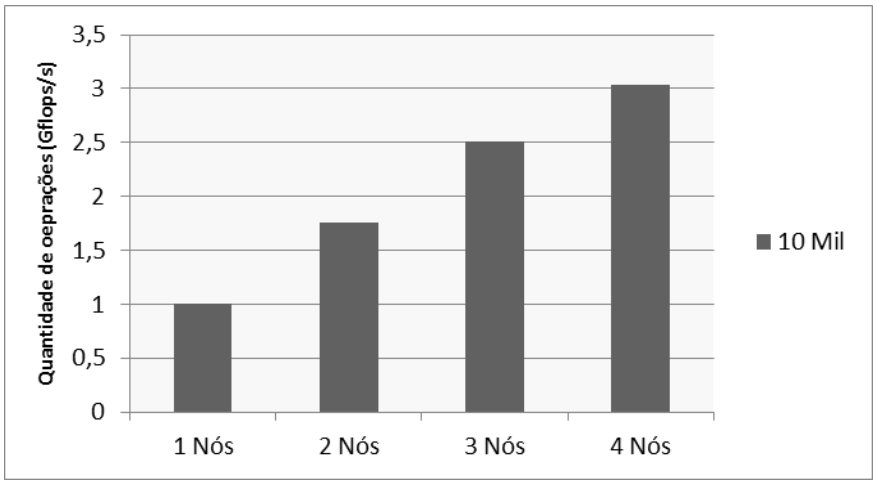


Figura 6.11 - *Speedup* do HPL para N = 10000
Fonte: Elaborada pelos autores

6.3.3. Cenário 1 - Consumo de Energia do *Cluster* com HPL e OpenMPI

Com o objetivo de verificar a eficiência energética do *cluster* embarcado, foi utilizado um circuito real que mediu a corrente, a potência e a tensão durante a execução do *benchmark* HPL. As Figuras 6.12, 6.13, 6.14 e 6.15 apresentam o consumo energético do *cluster* embarcado.

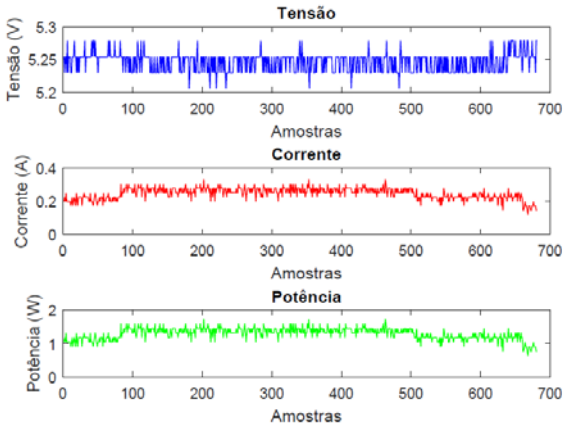


Figura 6.12 - Consumo de energia do *cluster* com 1 processador

Fonte: Elaborada pelos autores

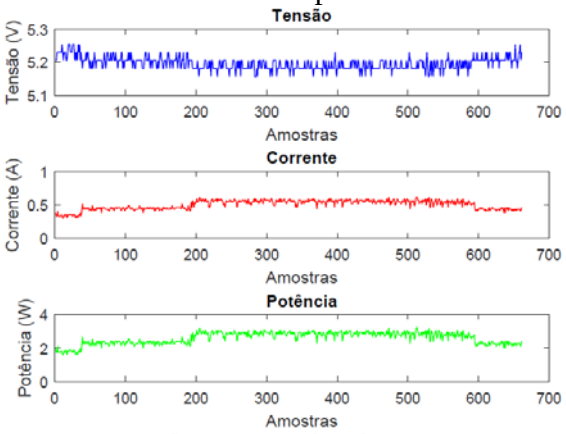


Figura 6.13 - Consumo de energia do *cluster* com 2 processadores
Fonte: Elaborada pelos autores

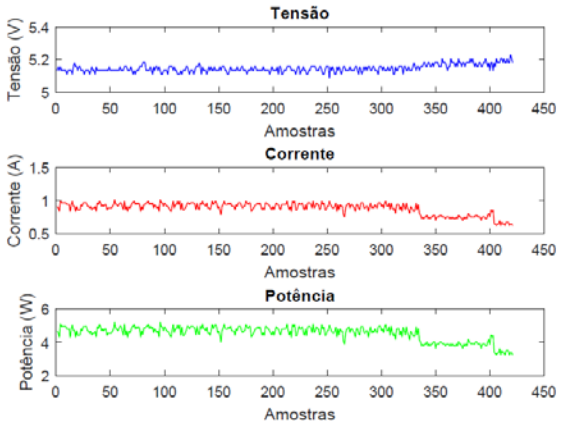


Figura 6.14 - Consumo de energia do *cluster* com 3 processadores
Fonte: Elaborada pelos autores

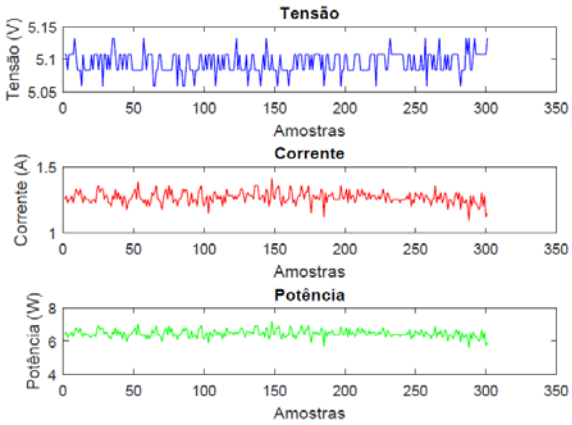


Figura 6.15 - Consumo de energia do *cluster* com 4 processadores
Fonte: Elaborada pelos autores

A análise dos dados coletados durante a execução do HPL mostra que o aumento do número de nós provocou um aumento na potência e na corrente consumidas pelo *cluster*. Os gráficos das Figuras 6.16 e 6.17 apresentam uma comparação dos dados obtidos para o consumo de energia do *cluster*.

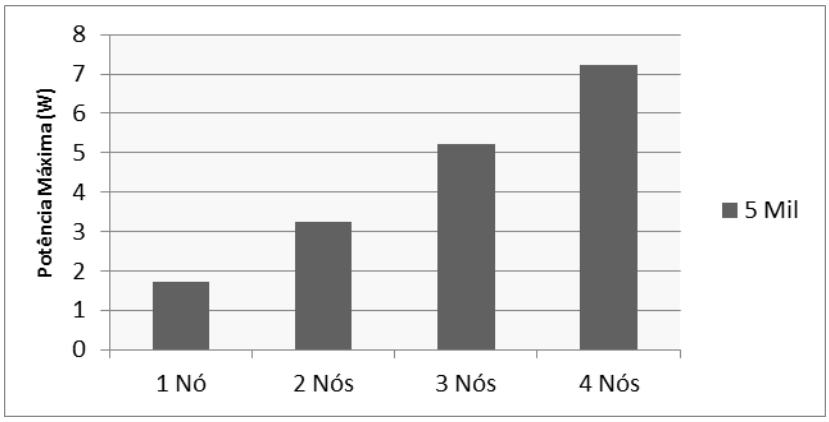


Figura 6.16 - Potência Máxima do *Cluster* com *Benchmark* HPL
Fonte: Elaborada pelos autores

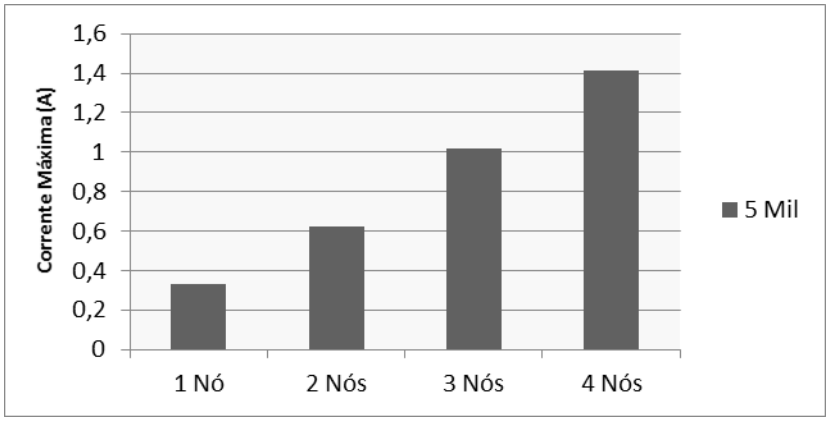


Figura 6.17 - Corrente Máxima do *Cluster* com *Benchmark* HPL
Fonte: Elaborada pelos autores

Analisando os gráficos das Figuras 6.16 e 6.17, pode-se perceber que o aumento do número de processadores provocou um aumento no consumo de energia do *cluster*. Com relação à potência máxima obtida, tem-se que o *cluster* com quatro processadores obteve um consumo 76% maior em relação ao *cluster* com apenas um processador. Já em relação à corrente máxima, o *cluster* com quatro processadores, obteve um consumo 76,63% maior, em relação ao cluster com apenas um processador. As Tabelas 6.7 e 6.8 apresentam os valores para a potência máxima, potência média, corrente máxima e corrente média.

Tabela 6.7 - Potência Máxima e Média

Número de Nós	**Potência Máxima (W)**	**Potência Média (W)**
1	1.7351	1.3210
2	3.2317	2.8395

3	5.2200	4.5292
4	7.2192	6.4615

Fonte: Elaborada pelos autores

Tabela 6.8 - Potência Máxima e Média

Número de Nós	**Corrente Máxima (W)**	**Corrente Média (A)**
1	0.3302	0.2505
2	0.6209	0.5474
3	1.0171	0.8805
4	1.4134	1.2683

Fonte: Elaborada pelos autores

6.3.4. Cenário 2 - Desempenho do *Cluster* com HPL e MPICH-2

Esta seção apresenta o resultado e a análise dos testes realizados para medir o desempenho do *cluster* durante a execução do *benchmark* HPL com a implementação MPICH-2 da biblioteca MPI. O gráfico da Figura 6.18 apresenta o resultado dos testes realizados para N = 5000 e N = 10000. As Tabelas 6.9 e 6.10 apresentam os intervalos de confiança.

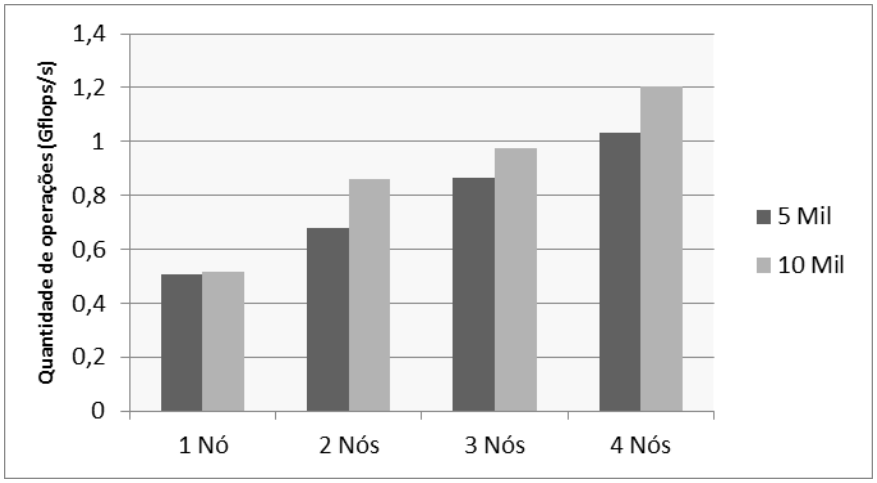


Figura 6.18 - Desempenho do *cluster* com HPL e biblioteca MPICH-2
Fonte: Elaborada pelos autores

Tabela 6.9 - Intervalos de confiança de 95% para N = 5000

Número de Nós	Média	Intervalo de Confiança
1	0,5059	(0,5060 – 0,5064)
2	0,6790	(0,6790 – 0,6820)
3	0,8649	(0,8608 – 0,8690)
4	1,0310	(1,0300 – 1,0310)

Fonte: Elaborada pelos autores

Tabela 6.10 - Intervalos de confiança de 95% para N = 10000

Número de Nós	Média	Intervalo de Confiança
1	0,5180	(0,5170 – 0,5190)
2	0,8604	(0,8567 – 0,8641)
3	0,9772	(0,9762 – 0,9782)
4	1,2050	(1,2045 – 1,2055)

Fonte: Elaborada pelos autores

Após a realização dos testes com o *benchmark* HPL e a implementação MPICH-2 da biblioteca MPI, a análise dos dados apresentados no gráfico da Figura 6.18, permite concluir que o aumento do número de processadores provocou um aumento no desempenho do *cluster*. Esse resultado já era esperado, pois para uma quantidade pequena de dados, o desempenho de um sistema paralelo pode não ser satisfatório, devido ao *overhead* proveniente da comunicação realizada entre os diferentes processadores durante a execução do programa paralelizado. Para N = 5000, a solução paralela com quatro processadores obteve um desempenho 51% melhor, quando comparada a solução sequencial com apenas um processador. Já para N = 10000, a solução com quatro processadores obteve um aumento de 57% no desempenho.

6.3.5. Cenário 2 - Tempo de Execução do *Cluster* com HPL e MPICH-2

Ao analisar os tempos de execução do HPL para N = 5000, pode-se perceber que o aumento do número de processadores do *cluster* provocou uma redução do tempo de

execução do *benchmark*. Os dados apresentados no gráfico da Figura 6.19 mostram que o *cluster* com quatro processadores obteve um desempenho 51% melhor, em relação ao *cluster* com apenas um processador. Em relação ao *speedup*, tem-se que a solução com quatro processadores obteve um desempenho duas vezes melhor, em relação à solução com apenas um processador. O gráfico da Figura 6.20 apresenta os *speedups* do *cluster* embarcado.

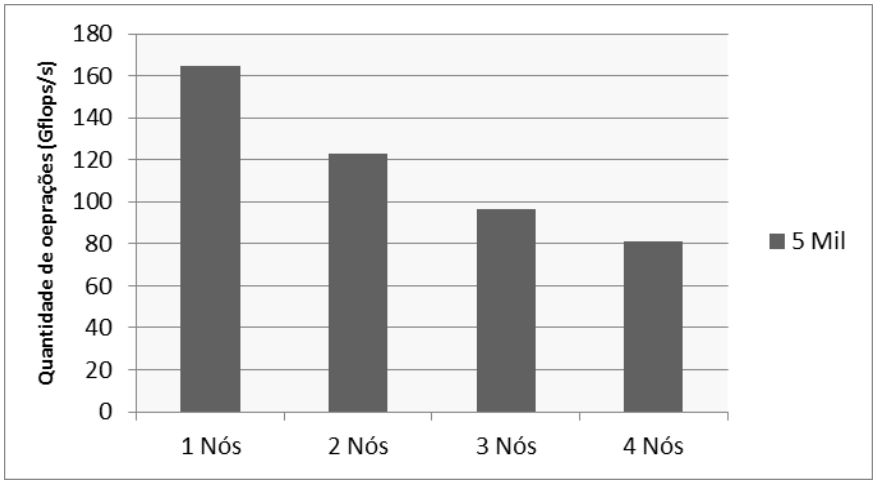


Figura 6.19 - Tempos de Execução do HPL para N = 5000
Fonte: Elaborada pelos autores

Tabela 6.11 - Intervalos de confiança de 95% para N = 5000

Número de Nós	**Média**	**Intervalo de Confiança**
1	164,76	(164,75 – 164,77)
2	122,78	(122,77 – 122,79)
3	96,4	(96,2 – 96,6)
4	80,86	(80,84 – 80,88)

Fonte: Elaborada pelos autores

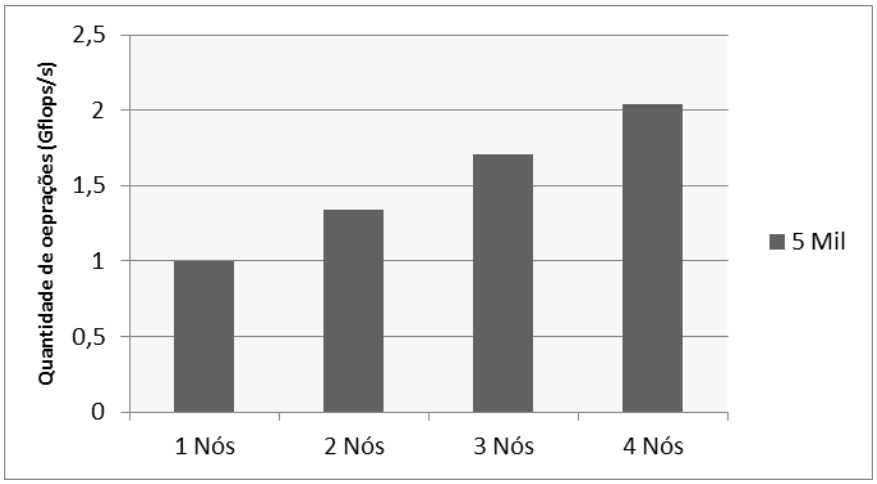


Figura 6.20 - *Speedup* do HPL para N = 5000
Fonte: Elaborada pelos autores

De forma similar ao que foi feito no cenário apresentado na seção anterior, também foram realizados testes que mediram o desempenho do *cluster* com MPICH-2 e HPL com N = 10000. A análise dos dados apresentados no gráfico da Figura 6.21 mostra que o aumento do número de processadores do cluster provocou uma redução no tempo de execução do benchmark HPL. O *cluster* com quatro processadores obteve um desempenho 57% maior em relação ao *cluster* com apenas um processador. Já em relação aos *speedups* apresentados no gráfico da Figura 6.22, pode-se perceber que o *cluster* com quatro processadores obteve um desempenho 2,3 vezes maior em relação ao *cluster* com apenas um processador.

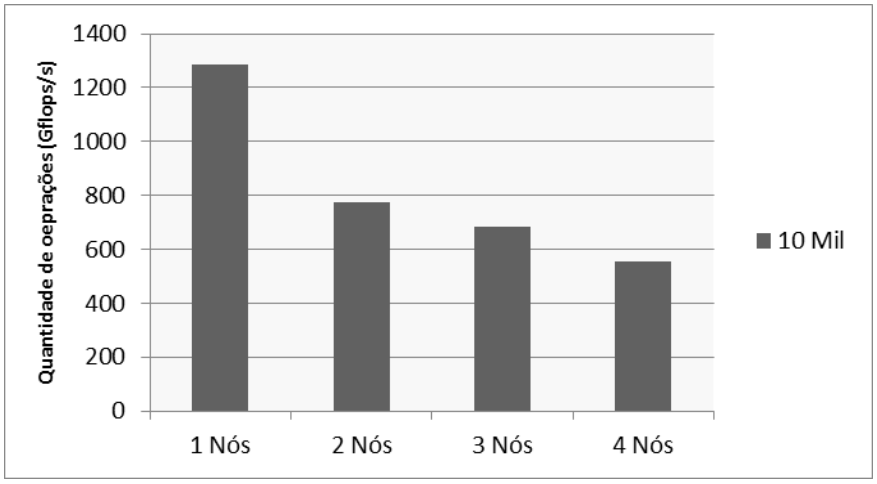


Figura 6.21 - Tempos de Execução do HPL para N = 10000
Fonte: Elaborada pelos autores

Tabela 6.12 - Intervalos de confiança de 95% para N = 10000

Número de Nós	Média	Intervalo de Confiança
1	1287,3	(1287,1 – 1287,5)
2	775,92	(775,91 – 775,93)
3	682,32	(682,31 – 682,32)
4	553,3	(553,29 – 553,3)

Fonte: Elaborada pelos autores

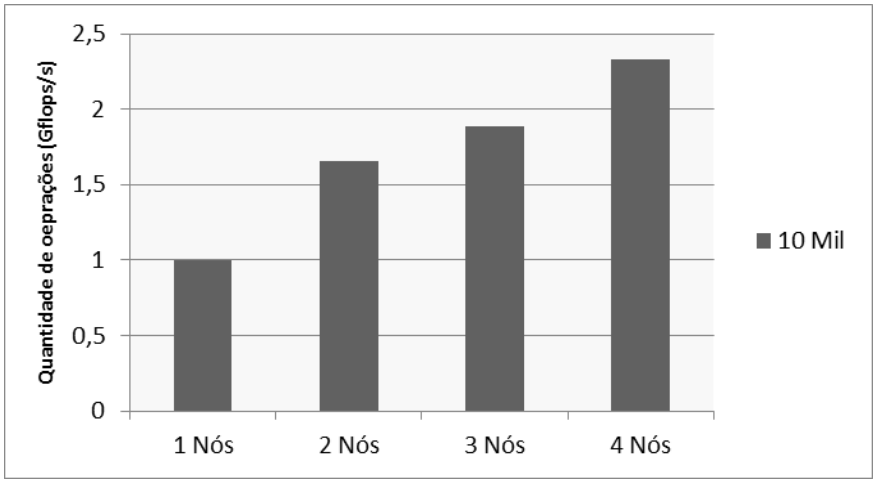


Figura 6.22 - *Speedup* do HPL para N = 10000
Fonte: Elaborada pelos autores

6.3.6. Cenário 2 - Consumo de Energia do *Cluster* com HPL e MPICH-2

Após os testes que mediram o desempenho do *cluster*, também foram realizados testes que mediram o consumo de energia durante a execução do *benchmark* HPL e implementação MPICH-2 da biblioteca MPI. Os gráficos das Figuras 6.23, 6.24, 6.25 e 6.26 apresentam os valores para a tensão, corrente e potência respectivamente.

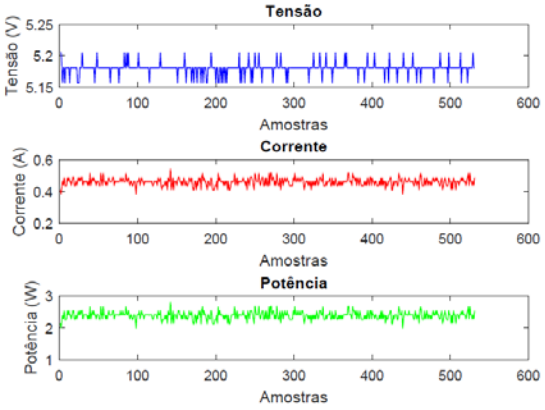


Figura 6.23 - Consumo de energia do *cluster* com 1 processador
Fonte: Elaborada pelos autores

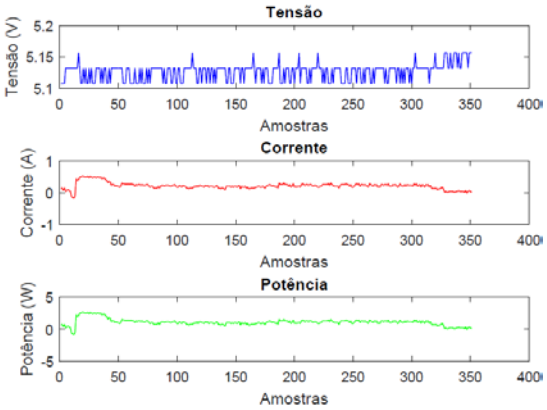


Figura 6.24 - Consumo de energia do *cluster* com 2 processadores
Fonte: Elaborada pelos autores

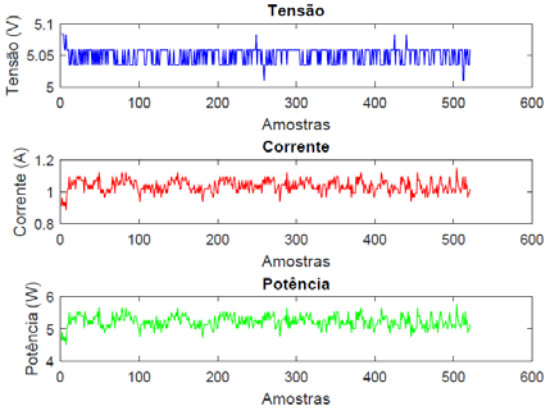


Figura 6.25 - Consumo de energia do *cluster* com 3 processadores
Fonte: Elaborada pelos autores

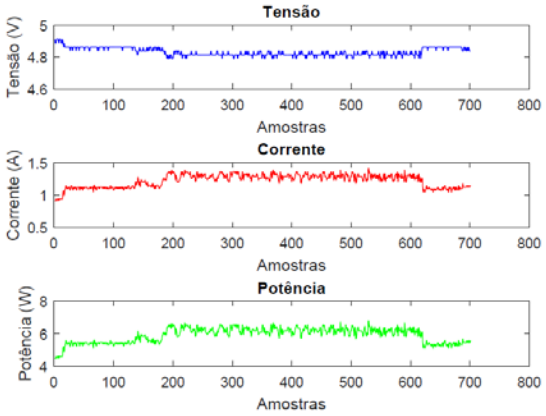


Figura 6.26 - Consumo de energia do *cluster* com 4 processadores
Fonte: Elaborada pelos autores

Em relação ao consumo de energia, pode-se verificar que o aumento do número de processadores do cluster provocou um aumento no consumo de energia. Os gráficos das Figuras 6.27 e 6.28 apresentam uma comparação dos dados obtidos para o consumo de energia do *cluster*.

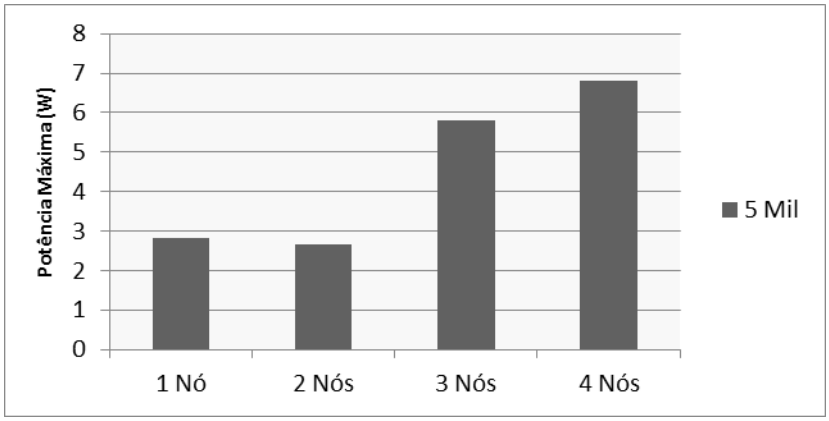


Figura 6.27 - Potência Máxima do *Cluster* com *Benchmark* HPL
Fonte: Elaborada pelos autores

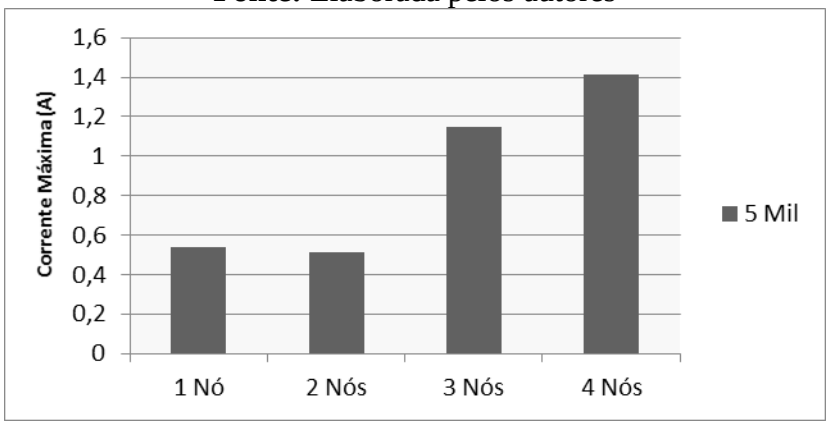


Figura 6.28 - Corrente Máxima do *Cluster* com *Benchmark* HPL
Fonte: Elaborada pelos autores

A análise dos dados comparados apresentados nos gráficos das Figuras 6.27 e 6.28 mostram que o aumento do número de processadores do *cluster* provocou um aumento no consumo de energia. Levando em consideração a potência máxima alcançada, pode-se perceber que o *cluster* com quatro processadores obteve um consumo 58% maior, quando comparado ao *cluster* com apenas um processador. Em relação à corrente máxima, o *cluster* com quatro processadores obteve um consumo 62% maior.

Tabela 6.13 - Potência Máxima e Média

Número de Nós	**Potência Máxima (W)**	**Potência Média (W)**
1	2.8059	2.3782
2	2.6439	1.0774
3	5.7855	5.1932
4	6.8047	5.8923

Fonte: Elaborada pelos autores

Tabela 6.14 - Corrente Máxima e Média

Número de Nós	**Corrente Máxima (W)**	**Corrente Média (A)**
1	0.5416	0.4591
2	0.5152	0.2103
3	1.1492	1.0349
4	1.4134	1.2197

Fonte: Elaborada pelos autores

6.3.7. Análise Comparativa entre os Cenários 1 e 2

As seções anteriores apresentaram os resultados das análises de desempenho do *cluster* embarcado com processadores ARM e *benchmark* HPL. Assim, foi possível verificar o comportamento do *cluster* durante a resolução de problemas de diferentes tamanhos e em diferentes ambientes de programação. Esta seção faz uma análise comparativa do desempenho do *cluster*, a fim de verificar qual das implementações da biblioteca MPI obteve o melhor desempenho.

Levando em consideração a resolução de sistemas lineares com cinco mil e dez mil variáveis, pode-se observar que o *cluster* implementado com a biblioteca OpenMPI obteve um melhor desempenho, quando comparado ao *cluster* implementado com a biblioteca MPICH-2. Os gráficos das Figuras 6.29 e 6.30 apresentam os dados comparados.

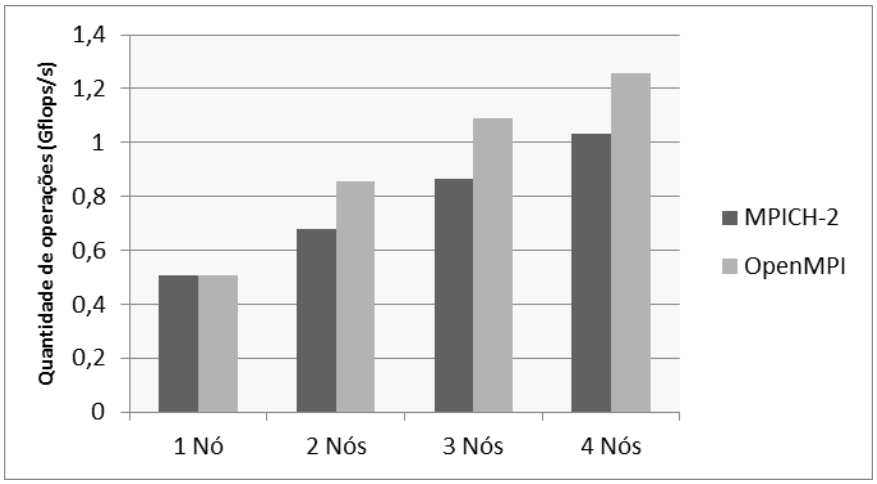


Figura 6.29 - Comparação das bibliotecas OpenMPI e MPICH-2 com N = 5000
Fonte: Elaborada pelos autores

A análise dos dados apresentados no gráfico da Figura 6.29 mostra que para a solução de sistemas lineares com cinco mil variáveis, o desempenho do *cluster* composto por quatro processadores e com a biblioteca OpenMPI foi 18% maior, quando comparado ao *cluster* com a biblioteca MPICH-2.

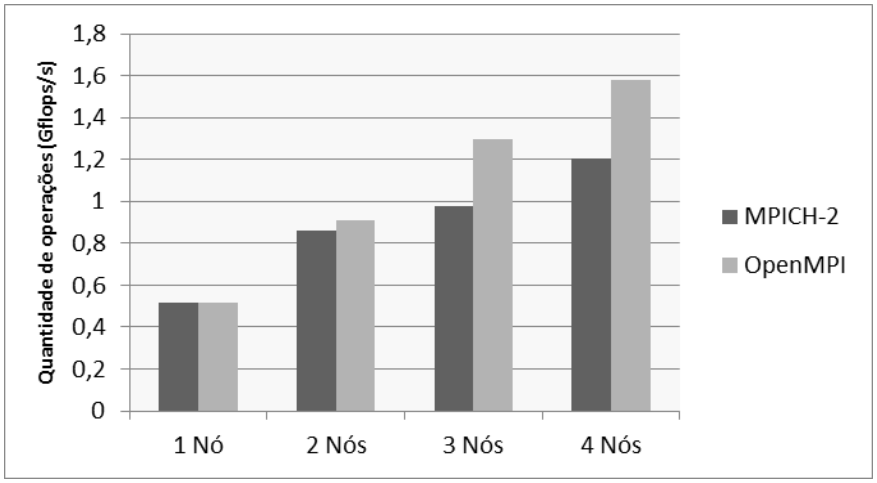


Figura 6.30 - Comparação das bibliotecas OpenMPI e MPICH-2 com N = 10000
Fonte: Elaborada pelos autores

Para a resolução de sistemas lineares com dez mil variáveis, a análise dos dados apresentados no gráfico da Figura 6.30 mostra que o *cluster* com a biblioteca OpenMPI

também obteve um melhor desempenho, quando comparado ao *cluster* com a biblioteca MPICH-2. Para quatro processadores, o aumento no desempenho foi de 24%.

Em relação ao tempo gasto pelo *cluster* para resolver um sistema linear com cinco mil variáveis, tem-se que o *cluster* implementado com a biblioteca OpenMPI obteve um desempenho superior, quando comparado ao *cluster* implementado com a biblioteca MPICH-2. A análise dos dados apresentados nos gráficos das Figuras 6.31 e 6.32 mostram que o *cluster* com quatro processadores e biblioteca OpenMPI obteve um tempo de execução 18% menor, em relação ao *cluster* com MPICH-2. Além disso, quando comparado ao *cluster* com apenas um processador, o *cluster* com OpenMPI alcançou um *speedup* de 2,5, enquanto o *cluster* com MPICH-2 alcançou um *speedup* de 2.

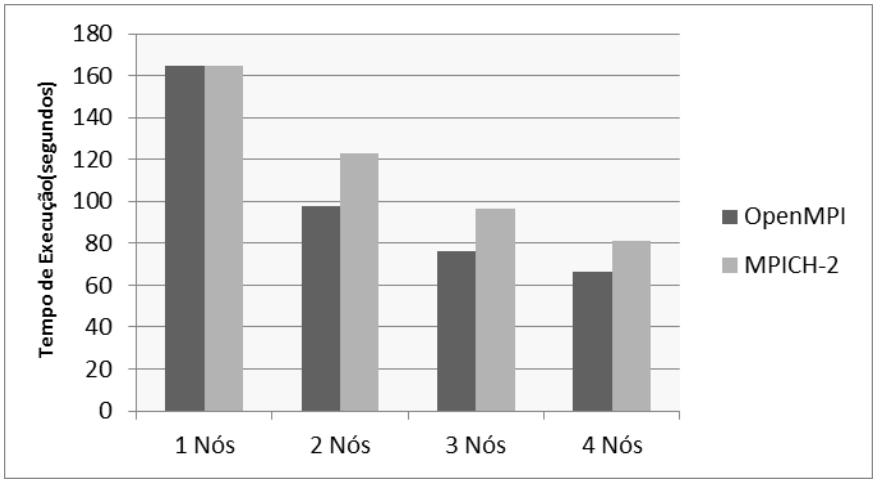


Figura 6.31 - Tempos das bibliotecas OpenMPI e MPICH-2 com N = 5000
Fonte: Elaborada pelos autores

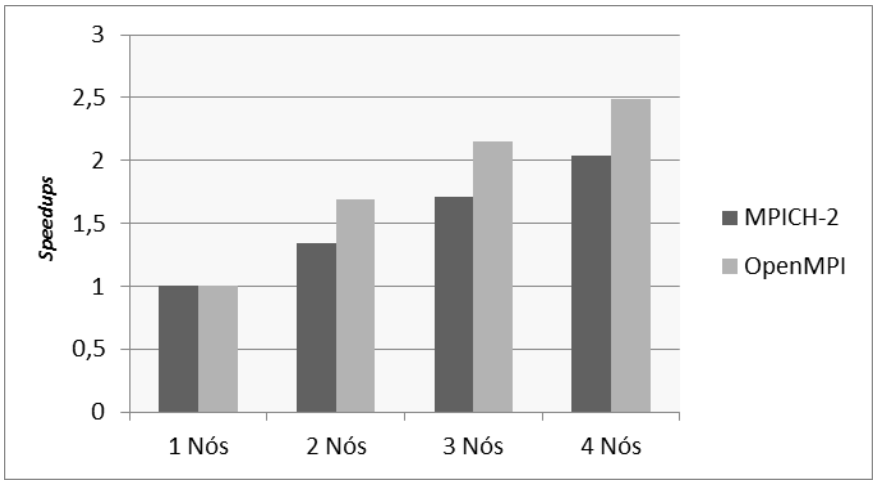


Figura 6.32 - *Speedups* das bibliotecas OpenMPI e MPICH-2 com N = 5000
Fonte: Elaborada pelos autores

Em relação ao comportamento do *cluster* durante a resolução de um sistema linear com dez mil variáveis, tem-se que o *cluster* implementado com a biblioteca OpenMPI também obteve um melhor desempenho. A análise dos dados apresentados nos gráficos das Figuras 6.33 e 6.34 mostram que para o *cluster* com quatro processadores, o tempo de execução com a biblioteca OpenMPI foi 24% menor, em relação ao *cluster* com a biblioteca MPICH-2.

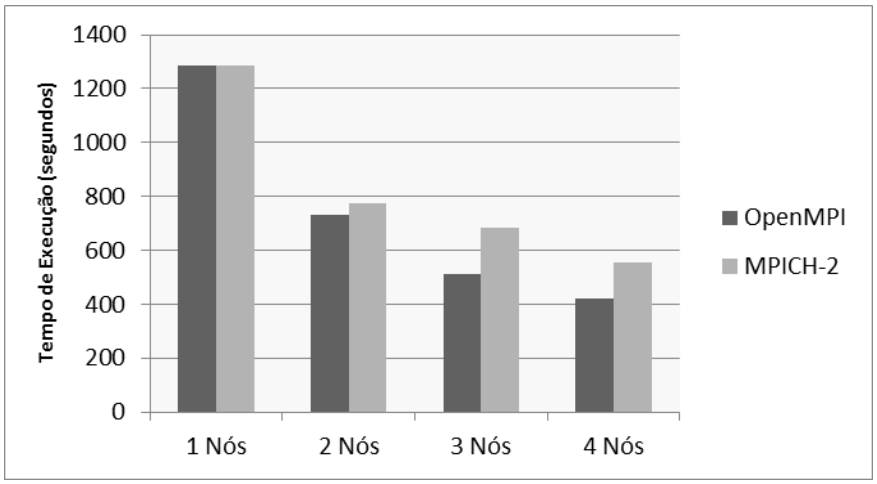


Figura 6.33 - Tempos das bibliotecas OpenMPI e MPICH-2 com N = 10000
Fonte: Elaborada pelos autores

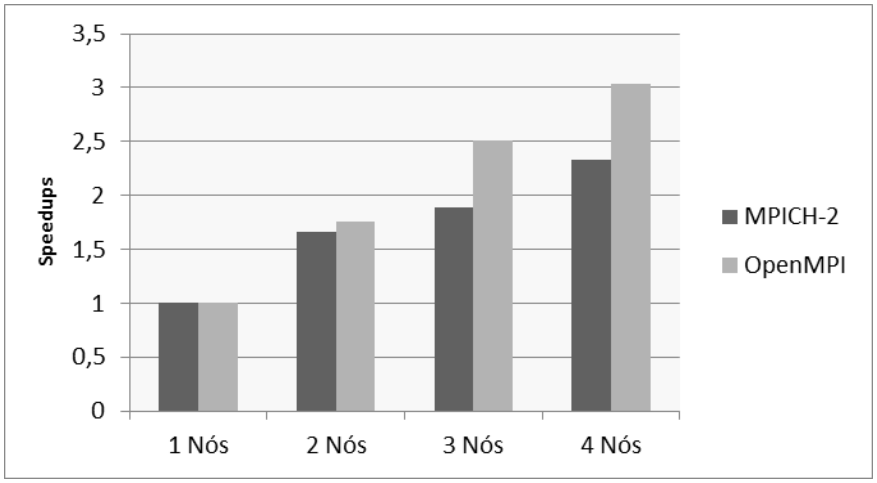


Figura 6.34 - *Speedups* das bibliotecas OpenMPI e MPICH-2 com N = 10000
Fonte: Elaborada pelos autores

Além da capacidade de processamento e do tempo de execução, também foi medido o consumo de energia do *cluster* com as bibliotecas OpenMPI e MPICH-2. A analise dos dados apresentados nos gráficos 6.35 e 6.36 mostram que a potência

máxima alcançada pelo *cluster* com a biblioteca OpenMPI foi 5,7% maior, quando comparada ao *cluster* com a biblioteca MPICH-2. Assim, pode-se concluir que o aumento no poder de processamento do *cluster* com a biblioteca OpenMPI fez com que o *cluster* consumisse mais energia durante o processamento.

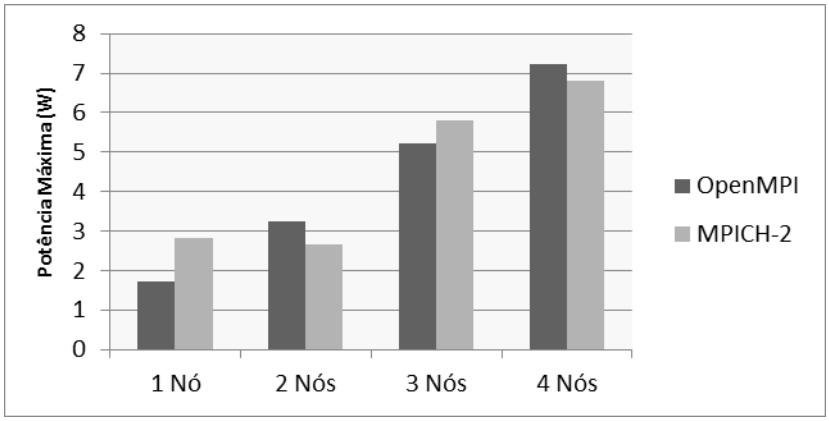


Figura 6.35 - Potência Máxima com N = 5000
Fonte: Elaborada pelos autores

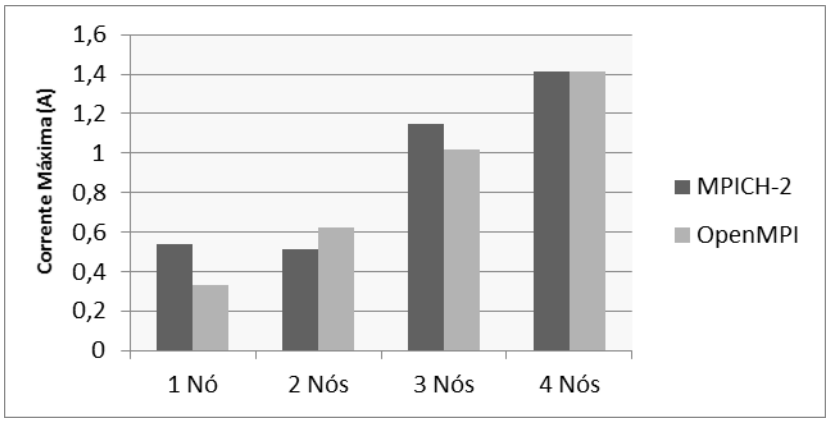


Figura 6.36 - Corrente Máxima com N = 5000
Fonte: Elaborada pelos autores

6.3.8. Cenário 3 - Desempenho do *Cluster* com HPCC com OpenMPI e MPICH-2

Com o *benchmark* FFT, foi possível verificar a capacidade de processamento do *cluster* após a realização do cálculo da Transformada Rápida de Fourier. Os dados coletados durante os testes são apresentados no gráfico da Figura 6.37.

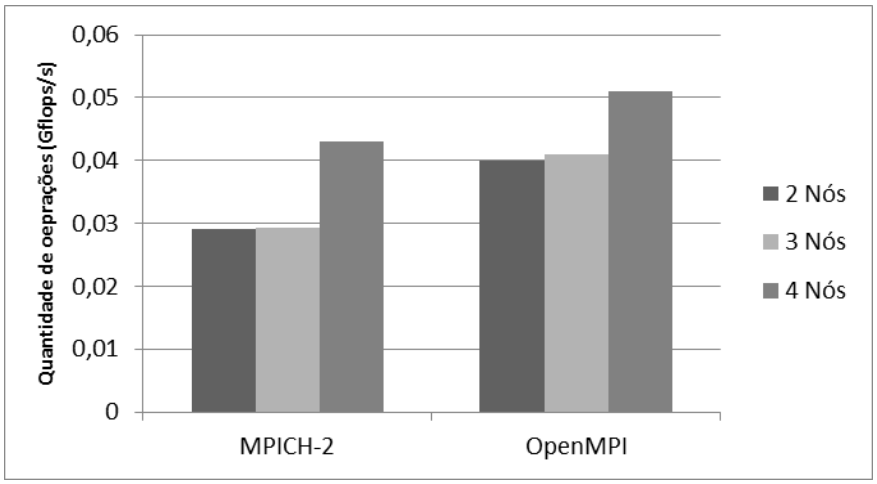


Figura 6.37 - *Benchmark* FFT
Fonte: Elaborada pelos autores

A análise dos dados apresentados na Figura 6.37 mostram que o *cluster* com o *benchmark* FFT e implementação OpenMPI da biblioteca MPI alcançou um melhor desempenho. Para o *cluster* composto por quatro processadores, pode-se perceber que a implementação OpenMPI obteve um desempenho 15% maior, em relação a implementação MPICH-2.

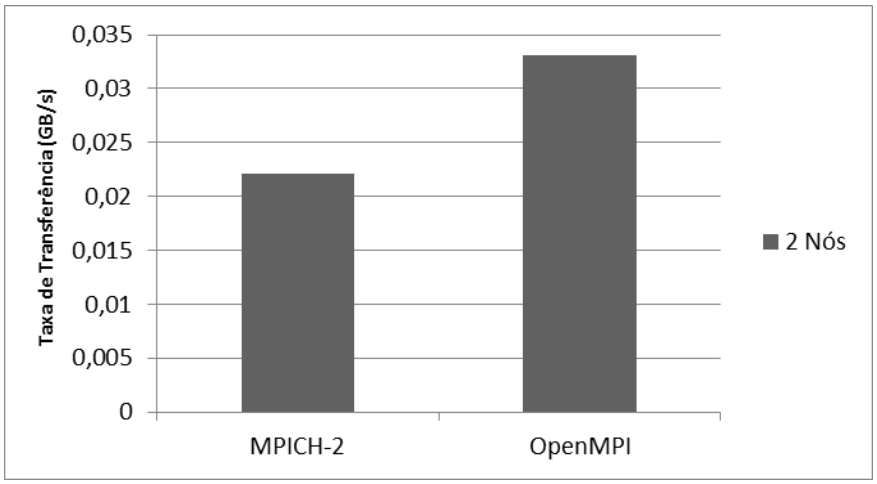


Figura 6.38 - *Benchmark* PTRANS
Fonte: Elaborada pelos autores

Em relação ao *benchmark* PTRANS, que realiza a medição da taxa de transferência de dados da memória, a análise dos dados apresentados na Figura 6.38 mostra que a taxa de transferência foi 33% maior quando a biblioteca OpenMPI foi utilizada.

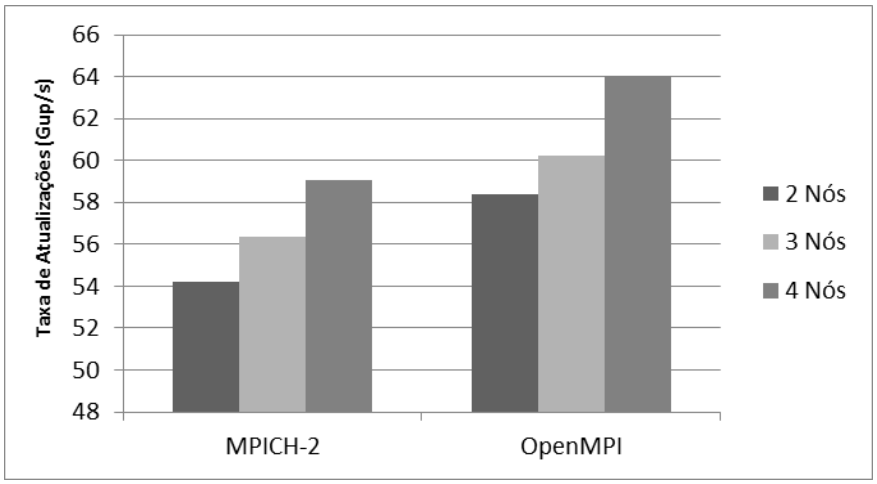


Figura 6.39 - *Benchmark* RandomAccess
Fonte: Elaborada pelos autores

O *benchmark* RandomAccess mede a taxa de atualização de dados na memória. Para isso, um inteiro é lido da memória, atualizado no processador e em seguida é escrito na mesma posição de memória. Assim, a análise dos dados apresentados no gráfico da Figura 6.39 mostra que para um *cluster* com quatro processadores, a taxa de atualizações da memória foi 8% maior quando a biblioteca OpenMPI foi utilizada.

Com o *benchmark* DGEMM, foi possível medir a taxa de execução de operações com números de ponto flutuante com precisão dupla durante a multiplicação de matrizes formadas por números reais. A análise dos dados apresentados no gráfico da Figura 6.40 mostra que o *cluster* com a implementação OpenMPI obteve um desempenho 7,7% maior, em relação ao *cluster* com a implementação MPICH-2.

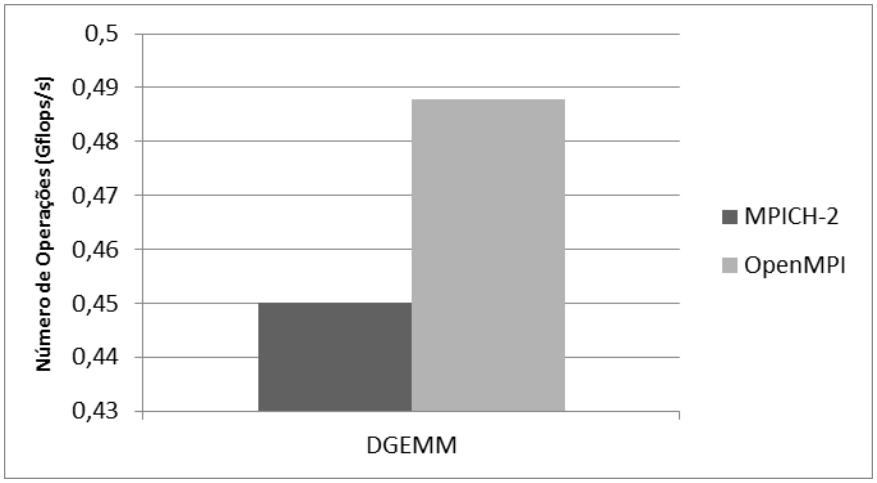


Figura 6.40 - *Benchmark* DGEMM
Fonte: Elaborada pelos autores

Com o *benchmark* b_eff foi possível medir a largura de banda utilizada durante a comunicação entre as placas. A análise do gráfico apresentado na Figura 6.41 mostra que a utilização da biblioteca OpenMPI obteve uma largura de banda 8% maior, em relação a biblioteca MPICH-2.

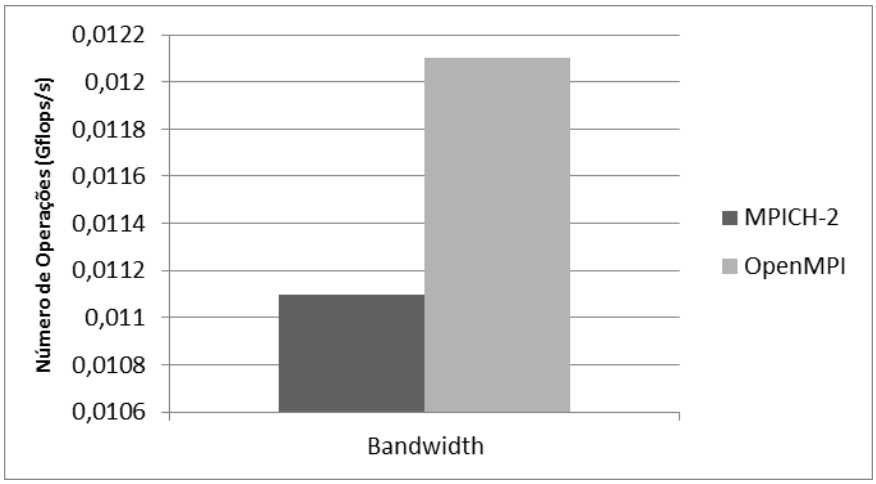


Figura 6.41 - *Benchmark* Bandwidth
Fonte: Elaborada pelos autores

6.3.9. Cenário 3 - Consumo de Energia do Cluster com HPCC e OpenMPI

Neste cenário de testes, foi analisado o consumo de energia do cluster durante a execução do benchmark HPCC e biblioteca OpenMPI. Os gráficos das Figuras 6.42, 6.43 e 6.44 apresentam os resultados dos testes para a tensão, corrente e potência.

Levando em consideração a potência máxima obtida durante os testes, pode-se perceber que o cluster com quatro processadores obteve um consumo 13% maior, em relação ao *cluster* com apenas dois processadores.

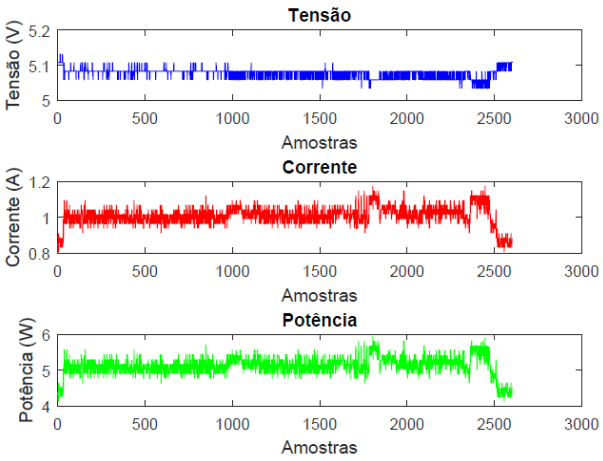


Figura 6.42 - Consumo de energia HPCC e OpenMPI com 2 nós
Fonte: Elaborada pelos autores

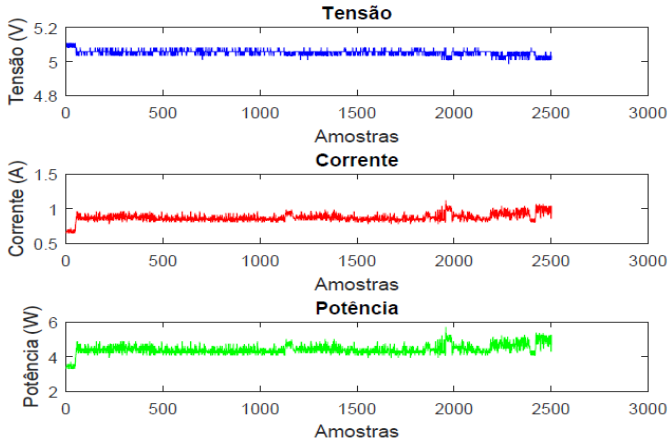


Figura 6.43 - Consumo de energia HPCC e OpenMPI com 3 nós
Fonte: Elaborada pelos autores

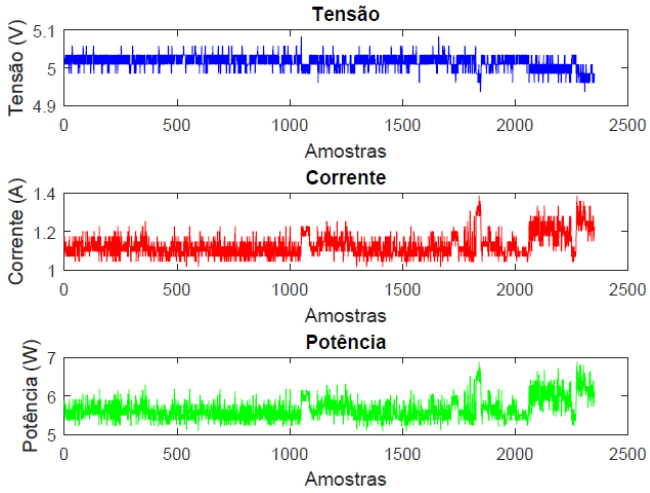


Figura 6.44 - Consumo de energia HPCC e OpenMPI com 4 nós
Fonte: Elaborada pelos autores

Tabela 6.15 - Potência Máxima e Média OpenMPI

Número de Nós	Potência Máxima (W)	Potência Média (W)
2	5,9473	5,1149
3	5,7074	4,4146
4	6,8809	5,6557

Fonte: Elaborada pelos autores

Tabela 6.16 - Corrente Máxima e Média OpenMPI

Número de Nós	Corrente Máxima (A)	Corrente Média (A)
2	1,1759	1,0075

3	1,1228	0,8736
4	1,3870	1,1276

Fonte: Elaborada pelos autores

6.3.10. Cenário 3 - Consumo de Energia do Cluster com HPCC e MPICH-2

Além de medir e analisar a capacidade de processamento do *cluster* com o *benchmark* HPCC, também foram realizados testes que mediram o consumo de energia do cluster durante a execução do *benchmark* HPCC. Os gráficos das Figuras 6.45, 6.46 e 6.47 apresentam os valores obtidos durante os testes para a tensão, corrente e potência.

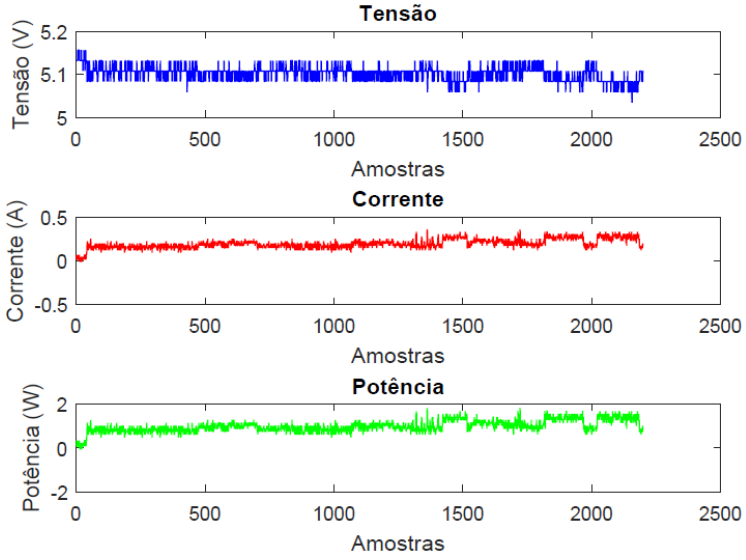


Figura 6.45 - Consumo de energia HPCC e MPICH-2 com 2 nós
Fonte: Elaborada pelos autores

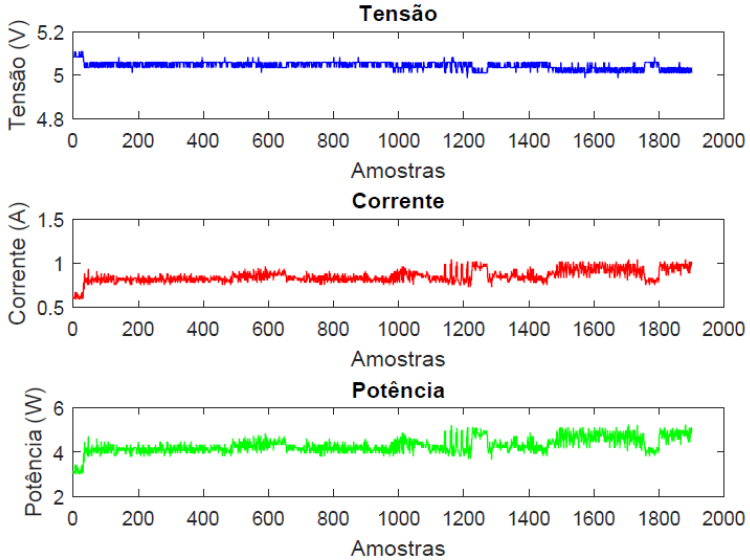


Figura 6.46 - Consumo de energia HPCC e MPICH-2 com 3 nós
Fonte: Elaborada pelos autores

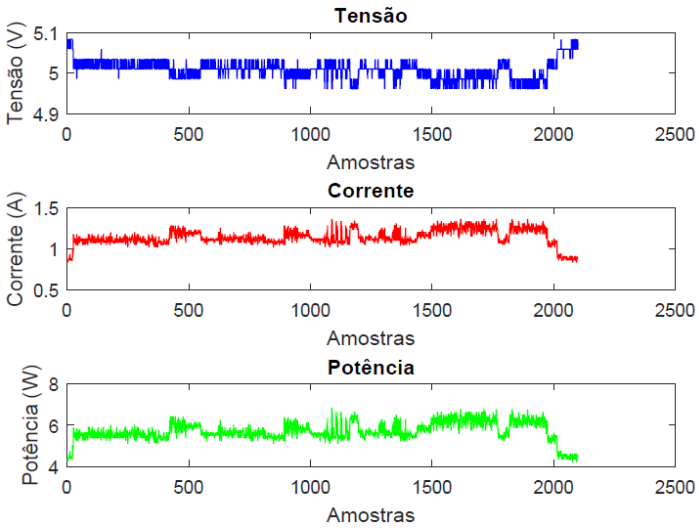


Figura 6.47 - Consumo de energia HPCC e MPICH-2 com 4 nós
Fonte: Elaborada pelos autores

A análise dos dados apresentados nos gráficos das Figuras 6.45, 6.46 e 6.47 mostra que a potência máxima obtida pelo *cluster* com quatro nós foi 73% maior, em relação ao *cluster* com apenas dois processadores.

Tabela 6.17 - Potência Máxima e Média MPICH-2

Número de Nós	Potência Máxima (W)	Potência Média (W)
2	1,8304	0,9919
3	5,2535	4,3007
4	6,8495	5,6991

Fonte: Elaborada pelos autores

Tabela 6.18 - Potência Máxima e Média MPICH-2

Número de Nós	Corrente Máxima (A)	Corrente Média (A)
2	0,3567	0,1945
3	1,0436	0,8530
4	1,3606	1,1383

Fonte: Elaborada pelos autores

Portanto, após a realização dos testes com o *benchmark* HPCC e as implementações OpenMPI e MPICH-2 das bibliotecas MPI, pode-se perceber que a biblioteca MPICH-2 obteve um menor consumo de energia. Os gráficos das Figuras 6.48 e 6.49 apresentam os resultados comparados.

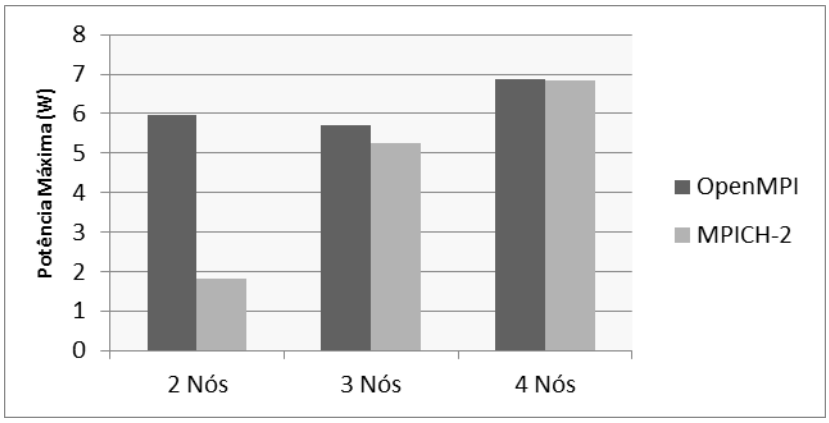


Figura 6.48 - Potência Máxima com N = 5000
Fonte: Elaborada pelos autores

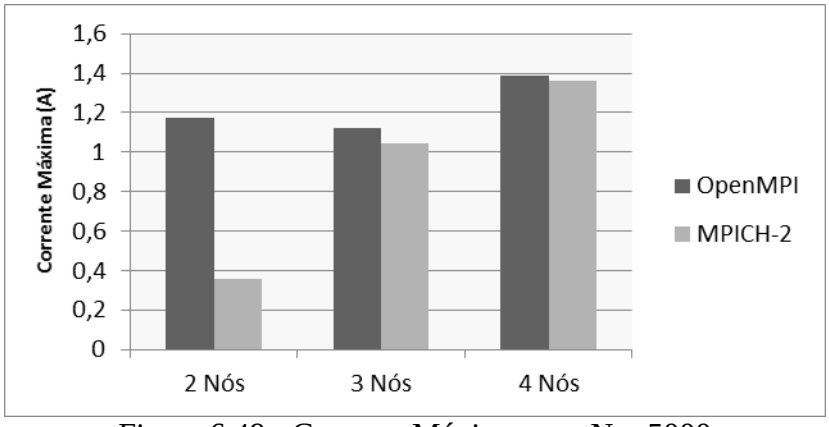


Figura 6.49 - Corrente Máxima com N = 5000
Fonte: Elaborada pelos autores

Portanto, após a realização dos testes com o *benchmark* HPCC e as implementações OpenMPI e MPICH-2 das bibliotecas MPI, pode-se perceber que a biblioteca MPICH-2 obteve um menor consumo de energia. Os gráficos das Figuras 6.50 e 6.51 apresentam os resultados comparados.

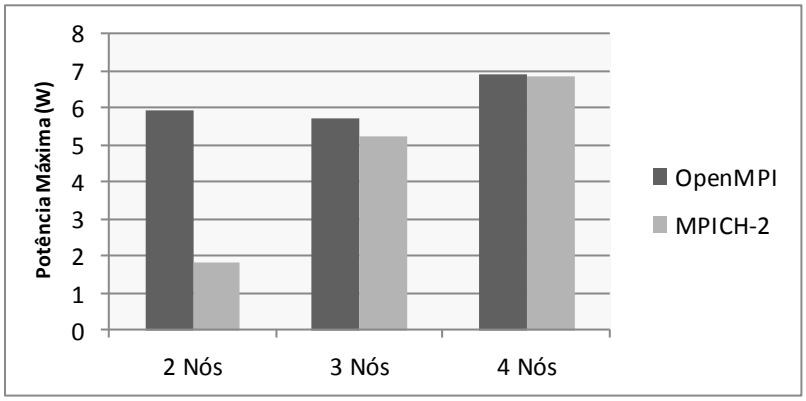


Figura 6.50 - Potência Máxima com N = 5000
Fonte: Elaborada pelos autores

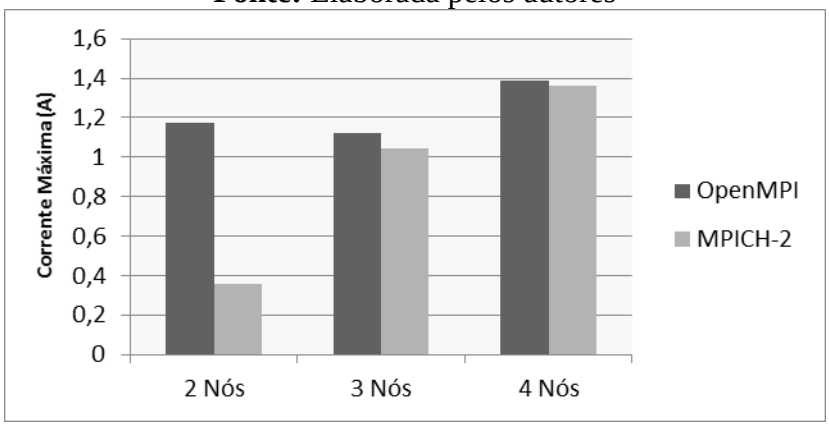


Figura 6.51 - Corrente Máxima com N = 5000
Fonte: Elaborada pelos autores

6.4. Considerações Finais

Este capitulo de livro teve como objetivo apresentar a implementação de um *cluster* embarcado com processadores ARM e a platorma Raspberry Pi e assim analisar o seu desempenho. Para isso, foram criados três cenários de testes que levaram em consideração os *benchmarks* HPL e HPCC, além das bibliotecas de processamento paralelo OpenMPI e MPICH-2.

Após a realização dos testes, foi possível verificar que em relação à capacidade de processamento do *cluster* embarcado com a biblioteca OpenMPI e *benchmark* HPL, o aumento do número de variáveis do sistema linear, de cinco mil para dez mil, fez com que o *cluster* alcançasse um melhor desempenho. Para cinco mil variáveis, o *cluster* com quatro processadores obteve um desempenho 59,78% melhor em relação ao *cluster* com apenas um processador. Já para o *cluster* com quatro processadores e um sistema linear com dez mil variáveis, o aumento no desempenho foi de 67%. O *speedup* do *cluster* com quatro processadores e um sistema linear com dez mil variáveis foi maior,

quando comparado ao *cluster* com quatro processadores e um sistema linear com cinco mil variáveis. Quando a biblioteca MPICH-2 foi utilizada, observou-se um comportamento semelhante, pois, para um sistema linear com cinco mil variáveis e um *cluster* com quatro processadores, o aumento no desempenho foi de 51%, já para um sistema linear com dez mil variáveis, o aumento foi de 57%. Assim, pode-se perceber que o *cluster* não terá um bom desempenho ao resolver problemas computacionais simples, pois o *overhead* existente na comunicação entre os processadores pode prejudicar o desempenho total.

Após a realização dos testes com as bibliotecas OpenMPI e MPICH-2, foi possível verificar que a biblioteca OpenMPI permitiu que o *cluster* alcançasse um melhor desempenho. O *cluster* com quatro processadores e biblioteca OpenMPI obteve um desempenho 18% maior durante a resolução de um sistema linear com cinco mil variáveis, e 24% maior durante a resolução de um sistema linear com dez mil variáveis.

Com o *benchmark* HPCC também foram realizados testes que mediram o desempenho do *cluster* durante a sua execução. A análise comparada mostra que o *cluster* com a implementação OpenMPI da biblioteca MPI obteve um melhor desempenho, assim como ocorreu com o *benchmark* HPL.

No capitulo, também foi abordado os testes que mediram o consumo de energia real do *cluster* embarcado durante a execução dos *benchmarks* e com diferentes implementações da biblioteca MPI. A análise comparada dos resultados obtidos, permite concluir que apesar da biblioteca OpenMPI melhorar significativamente o desempenho do *cluster*, ela fez com que o consumo de energia aumentasse, pois o aumento da capacidade de processamento do cluster veio acompanhado do aumento da potência gasta durante a execução dos *benchmarks*.

Referências

Rauber, T., Rünger, G. Parallel Programming: for Multicore and Cluster Systems 2nd ed. 2013 Edition

Moore, G. E. "Cramming more components onto integrated circuits". Electronics,Vol. 38, nº 8, Abril 1965.

Blume, H. *et al*. "OpenMP-based parallelization on an MPCore multiprocessor platform – A performance and power analysis". Journal of Systems Architecture 54.

MPI. Disponível em: <http://www.open-mpi.org/>. Acessado em: 20 de junho de 2016.

OpenMP. Disponível em: <http://openmp.org/wp/>. Acessado em: 17 de Junho de 2016.

Bez et al. Análise da Eficiência Energética de uma Aplicação HPC de Geofísica em um Cluster de Baixo Consumo. WSCAD 2015.

Reed, D. A. and Dongarra, J. (2015). Exascale Computing and Big Data. Communication of the ACM, 58(7):56–68.

Wazlawick, R. Metodologia de Pesquisa para Ciência da Computação. 6º Edição.

Lima, F. et al. Design and Performance of a Low Cost *Cluster* using ARM-based Platform. PDPTA 2016

Padoin, E. L., Velho, P., de Oliveira, D. A. G., Navaux, P. O. A., and Mehaut, J.-F. (2014). Tuning Performance and Energy Consumption of HPC Applications on ARM MPSoCs.

HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. Disponível em: <http://www.netlib.org/benchmark/hpl/> Acessado em: 20 de julho de 2016.

Freund, E. Estatística aplicada: economia, administração e contabilidade – 11º Edição. Bookman, 2006

Larson, R. Estatística aplicada – 4º Edição. São Paulo: Pearson Prentice Hall, 2010.

Pacheco, Peter S. "An introduction to parallel programming".

Silberschatz, Abraham, *et al*. "Sistemas Operacionais – Conceitos e Aplicações". 2013.

Gebali, Fayez. "Algorithms and Parallel Computing". 2011.

Jin, Haoqiang. "High performance computing using MPI and OpenMP on multi-core parallel systems". 2011.

Raspberry Pi. Disponível em: <http://www.raspberrypi.org/>. Acessado em: 15 de Março de 2016.

Raspbian. Disponível em: <http://www.raspbian.org/>. Acessado em: 15 de Março de 2016.

ARM Company Profile. Disponível em: <http://www.arm.com/about/company-profile/index.php>. Acessado em 01 de Junho de 2015.

Neill, Alexander Shabarshin Richard, and Carloni, Luca P. "A heterogeneous parallel system running OpenMPI on a broadband network of embedded set-top devices". In Proceedings of the 7th ACM International Conference on Computing Frontiers (CF '10), New York, NY, USA, pages 187-196, 2010.

McCool, Michael *et al*. "Structured Parallel Programming – Patterns for Efficient Computation".

Zomaya, Albert Z. *et al.* "Energy Efficient Distributed Computing Systems". 2012.

Cox, Simon J. *et al.* "Iridis-pi: a low-cost, compact demonstration cluster". Cluster Computing, June 2013.

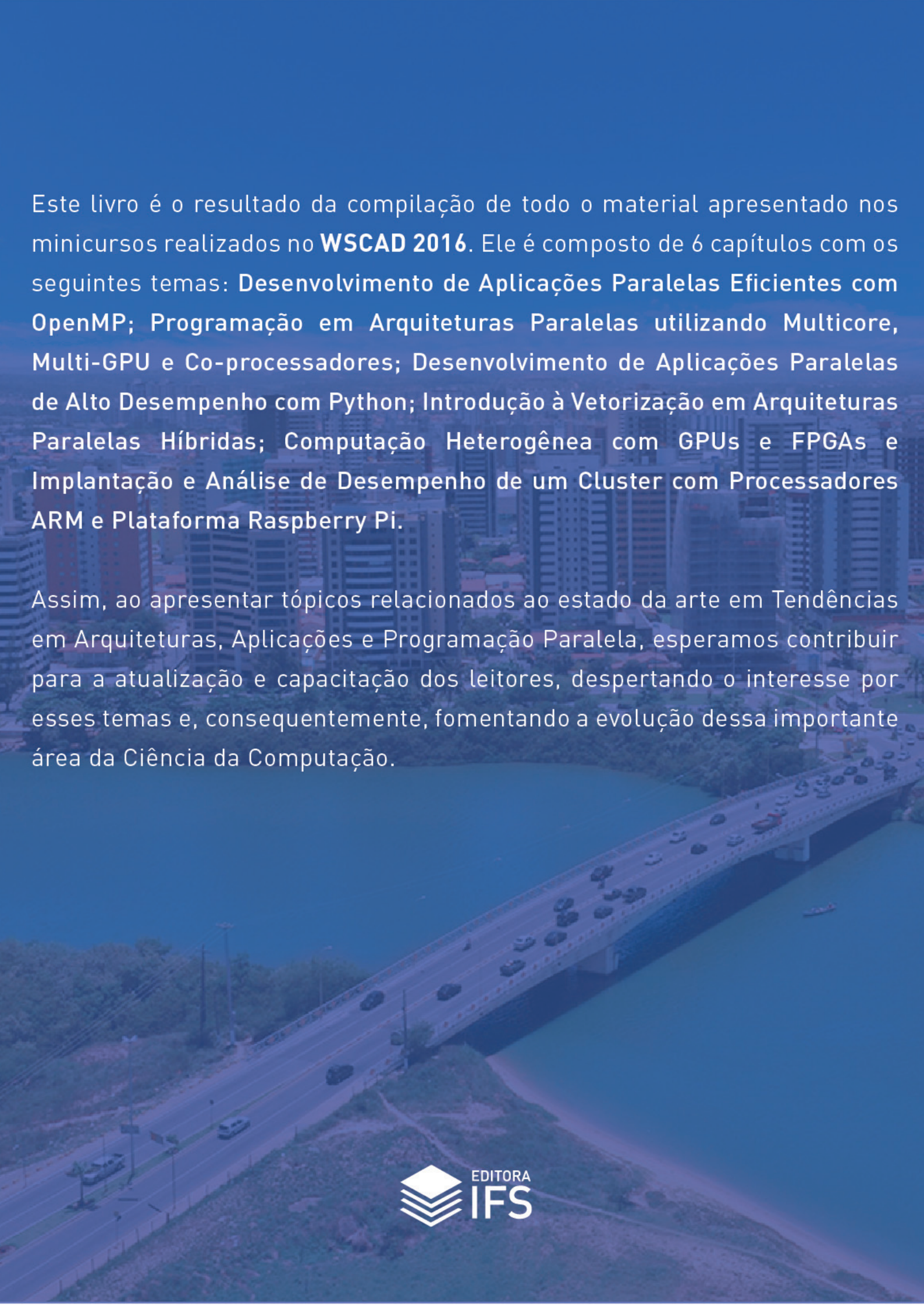
Silva, Renato *et al*. “avaliação do desempenho de Threads em user level utilizando sistema operacional Linux”. Revista de Informática Teórica e Aplicada, 2011.

Furlinger, Karl et al. The AppleTV-Cluster: Towards Energy Efficient Parallel Computing on Consumer electronic Devices.

H. Blume et al., “OpenMP-based parallelization on an MPCore multiprocessor platform – A performance and power analysis”, Journal of Systems Architecture 54, 2008.

Este livro é o resultado da compilação de todo o material apresentado nos minicursos realizados no **WSCAD 2016**. Ele é composto de 6 capítulos com os seguintes temas: **Desenvolvimento de Aplicações Paralelas Eficientes com OpenMP; Programação em Arquiteturas Paralelas utilizando Multicore, Multi-GPU e Co-processadores; Desenvolvimento de Aplicações Paralelas de Alto Desempenho com Python; Introdução à Vetorização em Arquiteturas Paralelas Híbridas; Computação Heterogênea com GPUs e FPGAs e Implantação e Análise de Desempenho de um Cluster com Processadores ARM e Plataforma Raspberry Pi.**

Assim, ao apresentar tópicos relacionados ao estado da arte em Tendências em Arquiteturas, Aplicações e Programação Paralela, esperamos contribuir para a atualização e capacitação dos leitores, despertando o interesse por esses temas e, consequentemente, fomentando a evolução dessa importante área da Ciência da Computação.



EDITORA IFS