Universidade Federal do Pampa
Adriano Marques Garcia

Classificação de um *Benchmark* Paralelo para Arquiteturas Multinúcleo

Alegrete
2016

Adriano Marques Garcia

Classificação de um *Benchmark* Paralelo para Arquiteturas Multinúcleo

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Márcia Cristina Cera

Coorientador: Arthur Francisco Lorenzon

Alegrete

2016

Adriano Marques Garcia

Classificação de um *Benchmark* Paralelo para Arquiteturas Multinúcleo

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Trabalho de Conclusão de Curso defendido e aprovado em 28 de Novembro de 2016.

Banca examinadora:

Márcia C. Cera

Márcia Cristina Cera
Orientador

P/ Márcia C. Cera - participou via sistema on-line.

Arthur Francisco Lorenzon
Co-orientador
UFRGS

Aline Vieira de Mello

Aline Vieira de Mello
UNIPAMPA

Cláudio Schepke

Cláudio Schepke
UNIPAMPA

Resumo

Este trabalho visa classificar um conjunto de treze aplicações utilizadas para medir o desempenho e o consumo de energia em arquiteturas multinúcleo. Essas aplicações estão paralelizadas utilizando quatro interfaces de programação paralela: OpenMP, Pthreads, MPI-1 e MPI-2. Elas foram desenvolvidas em estudos anteriores e classificadas de acordo com alguns critérios, como: quantidade de operações realizadas na memória, taxa de comunicação e quantidade de operações de troca de dados entre *threads*/processos. Este trabalho realiza uma série de testes nessas aplicações com o objetivo de classificá-las e mostrar que possuem características suficientes para serem utilizadas como um *benchmark* para avaliar o desempenho e o consumo de energia com diferentes interfaces de programação paralela em arquiteturas multinúcleo. As aplicações foram classificadas de acordo com: estruturas de dados, complexidades, TLP, uso de CPU e uso de memória. As treze aplicações e suas implementações com quatro IPPs abrangem os critérios de classificação de tal modo que diversos cenários possam ser avaliados, mostrando-se eficiente como *benchmark* para análise de desempenho e consumo de energia de arquiteturas multinúcleo.

Palavras-chave: *Benchmark*. Consumo de energia. Alto Desempenho. Programação paralela. Multinúcleo.

Abstract

This work aims to classify a set of thirteen applications used to measure performance and energy power consumption in multicore architectures. These applications are parallelized using four parallel programming interfaces: OpenMP, Pthreads, MPI-1 and MPI-2. They were developed in previous studies and classified according to some criteria, such as: number of operations performed in memory, communication rate and number of data exchange operations between threads/processes. In this work we perform a series of tests in these applications, aiming to classify them and show that they have enough characteristics to be used as a benchmark to evaluate the performance and energy consumption of different parallel programming interfaces in multicore architectures. The applications were classified according to: data structures, complexities, TLP, CPU usage and memory usage. The thirteen applications and their implementations with four PPIs cover the classification criteria in such a way that several scenarios can be evaluated, being efficient as benchmark for performance analysis and energy consumption of multicore architectures.

Key-words: Benchmark. Energy consumption. High performance. Parallel programming. Multicore.

Lista de ilustrações

Figura 1 – Comportamento da paralelização em termos de desempenho e energia. 20
Figura 2 – Exemplo de uma arquitetura multinúcleo 25
Figura 3 – Exemplo do modelo *Fork-Join* do OpenMP 28
Figura 4 – Exemplo de atribuição de iterações às *threads* 30
Figura 5 – Exemplo de comunicação entre processos MPI-1 34
Figura 6 – Relacionamento hierárquico resultante da criação de um único processo 35
Figura 7 – Decomposição de domínio 1D 53
Figura 8 – Decomposição de domínio 2D 53
Figura 9 – Principais características da primeira classificação do *benchmark*. . . . 57
Figura 10 – Principais características da segunda classificação do *benchmark*. 58
Figura 11 – Comportamento da comunicação das aplicações. 60
Figura 12 – Principais características da terceira classificação do *benchmark*. 60
Figura 13 – Organização hierárquica de memória do Intel® Core™ i7-4700QM. . . 65
Figura 14 – Organização hierárquica de memória do Intel® Xeon® E5-2609 v3. . . 66
Figura 15 – Gráfico com o TLP das aplicações HC 70
Figura 16 – Gráfico com o TLP das aplicações LC 71
Figura 17 – Gráfico com o TLP das aplicações LC 72
Figura 18 – Gráfico do uso de CPU e memória por aplicação utilizando PThreads. . 73
Figura 19 – Gráfico do uso de CPU e memória por aplicação utilizando PThreads. . 74
Figura 20 – Gráfico do uso de CPU e memória por aplicação utilizando PThreads. . 74
Figura 21 – Gráfico do uso de CPU e memória por aplicação utilizando OpenMP. . 75
Figura 22 – Gráfico do uso de CPU e memória por aplicação utilizando OpenMP. . 76
Figura 23 – Gráfico do uso de CPU e memória por aplicação utilizando OpenMP. . 76
Figura 24 – Gráfico do uso de CPU e memória por aplicação utilizando MPI-1. . . 78
Figura 25 – Gráfico do uso de CPU e memória por aplicação utilizando MPI-1. . . 78
Figura 26 – Gráfico do uso de CPU e memória por aplicação utilizando MPI-1. . . 79
Figura 27 – Gráfico do uso de CPU e memória por aplicação utilizando MPI-2. . . 80
Figura 28 – Gráfico do uso de CPU e memória por aplicação utilizando MPI-2. . . 81
Figura 29 – Gráfico do uso de CPU e memória por aplicação utilizando MPI-2. . . 81
Figura 30 – Uso de CPU por *thread* da aplicação GL em PThreads. 83
Figura 31 – Uso de CPU por *thread* da aplicação MM em PThreads. 83
Figura 32 – Gráfico da média de uso de CPU e memória por IPP 84

Lista de tabelas

Tabela 1 – Pontos de comparação entre os *benchmarks* 39
Tabela 2 – Organização do *benchmark* atual de acordo com a comunicação entre *threads*/processos. 61
Tabela 3 – Tamanho das entradas das aplicações. 68
Tabela 4 – Dados obtidos através da análise das aplicações sequenciais. 70
Tabela 5 – Visão geral do uso de CPU e memória das aplicações com PThreads. . 75
Tabela 6 – Visão geral do uso de CPU e memória das aplicações com OpenMP. . . 77
Tabela 7 – Visão geral do uso de CPU e memória das aplicações com MPI-1. . . . 79
Tabela 8 – Visão geral do uso de CPU e memória das aplicações com MPI-2. . . . 82

Lista de siglas

CPU *Central Processing Unit*

DFT *Discrete Fourier Transform*

EEMBC *Embedded Microprocessor Benchmark Consortium*

FIFO *First In, First Out*

HC *High Communication*

HPL *High-Performance Linpack*

ILP *Instruction-Level Parallelism*

IPP Interface de Programação Paralela

LC *Low Communication*

MPI *Message-Passing Interface*

NAS *Numerical Aerodynamic Simulation*

OpenMP *Open Multi-Processing Application Program Interface*

PAPI *Performance Application Programming Interface*

PThreads *POSIX Threads*

SPEC *Standard Performance Evaluation Corporation*

TLP *Thread-Level Parallelism*

Sumário

1 INTRODUÇÃO 19
1.1 Objetivos 21
1.2 Organização do Texto 21

2 FUNDAMENTAÇÃO TEÓRICA 23
2.1 Benchmarks 24
2.2 Arquiteturas Multinúcleo 24
2.3 Programação Paralela 26
2.4 Interfaces de Programação Paralela 27
2.4.1 OpenMP 28
2.4.2 POSIX Threads 30
2.4.3 *Message-Passing Interface* 32
2.4.3.1 MPI-1 33
2.4.3.2 MPI-2 35
2.5 Estado da Arte 36
2.5.1 Trabalhos Relacionados 36
2.5.2 Contexto deste Trabalho 38
2.5.3 Comparação entre os Benchmarks 39
2.6 Balanço do Capítulo 40

3 *BENCHMARK* ESTUDADO 41
3.1 Apresentação das Aplicações 41
3.1.1 Cálculo do Pi 41
3.1.2 Série Harmônica 42
3.1.3 Dijkstra 43
3.1.4 Método de Jacobi 43
3.1.5 Multiplicação de Matrizes 44
3.1.6 Similaridade entre Histogramas 45
3.1.7 Produto Escalar 46
3.1.8 Jogo da Vida 46
3.1.9 Integração Numérica 47
3.1.10 Gram-Schmidt 48
3.1.11 Ordenação Par-Ímpar 49
3.1.12 Turing Ring 49
3.1.13 Transformada Discreta de Fourier 50
3.2 Paralelização das Aplicações 51

3.2.1 Aplicações que utilizam vetores . . . 52
3.2.2 Aplicações que utilizam matrizes . . . 53
3.2.3 Aplicações que não operam sobre dados estruturados . . . 54
3.3 Balanço do capítulo . . . 54

4 CLASSIFICAÇÃO DO *BENCHMARK* . . . 55
4.1 Histórico de Classificações . . . 55
4.1.1 *CPU-Bound* e *Memory-Bound* . . . 56
4.1.2 *CPU Bound*, *Weakly Memory-Bound* e *Memory-Bound* . . . 58
4.1.3 Alta e Baixa Demanda de Comunicação . . . 59
4.2 Definição de Critérios de Classificação . . . 60
4.2.1 *Thread-Level Parallelism* . . . 62
4.2.2 Demanda de CPU e de Memória . . . 62
4.3 Balanço do Capítulo . . . 63

5 RESULTADOS . . . 65
5.1 Ambiente de Execução . . . 65
5.1.1 Arquiteturas Utilizadas . . . 65
5.1.2 Extração dos Dados . . . 66
5.1.2.1 Ferramentas Utilizadas . . . 66
5.1.2.2 Coleta dos Dados . . . 67
5.1.2.3 Problemas Encontrados . . . 67
5.2 Complexidade das Aplicações . . . 69
5.3 TLP . . . 70
5.4 Uso de CPU e Memória por IPP . . . 72
5.4.1 PThreads . . . 73
5.4.2 OpenMP . . . 75
5.4.3 MPI-1 . . . 77
5.4.4 MPI-2 . . . 80
5.4.5 Comparação entre as IPPs . . . 83

6 CONCLUSÃO . . . 87

REFERÊNCIAS . . . 89

APÊNDICES 95

APÊNDICE A – ESTRATÉGIA DE PARALELIZAÇÃO . . . 97
A.1 Cálculo do Pi . . . 97
A.2 Série Harmônica . . . 97

A.3 Dijkstra . . . 98
A.4 Método de Jacobi . . . 99
A.5 Multiplicação de Matrizes . . . 100
A.6 Similaridade entre Histogramas . . . 100
A.7 Produto Escalar . . . 101
A.8 Jogo da Vida . . . 101
A.9 Integração Numérica . . . 102
A.10 Gram-Schmidt . . . 103
A.11 Ordenação Par-Ímpar . . . 103
A.12 Turing Ring . . . 104
A.13 Transformada Discreta de Fourier . . . 105

Índice . . . 107

1 Introdução

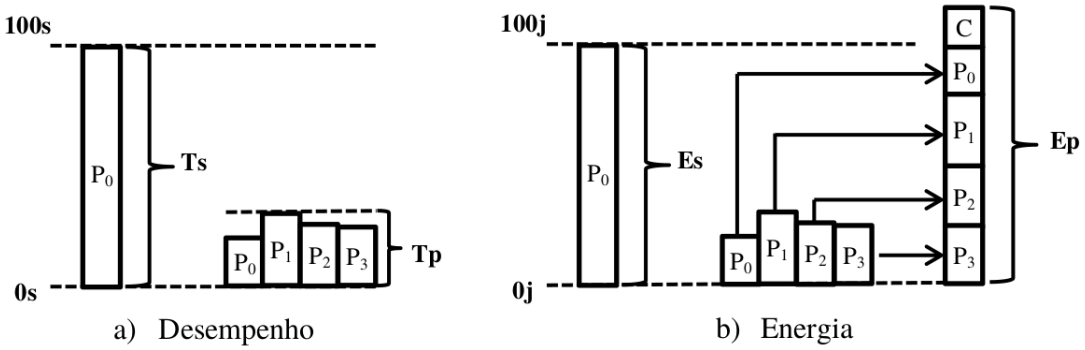
A maioria dos processadores desenvolvidos atualmente possuem vários núcleos de processamento. Esses processadores multinúcleo fazem parte da maioria dos sistemas atuais e podem ser encontrados tanto em supercomputadores, quanto em computadores pessoais e sistemas embarcados. Também, boa parte dos dispositivos móveis que dependem de baterias (*smartphones*, *tablets*, etc.) já possuem processadores multinúcleo. Esse tipo de arquitetura permite a execução de tarefas de forma concorrente, podendo-se extrair o máximo de desempenho possível.

Nos últimos anos, o aumento da complexidade das aplicações para sistemas embarcados tem demandado uma maior eficiência computacional e energética. Aliada a isso, o advento da computação *exascale*, com poder de processamento 100 vezes maior que os processadores *petascale*, lançados em 2008 e usados até hoje (CAPPELLO et al., 2009), trás consigo a necessidade de aumentar o desempenho com o menor impacto possível no consumo de energia. Essa tecnologia possui previsão de lançamento para 2020, sendo que a Intel prometeu lançar no mercado a primeira arquitetura *exascale* já em 2018 (FELDMAN, 2012). Entretanto, os computadores *exascale* poderão chegar ao consumo de energia que corresponde à potência fornecida por uma usina nuclear de médio porte (WEHNER; OLIKER; SHALF, 2009). Com isso, fica claro que o desafio não deve ser somente aumentar o desempenho, mas também consumir a menor quantidade de energia possível.

O número de sistemas embarcados que utilizam processadores multinúcleo cresce constantemente. A computação paralela visa utilizar múltiplos processadores para executar partes diferentes de um mesmo programa simultaneamente. Entretanto, os processadores deverão ser capazes de trocar informações em determinado momento da execução. A maior parte das arquiteturas atuais possuem regiões de memória compartilhada, onde pode ocorrer essa troca de informações simultaneamente. Entretanto, este acesso se dá em regiões da memória que estão hierarquicamente mais distantes do processador (memória *cache* L3 e principal), e que possuem maior consumo de energia e tempo de acesso, quando comparado ao acesso à memória privada de cada processador (memória *cache* L1 e L2) (KORTHIKANTI; AGHA, 2010).

Enquanto a paralelização possibilita aumentar o desempenho dos sistemas computacionais, a necessidade de comunicação entre os processadores pode levar a um gargalo no consumo de energia. Por exemplo, sistema de memória interno do processador pode ser responsável por até 80% do consumo total de energia dos sistemas embarcados (JI; WANG; ZHOU, 2008). Essa situação pode ser exemplificada pela Figura 1, que apresenta

Figura 1 – Comportamento da paralelização em termos de desempenho e energia.



Fonte: Lorenzon (2014).

uma análise do comportamento de desempenho e consumo de energia considerando a execução paralela de uma aplicação em um processador com quatro núcleos (P_0 , P_1 , P_2 e P_3).

Uma aplicação paralelizada geralmente obtém ganho de desempenho com relação à versão sequencial, pois o tempo total de computação é distribuído entre os processadores. A Figura 1a, por exemplo, mostra que a aplicação sequencial levou 100 segundos para executar (Ts) enquanto que a versão paralela levou aproximadamente 30 segundos (Tp), representando ganho de aproximadamente 3,3 vezes no desempenho. No entanto, este mesmo comportamento não acontece no consumo de energia (Figura 1b), onde a energia total consumida pela execução paralela (Ep) corresponde à soma da energia consumida em cada um dos quatro processadores P_0 , P_1 , P_2 e P_3), adicionada à energia consumida para comunicação (C) entre os processadores, que varia de acordo com a aplicação. Por exemplo, aplicações com grande quantidade de comunicação terão, por consequência, consumo de energia maior que aplicações com pouca comunicação.

Assim, o ganho de desempenho e o consumo de energia podem variar de acordo com a arquitetura do processador e organização hierárquica da memória; o modelo de comunicação de cada Interfaces de Programação Paralela (IPPs); e também com as características de cada aplicação, que podem variar se a aplicação faz um uso mais intensivo de *Central Processing Unit* (CPU) ou de memória. Essa variedade de fatores torna muito difícil classificar sistemas de acordo com o seu desempenho ou eficiência energética.

São muitos fatores que precisam ser considerados na hora de analisar um sistema em relação ao seu consumo de energia e desempenho. As diferenças entre arquiteturas de processadores, entre as IPPs utilizadas e as características das aplicações que executarão nesse ambiente podem impactar tanto no desempenho quanto na eficiência energética. Para contornar essa dificuldade imposta por esses diversos fatores, existe a possibilidade

de utilizar *benchmarks* específicos para essa finalidade.

Um *benchmark* tem como objetivo aplicar uma série de testes padronizados para a realização de medições em sistemas computacionais, permitindo que esses resultados possam ser comparados entre diferentes arquiteturas (SPEC, 2016). Isso permite que essas arquiteturas possam ser classificadas de acordo com critérios pré-definidos por esses *benchmarks*. Esses critérios podem variar de consumo de CPU à segurança dos dados, por exemplo.

1.1 Objetivos

O cenário apresentado nesta introdução mostra que, embora a paralelização permita ganhos de desempenho, isso pode levar a um maior consumo energético. Afinal, uma aplicação paralela necessita trocar informações entre os processadores em regiões da memória que estão hierarquicamente mais distantes do processador e que possuem maior consumo de energia. Deve-se considerar, ainda, que a hierarquia de memória de processadores embarcados é diferente quando comparada a de propósito geral, em termos de tamanho, tempo de acesso, consumo de energia e assim por diante. Ademais, o paralelismo pode ser explorado com diferentes IPPs, sendo que cada uma tem suas particularidades em termos de sincronização e comunicação.

Este trabalho reúne um conjunto de 13 aplicações desenvolvidas com o propósito de avaliar o desempenho e o consumo de energia em arquiteturas multinúcleos. Essas aplicações foram desenvolvidas e classificadas de acordo com diferentes critérios em estudos anteriores. O objetivo deste trabalho é realizar uma série de testes nessas aplicações afim de mostrar que elas possuem características suficientes para serem utilizadas como um *benchmark* para avaliar o desempenho e o consumo de energia de diferentes arquiteturas multinúcleo.

Para atingir os objetivos propostos são analisadas as complexidade e estruturas de dados utilizadas, o *Thread-Level Parallelism* (TLP) e o uso de CPU e memória. As aplicações também são analisadas considerando cada IPP individualmente, permitindo observar o impacto que o uso de *threads*/processos e dos diferentes modelos de comunicação das IPPs causam na classificação dessas aplicações.

1.2 Organização do Texto

Este trabalho esta organizado da seguinte forma: no Capítulo 2 são apresentados os fundamentos sobre os quais este trabalho se baseia. Primeiramente é discutida a importância dos *benchmarks* na computação. Na sequência, é apresentada uma visão geral sobre arquiteturas multinúcleo. Esse capítulo também contextualiza a programação

paralela, apresentando as quatro IPPs utilizadas na paralelização das aplicações. Por fim, é feita uma discussão sobre os principais trabalhos relacionados que foram encontrados através de um estudo bibliográfico, justificando assim a utilização do *benchmark* proposto neste trabalho.

O Capítulo 3 apresenta todo o conjunto das aplicações utilizadas neste trabalho, descrevendo aspectos como complexidade, suas principais características e seus algoritmos. Ainda nesse capítulo são discutidas as estratégias utilizadas na paralelização das aplicações utilizando as quatro IPPs diferentes. Mais detalhes da paralelização dessas aplicações são apresentados no Apêndice A.

O histórico de classificações realizado em estudos anteriores é descrito detalhadamente no Capítulo 4. Além das três fases de classificação realizada por esses estudos, também são apresentados nesse Capítulo os novos critérios de classificação.

O Capítulo 5 apresenta os resultados deste trabalho. Iniciando com a apresentação do ambiente de execução, onde também são discutidos os problemas enfrentados ao longo do estudo. Na sequência são apresentados os resultados de acordo com os critérios utilizados. Por fim, nós concluímos o trabalho, fazendo uma revisão dos objetivos e definindo os trabalhos futuros.

2 Fundamentação Teórica

Na computação, sempre existiu a necessidade de comparar, classificar ou testar o desempenho de diferentes sistemas para os mais diversos propósitos. Com a evolução desses sistemas, tornou-se inviável fazer essa tarefa através de testes simples e comparação das especificações de cada arquitetura (GRAY, 1992). Nesse momento, viu-se necessário criar uma forma de padronizar um método que suprisse essa necessidade crescente. Com isso surgiram os *benchmarks*, que hoje são utilizados em diversas áreas da computação.

Uma das áreas que necessita estar constantemente testando a eficiência e comparando o desempenho de sistemas é a área de Processamento de Alto Desempenho. Nessa área visa-se aumentar o desempenho de uma determinada aplicação, geralmente através do uso de programação paralela. De acordo com Rauber e Rünger (2010), a paralelização consiste em dividir as tarefas de uma aplicação e executá-las concorrentemente com a finalidade de reduzir seu tempo total de execução. Sua utilização torna-se essencial em aplicações científicas que necessitam de grande poder computacional, como cálculos da previsão do tempo, cálculos de sequências de DNA e de genoma, entre outras diferentes aplicações. Mais atualmente, com a popularização das arquiteturas multinúcleo, as aplicações de uso geral (filtros de imagens e som, editores gráficos, servidores de internet, etc) também têm tirado proveito da programação paralela.

Paralelizar uma aplicação pode não ser uma tarefa fácil. Para isso, IPPs são capazes de tornar esse processo menos árduo para o programador. Porém, cada interface possui suas próprias características e especificações que devem ser cuidadosamente estudadas para implementar uma aplicação paralela. O programador deve seguir alguns cuidados básicos, ficando atento à que partes do código serão executadas simultaneamente e, dependendo da interface, deve inclusive definir como será feita a comunicação e a sincronização dos dados.

Neste capítulo contextualizamos os *benchmarks* na seção 2.1. Na sequência mostramos o funcionamento das arquiteturas multinúcleo e sua estrutura de memória (2.2). Uma visão geral sobre paralelização de algoritmos é apresentada na seção 2.3. As IPPs utilizadas neste trabalho estão definidas na seção 2.4, onde são detalhadas as características individuais de cada IPP. Por fim, na seção 2.6 revisamos os principais aspectos abordados neste Capítulo.

2.1 Benchmarks

Com a evolução das arquiteturas computacionais, a comparação do desempenho de diferentes sistemas de computação, somente olhando suas especificações, se tornou uma tarefa difícil. Historicamente, os fabricantes costumavam classificar os desempenhos de seus sistemas utilizando uma variedade de métricas distintas. Isso gerava muita confusão entre os consumidores e muitas vezes essas informações podiam não ser credíveis. Para resolver esse problema, um grupo de fabricantes se reuniu com o objetivo de desenvolver séries de testes padronizados para a realização de medições nesses sistemas, permitindo que esses resultados pudessem ser comparados entre diferentes arquiteturas (SPEC, 2016). Esse processo também serviria para educar os consumidores sobre o desempenho de seus produtos. Foi nesse contexto que surgiram os *benchmarks*.

Na computação, *benchmark* é a ação de comparar o desempenho relativo de um objeto ou produto por meio da execução de um programa de computador (GRAY, 1992). Para extrair dados corretos sobre os diferentes produtos e objetos, com a finalidade de compará-los de maneira equivalente, uma série de testes padrões e ensaios são realizados. Ainda assim, o termo *benchmark* é comumente utilizado para definir os próprios programas desenvolvidos para executar o processo. Já o termo *benchmarking* é associado ao processo de avaliação das características e do desempenho de um *hardware* de computador, como por exemplo, o desempenho da operação de ponto flutuante de uma CPU.

Neste trabalho, estudaremos um *benchmark* para medir o desempenho e o consumo energético de diferentes IPPs em arquiteturas multinúcleo. Essas arquiteturas podem incluir tanto processadores de propósito geral quanto processadores embarcados. O *benchmark* atualmente consiste de 13 aplicações paralelas implementadas utilizando quatro IPPs diferentes: *POSIX Threads* (PThreads), *Open Multi-Processing Application Program Interface* (OpenMP), *Message-Passing Interface* (MPI)-1 e MPI-2.

2.2 Arquiteturas Multinúcleo

Uma arquitetura multinúcleo consiste de um sistema computacional que possui um processador com dois ou mais núcleos de processamento no interior do processador. Estes núcleos são responsáveis por dividir as tarefas entre si, ou seja, permitem trabalhar em um ambiente multitarefa. Em processadores de um só núcleo, as funções de multitarefa podem ultrapassar a capacidade da CPU, o que resulta em queda no desempenho enquanto as operações aguardam para serem processadas (ASANOVIC et al., 2006). Em processadores de múltiplos núcleos o sistema operacional trata cada um desses núcleos como um processador diferente. Na maioria dos casos, cada unidade possui sua própria memória *cache* e pode processar várias instruções simultaneamente. Adicionar novos núcleos de processamento a um processador possibilita que as instruções das aplicações

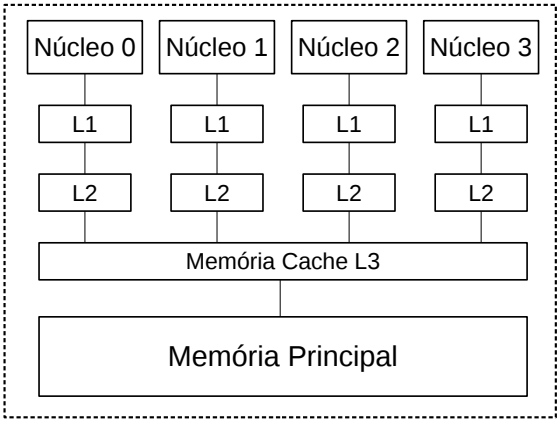
sejam executadas em paralelo, como se fossem dois ou mais processadores distintos.

Arquiteturas multinúcleo caracterizam-se por possuírem múltiplas unidades de processamento, compartilhando acesso a um mesmo espaço de endereçamento (HENNESSY; PATTERSON, 2011). Essas arquiteturas são capazes de realizar a execução concorrente de diferentes fluxos de instruções sobre unidades de processamento independentes. O espaço de endereçamento serve como meio para a comunicação (troca de dados) entre os processadores, através de instruções do tipo *load* e *store* em regiões compartilhadas da memória.

Existem, porém, muitos desafios enfrentados pelas arquiteturas multinúcleo. O acesso aos dados é um dos mais relevantes. Normalmente, processadores com um único núcleo já incluíam níveis de memória *cache intra-chip* para diminuir a latência média durante o acesso à memória (KORTHIKANTI; AGHA, 2010). No entanto, quando estamos falando de arquiteturas multinúcleo, o problema se agrava, afinal é necessária a comunicação entre estes núcleos. Isso pode se tornar um grande problema nas arquiteturas atuais, que facilmente possuem 8, 16 ou ainda mais núcleos necessitando comunicar-se entre si. Dessa forma, faz-se necessário adaptar a organização da arquitetura incluindo níveis de memória compartilhada.

Na Figura 2 podemos ver a estrutura de uma arquitetura multinúcleo. Neste exemplo, existem quatro núcleos, cada um com memórias *cache* L1 e L2 privadas, o que significa que somente o núcleo em questão pode acessá-las. Portanto, não existe comunicação entre os núcleos nesses dois níveis de memória. A comunicação neste caso só pode ser realizada através dos níveis inferiores na hierarquia de memória, os quais possuem um custo (tempo e potência) mais elevados para cada acesso. Esses níveis inferiores são representados pela memória *cache* L3 e a memória principal.

Figura 2 – Exemplo de uma arquitetura multinúcleo



Fonte: o próprio autor.

A Figura 2 também permite observarmos que o sistema de memórias *cache* ocupam uma área significativa dentro de uma arquitetura multinúcleo. Afinal, as *caches* possuem um papel fundamental na comunicação entre os núcleos. A vazão dos dados é definida de acordo com a capacidade e velocidade dessas memórias. Portanto, otimizar um algoritmo para reduzir a quantidade de comunicação necessária torna possível melhorar a utilização das memórias *cache*. Isso possibilita maiores ganhos de desempenho e uma maior economia de energia (KORTHIKANTI; AGHA, 2010).

É notável então que um grande gargalo na comunicação entre os núcleos se encontra na região de memória compartilhada. Essa situação tende a piorar de acordo com a dependência de dados entre as *threads*. Se uma determinada *thread* que está executando em um núcleo depende de um dado que está sendo utilizado por uma *thread* em outro núcleo, a primeira *thread* ficará bloqueada até que esse dado seja liberado. Somente então essa *thread* poderá continuar ou finalizar sua execução. Quanto maior for a necessidade de comunicação, maior será o tempo e energia consumidos por uma aplicação apenas com comunicação.

2.3 Programação Paralela

A programação paralela consiste na divisão de tarefas de uma aplicação com a finalidade de reduzir seu tempo total de execução (RAUBER; RÜNGER, 2010). Opera sob o princípio de que grande problemas geralmente podem ser divididos em problemas menores, que então são resolvidos concorrentemente (em paralelo) (ALMASI, 1990). Existem diferentes formas de computação paralela: em bit, instrução, de dado ou de tarefa. A técnica de exploração do paralelismo já é empregada por vários anos, principalmente na computação de alto desempenho, mas recentemente o interesse no tema cresceu devido à grande difusão das arquiteturas paralelas. Adicionalmente, além da busca por desempenho, também tem sido tema de estudos a preocupação com o consumo de energia dessas arquiteturas.

A computação paralela se tornou o paradigma dominante nas arquiteturas de computadores sob forma de processadores multinúcleo (ASANOVIC et al., 2006). Ela tem sido muito utilizada no desenvolvimento de aplicações científicas que necessitam de grande poder computacional, como cálculos da previsão do tempo, cálculos de sequências de DNA e de genoma, entre outras diferentes aplicações.

Programas paralelos são mais difíceis de programar em relação aos sequenciais, pois a concorrência introduz diversas novas classes de defeitos potenciais, como os causados pela dependência de dados, por exemplo. A comunicação e a sincronização entre diferentes subtarefas é tipicamente uma das maiores barreiras para atingir grande desempenho em programas paralelos (ALMASI, 1990).

A dependência de dados é um fator fundamental na implementação de algoritmos paralelos. Nenhum programa pode executar mais rápido que a maior cadeia de cálculos dependentes (conhecido como caminho crítico), já que o cálculo depende do cálculo anterior da cadeia, sendo executado sequencialmente (FOSTER, 1995). Entretanto, a maioria dos algoritmos não consiste de somente uma longa cadeia de cálculos dependentes, geralmente há oportunidades para executar cálculos independentes em paralelo.

Os programas de computadores são geralmente desenvolvidos para executarem de forma sequencial, executando uma instrução de cada vez. No entanto, existe um grande potencial na paralelização dessas instruções e que não são exploradas. Arquiteturas superescalares podem executar simultaneamente instruções independentes dentro de um programa, desde que haja unidades funcionais suficientes para tal. Essas arquiteturas, então, são capazes de extrair o paralelismo em uma granularidade mais fina no nível de instruções (*Instruction-Level Parallelism* (ILP) – Paralelismo no Nível de Instrução). Isso também pode ser aplicado para grupos de instruções, que podem ser executados de forma concorrente, ou seja, ao mesmo tempo. Neste caso, a granularidade é mais grossa, e o paralelismo a ser explorado é o do nível de *threads* (TLP – Paralelismo no Nível de *Threads*) (RAUBER; RÜNGER, 2010).

2.4 Interfaces de Programação Paralela

Existem diversos modelos computacionais utilizados em computação paralela, tais como: paralelismo de dados, memória compartilhada, troca de mensagens, operações em memória remota, entre outros. Tais modelos se diferenciam em vários aspectos, como por exemplo, se a memória disponível é localmente compartilhada ou geograficamente distribuída e, volume de comunicação tanto em hardware como em software (GROPP; LUSK; THAKUR, 1999).

Escolher entre esses modelos depende fortemente do problema a ser resolvido e a arquitetura de computação paralela alvo. Esta situação fez com que vários fabricantes de sistemas de computação paralela desenvolvessem suas próprias bibliotecas de interface, focadas apenas em suas características, muitas vezes, inexistentes em outras arquiteturas. Desta forma, o código desenvolvido com bases nestas bibliotecas proprietárias não era portável. Portanto, impulsionada pelos avanços na tecnologia de sistemas paralelos de computação, surgiu a necessidade de criar interfaces de programação que fossem eficientes, funcionais e portáveis. Interfaces capazes de executar nas mais variadas arquiteturas disponíveis no mercado. Foi neste contexto que as IPPs OpenMP, PThreads e MPI foram concebidas.

2.4.1 OpenMP

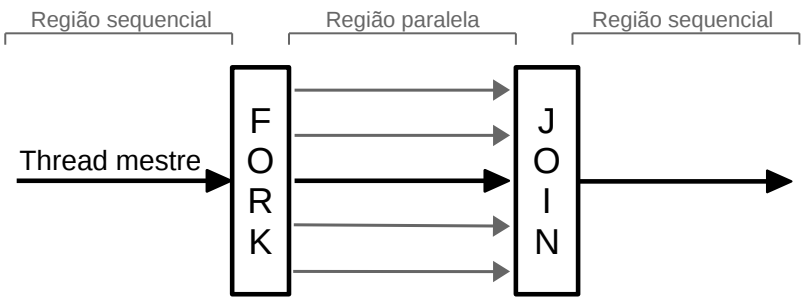
O OpenMP foi criado para direcionar, explicitamente, o paralelismo em ambientes computacionais paralelos com memória compartilhada. Ele foi definido através de um esforço conjunto de fabricantes de hardware e desenvolvedores de software. O principal objetivo na definição do OpenMP foi a especificação de um modelo portável que fornecesse aos programadores uma interface simples e flexível para a programação paralela em sistemas de memória compartilhada.

O padrão OpenMP consiste em uma série de diretivas de compilação, biblioteca de funções e um conjunto de variáveis de ambiente que influenciam a execução de programas paralelos (RAUBER; RÜNGER, 2010). Essas diretivas são inseridas no código sequencial e o código paralelo é gerado pelo compilador a partir delas. Essa interface opera sobre a base do modelo de execução *fork-join*. Tal interface recebe versões em C/C++ e Fortran em todas as arquiteturas disponíveis no mercado, incluindo os sistemas operacionais baseados em Unix e Windows.

O construtor paralelo é a diretiva mais importante do OpenMP, seu trabalho é informar ao compilador a região do código que será executada em paralelo. Este construtor segue o modelo *Fork-Join* (OpenMP ARB, 2015), apresentado na Figura 3. Nesse modelo, temos primeiro um fluxo de execução principal, chamado de *thread* mestre. Quando esta *thread* encontrar o construtor paralelo, ela cria um novo grupo de *threads* escravas (*Fork*). Ao final da região paralela há uma barreira implícita que faz com que as *threads* aguardem até que todas as demais *threads* cheguem naquele ponto (*Join*). A partir deste ponto, somente a *thread* mestre continua a execução do código (TERUEL et al., 2009). Esse construtor é definido pela diretiva `#pragma omp parallel` no OpenMP e inicia uma região paralela.

Existem também construtores de compartilhamento de trabalho que são responsáveis pela divisão de trabalho entre as *threads*. Estes construtores devem estar localizados

Figura 3 – Exemplo do modelo *Fork-Join* do OpenMP



Baseado em OpenMP ARB (2015).

dentro da região paralela. Existe uma barreira implícita no final do construtor, isto é, todas as *threads* esperam até que a última *thread* finalize sua execução. Eles são divididos em: laços paralelos, seções paralelas e construtor único de trabalho (REVISTABW, 2015).

- **Laços paralelos**: O propósito deste construtor é distribuir as iterações do laço de repetição (`for`) entre as *threads*. Um subconjunto contíguo de iterações (*chunk*) é quem determina a granularidade da distribuição da carga de trabalho. Os *chunks* podem ser escalonados entre as *threads* através de diferentes escalonadores:

- **Dinâmico** (*dynamic*): as iterações são atribuídas para as *threads* conforme solicitação. Cada *thread* executa um bloco de iterações e após sua execução solicita outro. Dessa forma o acesso não é necessariamente sequencial;

- **Estático** (*static*): os *chunks* são atribuídos para as *threads* estaticamente através da política *round-robin*, ordenados pelo número identificador da *thread*;

- **Guiado** (*guided*): é similar ao escalonamento dinâmico, no entanto, o tamanho do *chunk* decrementa a cada iteração;

- ***Runtime***: a decisão do tipo de escalonamento é realizado em tempo de execução. Ao término da computação, as *threads* aguardam as demais em uma barreira implícita.

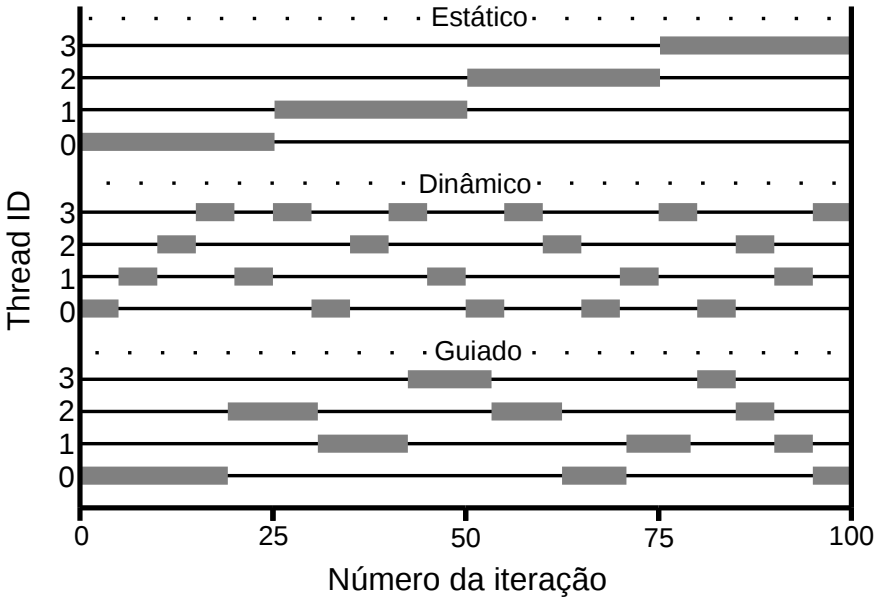
Esses modelos de distribuição são definidos no OpenMP pela cláusula `schedule()`. Essa cláusula descreve como as iterações do laço de repetição serão distribuídas. Ela é utilizada junto à diretiva `#pragma omp for` e faz com que as iterações de um laço `for` dentro da região paralela seja dividido entre as *threads*.

O comportamento desses escalonadores pode ser visto na Figura 4, onde temos um laço com 100 iterações (eixo X) que são divididas entre 4 *threads* (eixo Y). Nesse exemplo, o escalonamento estático distribui *chunks* de 25 iterações uniformemente, ordenados pela identificador de cada *thread*. Para o escalonamento dinâmico temos *chunks* de 5 iterações e a cada solicitação as *threads* receberão outros *chunks* do mesmo tamanho. Por fim, o escalonamento guiado também distribuirá *chunks* de 5 iterações, porém as primeiras solicitações recebem porções maiores.

- **Seções Paralelas**: Este construtor permite que sejam atribuídas entre as *threads* diferentes porções de trabalho. Dessa forma, é possível especificar diferentes regiões do código para serem executadas em paralelo por diferentes *threads*, assim, cada *thread* torna-se responsável pela execução de cada bloco.

- **Construtor Único**: Neste construtor, um bloco de execução de código é atribuído para ser executado por uma única *thread*. Enquanto a *thread* escolhida está exe-

Figura 4 – Exemplo de atribuição de iterações às *threads*



Fonte: Lorenzon (2014).

cutando este bloco de trabalho, as demais aguardam em uma barreira implícita, no final desta região.

O OpenMP possui também diretivas de sincronização que servem para evitar condições de corrida, que acontece quando duas ou mais *threads* tentam acessar o mesmo dado na memória. Existem três dessas diretivas que são comumente utilizadas, que são: *critical*, que serve para restringir a execução de uma determinada tarefa a apenas uma *thread* por vez; *atomic*, onde uma única *thread* atualiza uma região de memória atomicamente; e *barrier*, que possui o propósito de sincronizar todas as *threads* em um dado momento da execução. Todas essas diretivas atuam de forma explícita.

Dessa forma, as *threads* entrarão em estado de *busy-waiting* toda vez que houver um ponto de sincronização, tanto explícito quanto implícito. Quando isso ocorre, as *threads* precisam fazer acessos à memória repetidamente para verificar o estado da variável de controle até o final da sincronização.

2.4.2 POSIX Threads

PThreads é um exemplo de um modelo de execução que existe independentemente de uma linguagem de programação, mesmo sendo baseada na linguagem C (RAUBER; RÜNGER, 2010). Esse modelo permite que um programa controle vários fluxos de trabalho paralelamente, onde cada fluxo de trabalho é referido como uma *thread*. A criação e o controle sobre esses fluxos é feita através da interface *POSIX Threads*.

Diferente do OpenMP, onde o paralelismo é expresso em alto nível de abstração com a inserção de diretivas no código sequencial, em PThreads o paralelismo é explícito através de funções da biblioteca. Ou seja, o programador é responsável por realizar o gerenciamento das *threads* (criação/finalização), a distribuição da carga de trabalho e o controle de execução (BUTENHOF, 1997). PThreads compreende algumas sub-rotinas que podem ser classificadas em quatro grupos principais: gerenciamento de *threads*, *mutexes*, variáveis de condição e de sincronização.

O gerenciamento de *threads* engloba as principais rotinas que são responsáveis pela criação e finalização das *threads*. De forma similar ao OpenMP, inicialmente existe um fluxo de execução que executa a parte sequencial do código. Quando for necessário, novas *threads* são criadas através da função `pthread_create()`. A finalização de uma *thread* pode ocorrer explicitamente através da função `pthread_exit()`. A sincronização entre as *threads* é realizada pela chamada à função `pthread_join()`, que bloqueia as *threads* até a finalização das demais.

Na criação da *thread*, é passada uma função que especifica o que cada thread deverá computar como parâmetro na função `pthread_create()`. O programador definirá essa função de acordo com o balanceamento de carga que for aplicado. Adicionalmente, cada *thread* pode ter uma função diferente.

Para implementar seções críticas e operações atômicas são utilizadas operações de *mutex* (GRAMA, 2003). Um *mutex* é uma variável de controle que pode ter dois estados: livre ou ocupado. Essas operações são definidas através de funções do tipo `pthread_mutex()`. A chamada da função `pthread_mutex_lock()` define o início de uma seção crítica. Se uma *thread* acessar a região crítica esta função bloqueia as demais, não permitindo que acessem essa região. Somente quando a *thread* que está acessando a região crítica executar a função `pthread_mutex_unlock()`, que define o fim da seção crítica, o acesso poderá ser liberado para as demais threads.

Tradicionalmente são utilizadas essas funções do tipo *mutex* para habilitar acesso exclusivo de uma *thread* a uma região compartilhada da memória. No entanto, algumas situações necessitam uma abordagem mais dinâmica para gerenciar acesso exclusivo a estes recursos. Dessa forma, variáveis de condição permitem uma *thread* aguardar até que certa condição seja satisfeita para ter acesso à região exclusiva. Em PThreads, tais variáveis são implementadas através da chamada de funções do tipo `pthread_cond()`.

Após uma *thread* terminar sua execução de uma região crítica, ela deve seguir seu fluxo de execução. Entretanto, dependendo da aplicação, a próxima etapa só poderá começar após a finalização da etapa atual. Assim, é necessário a utilização de barreiras entre estas etapas, que permitam que as *threads* só continuem sua execução quando todas as demais estiverem no mesmo ponto de execução. Este comportamento é fornecido através de funções de sincronização do tipo `pthread_barrier()`.

Nesta IPP, quando vários processos estiverem bloqueados no *mutex*, será feita uma escolha aleatória permitindo que um deles entre na região crítica. Neste tipo de sincronização, as *threads* que estão aguardando perdem prioridade e ficam em estado de espera, ganhando prioridade novamente quando forem reescalonadas. Embora esta técnica não impacte no número de acessos à memória, a troca de contexto pode levar a uma redução no desempenho se a sincronização for constantemente exigida (TANENBAUM et al., 1987).

2.4.3 *Message-Passing Interface*

O padrão MPI é uma biblioteca padrão para comunicação em memória distribuída. Foi inicialmente definida através da participação da comunidade de fabricantes e pesquisadores da área de Processamento de Alto Desempenho. Ele consiste numa interface de troca de mensagens e provê funções para linguagem C, C++ e sub-rotinas para Fortran-77 e Fortran-95. No padrão MPI, uma aplicação é constituída por um ou mais processos que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre os processos. A concepção do MPI envolveu um processo de padronização englobando um grupo de 60 pessoas de 40 organizações, principalmente dos Estados Unidos e da Europa (GROPP; LUSK; THAKUR, 1999). Neste trabalho, serão abordadas as normas MPI-1 e MPI-2.

Elementos importantes em implementações paralelas são a comunicação de dados entre processos paralelos e o balanceamento da carga. Quanto à comunicação, o MPI é capaz de suportar comunicação assíncrona e programação modular, através de mecanismos de comunicadores que permitem ao usuário MPI definir módulos que encapsulem estruturas de comunicação interna. Isso permite que os processos usem mecanismos de comunicação ponto a ponto (operações para enviar mensagens de um determinado processo a outro). Dessa forma, um grupo de processos pode invocar operações coletivas de comunicação para executar operações globais.

Nos sistemas multinúcleo, as operações de comunicação através de troca de mensagens são abstraídas para filas de mensagens, que são objetos similares a *pipes* e filas do tipo *First In, First Out* (FIFO). Se tomarmos como exemplo, uma comunicação ponto a ponto entre os processos P_0 e P_1. Quando o processo P_0 realiza a operação *send*, a mensagem é incluída em uma fila. O processo P_1, que está executando uma operação *receive*, extrai da fila de mensagens a primeira mensagem que satisfaça as características da mensagem que está aguardando (POLETTI et al., 2007). Estas operações utilizando filas de mensagens são realizadas em regiões compartilhadas da memória, do mesmo modo em que é feita a comunicação através de variáveis compartilhadas.

2.4.3.1 MPI-1

MPI-1 especifica operações de comunicações ponto a ponto e coletivas, dentre outras características. Em um programa desenvolvido utilizando MPI-1 todos os processos são criados estaticamente no início da execução, portanto a quantidade de processos permanece inalterada durante a execução do programa. Ao iniciar o programa, uma função de inicialização do ambiente de execução MPI é executada por cada processo. Essa função é o `MPI_Init()`. Enquanto que um processo MPI é finalizado através da chamada à função `MPI_Finalize()`.

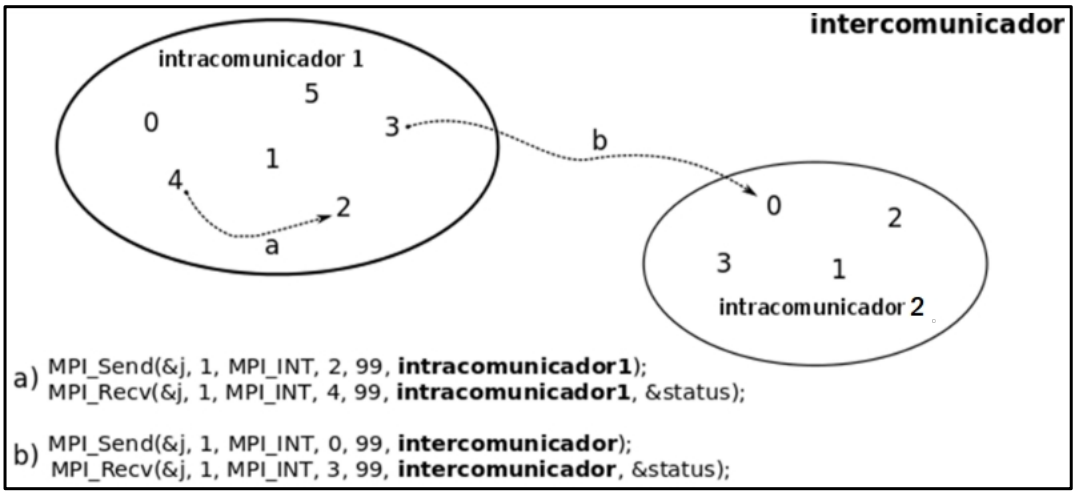
Quando o ambiente de execução é inicializado, identificadores (*rank*) são atribuídos a cada processo MPI dentro do comunicador através da função `MPI_Comm_rank()`. Um comunicador identifica um grupo de processos (conjunto ordenado de n processos) e representa os canais de comunicação por onde os dados são transmitidos entre os processos MPI. Existe também, por padrão, um comunicador pré-definido, chamado `MPI_COMM_WORLD`. Por fim, a função `MPI_Comm_size()` é utilizada para obter o número total de processos de um determinado comunicador.

É possível dividir os comunicadores de duas formas: intracomunicadores, quando a comunicação é interna a um grupo de processos de um mesmo comunicador; e intercomunicadores, quando a processos de intracomunicadores diferentes se comunicam. A Figura 5 ilustra a comunicação entre os processos utilizando intracomunicadores (comunicação representada em (a)) e intercomunicadores (comunicações representada em (b)). Conforme também pode ser observado na Figura 5, a comunicação entre os processos ocorre através do uso de primitivas de envio/recebimento. Nela, são apresentados dois cenários (a e b) de comunicação ponto a ponto, utilizando as primitivas `MPI_Send()` para envio e `MPI_Recv()` para recebimento.

O primeiro caso (a) refere-se à comunicação ponto a ponto dentro do `intracomunicador1`. Nele, o processo 4 envia um dado para o processo 2. Portanto, o processo 4 executará a função `MPI_Send()`, e o processo 2 a função `MPI_Recv()`. Já o segundo cenário (b), corresponde à comunicação através do intercomunicador entre dois intracomunicadores diferentes (`intracomunicador1` e `intracomunicador2`). Nele, o processo 3 do `intracomunicador1` envia um dado através da função `MPI_Send()` para o processo 0 do `intracomunicador2`, que o recebe através da função `MPI_Recv()`.

Por realizarem operações síncronas, estas funções de comunicação são utilizadas como pontos de sincronismo entre os processos envolvidos. Isso significa que enquanto todos os dados não forem enviados ou copiados para o *buffer* de memória, a função irá ficar bloqueada. Ao término dessa operação, cada processo segue o seu fluxo de execução. No entanto, manter os processos bloqueados até a comunicação estar concluída pode se tornar o gargalo da aplicação. Esta limitação pode ser contornada através do uso de

Figura 5 – Exemplo de comunicação entre processos MPI-1



Exemplo de comunicação entre: a) processos internos a um intracomunicador (p_4 envia para p_2) e b) utilizando intercomunicador em intracomunicadores diferentes (p_3 do intracomunicador1 envia para p_0 do intracomunicador2)

Fonte: Maillard e Cera (2010)

operações assíncronas.

Para realizar uma operação de envio assíncrona é utilizada a função `MPI_Isend()`. Essa função retorna após a mensagem ser copiada para o *buffer* de envio. Deste modo, o processo que a enviou pode seguir com a sua execução enquanto a mensagem está sendo transmitida. De forma similar, a operação de recebimento não bloqueante (`MPI_Irecv()`) inicia a operação, mas não a completa. A função somente completará quando a mensagem estiver armazenada no *buffer* de recebimento. Isso permite que o processo possa computar sobre outros dados enquanto espera. Porém, deve ser realizada uma verificação em algum momento da execução, a fim de saber se a operação está completa ou não. Dessa forma, as funções `MPI_Wait()` e `MPI_Waitany()` podem ser utilizadas neste contexto, indicando que somente quando o processo concluir as comunicações iniciadas é que ele seguirá seu fluxo de execução. Também é possível utilizar as operações bloqueantes e não-bloqueantes em conjunto (por exemplo: `MPI_Send()` e `MPI_Irecv()`). Em termos de comunicação, o uso adequado destas funções pode influenciar na eficiência da aplicação.

As operações coletivas em MPI podem ser divididas em: operações de sincronização, movimento de dados e computação coletiva. As operações de sincronização são aquelas que só podem continuar o fluxo de execução após todos os demais processos do grupo atingirem o ponto de sincronização. As operações de movimento de dados, por outro lado, servem para realizar a troca de informações entre vários processos. As

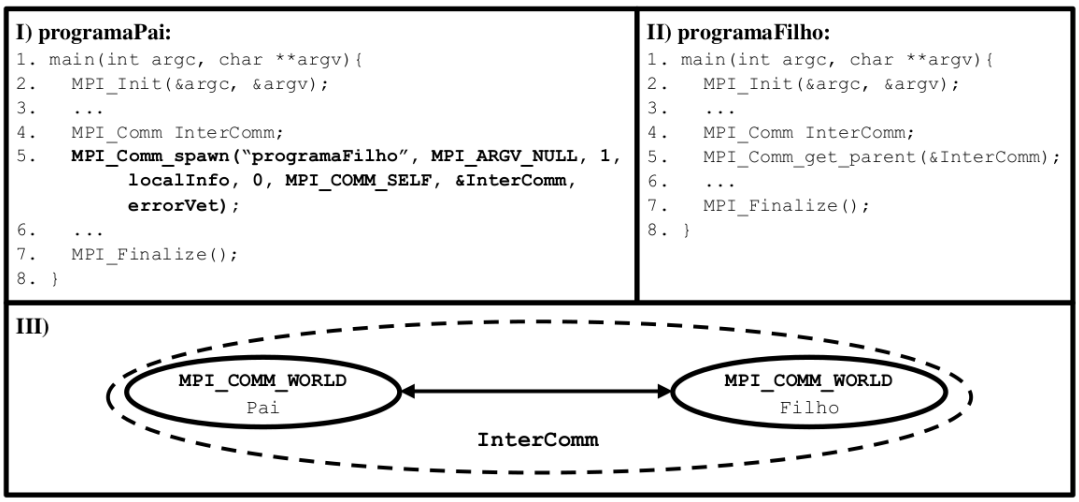
principais são *broadcast* (`MPI_Bcast()`), *scatter/gatter* (`MPI_Scatter()/MPI_Gatter()`) e todos para todos (`MPI_Alltoall()`). Por fim, as operações de computação coletiva, também chamadas de operações de reduções. Essas operações realizam operações sobre os dados provenientes de diversos processos. As operações mais comuns são: soma, multiplicação, máximo e mínimo valor, entre outras.

2.4.3.2 MPI-2

O padrão MPI-2 adiciona novos conjuntos de tópicos ao MPI. Essa IPP se diferencia do MPI-1 nas seguintes características: criação dinâmica de processos, operações de entrada e saída paralela, comunicações assimétricas e coletivas estendidas (GROPP; LUSK; THAKUR, 1999).

Tradicionalmente, aplicações implementadas com MPI-2 iniciam a execução com um único processo. Para a criação de processos dinamicamente é utilizada a primitiva `MPI_Comm_spawn()`. Um processo da aplicação MPI, o qual será chamado de pai, invoca essa primitiva. Sua invocação faz com que seja criado um novo processo, chamado filho, o qual não precisa ser idêntico ao pai. Após a criação de um processo filho, ele irá pertencer a um intracomunicador diferente do pai e a comunicação entre eles ocorrerá através de um intercomunicador. No processo filho, a execução da função `MPI_Comm_get_parent()` é responsável por retornar o intercomunicador que o liga com

Figura 6 – Relacionamento hierárquico resultante da criação de um único processo



I) Pseudocódigo do programa executado pelo processo pai; **II**) Pseudocódigo do programa executado pelo processo filho; e **III**) Ilustração do relacionamento resultante entre pai e filho após a criação do processo filho.

Fonte: Lorenzon, Cera e Rossi (2012)

o pai. No processo pai, o intercomunicador que o liga ao filho é retornado na própria execução da função `MPI_Comm_spawn()`.

A criação de um processo utilizando esta IPP é ilustrado na Figura 6. A primeira área (**I**) nessa figura apresenta o pseudocódigo denominado `programaPai` que será executado pelo processo pai. Na linha 2 ocorre a inicialização do ambiente MPI e na linha 7 sua finalização. A criação do processo filho ocorre na linha 5. Esse processo irá executar o programa chamado de `programaFilho`, que é definido pelo primeiro parâmetro. O sétimo parâmetro indica o intercomunicador que será utilizado para comunicações entre o pai e o(s) filho(s) (`InterComm`). A inicialização das variáveis, bem como outras definições não relevantes foram omitidas. Ainda nessa Figura 6, na área **II**, é apresentado o pseudocódigo do processo filho.

Quando o ambiente MPI for inicializado no processo filho, será executada a função `MPI_Comm_get_parent()` (linha 5). O intercomunicador `InterComm` será retornado por essa função e ligará o seu intracomunicador com o do pai. A área **III** ilustra o relacionamento hierárquico resultante após a criação do processo filho, sendo que cada processo está interno ao seu intracomunicador (elipse) e interligado através do intercomunicador, representado pela elipse pontilhada.

O MPI-2 permite ainda comunicações coletivas através de intercomunicadores, além das possíveis comunicações ponto a ponto (utilizando inter e intracomunicadores) e coletivas (utilizando intracomunicadores) descritas em MPI-1. Portanto, um processo que pertence a um intracomunicador pode comunicar-se com n processos filhos que estão em outro intracomunicador, através do intercomunicador que os liga.

2.5 Estado da Arte

Existem diversos *benchmarks* desenvolvidos para atender diferentes propósitos. Através de um estudo bibliográfico, foram procurados *benchmarks* que possuem propósitos semelhantes e as mesmas arquiteturas alvo do *benchmark* estudado neste trabalho. Portanto, foram considerados *benchmarks* que fornecem um conjunto de aplicações paralelas para arquiteturas multinúcleo embarcadas ou de propósito geral. Dessa forma, identificamos os seguintes *benchmarks* relacionados: EEMBC, Mibench, ParMiBench, SPEC, Linpack, NAS e Adept Project.

2.5.1 Trabalhos Relacionados

O *Embedded Microprocessor Benchmark Consortium* (EEMBC) é um *benchmark* para sistemas embarcados de código fechado (oferece licenças acadêmicas). Ele fornece um conjunto de 23 aplicações implementadas em C e organizadas em várias categorias, de forma a atender uma grande parcela do mercado (POOVEY et al., 2009). Existem

também outras subdivisões desse *benchmark*, por exemplo, o IoT *Benchmark* e o ULP-Bench, que focam em medir a eficiência energética dos processadores embarcados. Para medir desempenho em arquiteturas multinúcleo existe o MultiBench, que se destaca por usar aplicações que aceitam uma grande diversidade de cargas de trabalho. Por fim, o AutoBench 2.0 é voltado para sistemas automotivos e avalia o desempenho de sistemas embarcados com processadores multinúcleo.

MiBench é um *benchmark* para medir o desempenho de sistemas embarcados com um único processador e que trabalha com grandes e pequenos conjuntos de dados. A proposta é que os conjuntos de dados menores atendam à pequenas aplicações embarcadas, enquanto o grande conjunto de dados fornece uma aplicação mais estressante para sistemas embarcados de grande porte (GUTHAUS et al., 2001). É uma ferramenta baseada no *benchmark* EEMBC, porém, ao contrário do mesmo, possui código aberto. Esse conjunto consiste de 35 aplicações implementadas em C que estão divididas em 6 domínios diferentes, de forma semelhante ao EEMBC. Esses domínios são: controle industrial e sistemas automotivos, dispositivos pessoais, dispostivos de escritórios, redes, segurança e telecomunicações.

O ParMiBench é um *benchmark* de código aberto que serve, especificamente, para medir o desempenho em sistemas embarcados que possuem mais de um processador (IQBAL; LIANG; GRAHN, 2010). A estrutura desse *benchmark* é baseada na mesma estrutura adotada pelo MiBench e pelo EEMBC, que organizam suas aplicações em categorias e domínios de aplicações. O ParMiBench adota 4 dessas categorias, entre elas: controle industrial e sistemas automotivos, redes, dispostivos de escritório e segurança. Seu conjunto consiste de 7 aplicações paralelas implementadas utilizando PThreads.

O *Standard Performance Evaluation Corporation* (SPEC) é um *benchmark* de código fechado, porém oferece licenças acadêmicas de mais fácil acesso e mais abrangentes que o EEMBC. Esse *benchmark* é voltado para arquiteturas de propósito geral, porém está subdividido em diversos grupos com arquiteturas alvo específicas, podendo ser usado para diversos fins, como: servidores Java, sistemas de arquivos, sistemas de alto desempenho, teste de CPU, entre outros. Neste trabalho consideraremos os seguintes grupos: SPEC MPI2007, SPEC OMP2012 e SPEC Power. O SPEC MPI2007 é um conjunto de 18 aplicações implementadas em MPI focado em testar computadores de alto desempenho. Já o SPEC OMP2012 utiliza 14 aplicações científicas implementadas em OpenMP, oferecendo métricas opcionais de consumo de energia baseadas no SPEC Power. Por fim, o SPEC Power testa o consumo energético e o desempenho de servidores utilizando aplicações *CPU/Memory-Bound* implementadas em C e Fortran.

O *High-Performance Linpack* (HPL) consiste de um pacote de softwares que resolvem sistemas lineares aleatórios de dupla precisão aritmética em arquiteturas de alto desempenho (PETITET, 2004). Ele executa um programa de ensaios e de temporização

para quantificar a precisão da solução obtida, bem como o tempo que levou para computar. Seu código é aberto e suas 7 aplicações formam uma coleção de sub-rotinas em Fortran, em sua maioria *CPU-Bound*. As implementações paralelas utilizam MPI. Ele é o principal *benchmark* que compõe o chamado *High-Performance Computing Benchmark Challenge*, que é uma lista com os 500 computadores de alto desempenho mais rápidos do mundo.

Benchmarks tradicionais que existiam antes do *Numerical Aerodynamic Simulation* (NAS), como o *benchmark* HPL, eram em sua maioria especializados em computadores vetoriais. Eles geralmente sofriam de deficiências, incluindo restrições que impediam o uso de paralelização e problemas de tamanho insuficiente, o que os tornava impróprios para sistemas altamente paralelos. Para atender a essa demanda, em 1991 foi desenvolvido o NAS (BAILEY et al., 1991). Ele foi projetado inicialmente para testar o desempenho de uma arquitetura que deveria ser capaz de simular todo o sistema de um veículo espacial. Atualmente, o NAS é um pequeno conjunto de programas de código aberto que servem para avaliar o desempenho de supercomputadores paralelos. O *benchmark* é derivado de aplicações físicas de dinâmica de fluídos e consiste em quatro núcleos e três pseudo-aplicações. Os quatro núcleos são: 1) EP - *Embarrassingly Parallel*; 2) CG - *Conjugate Gradient*; 3) MG - *Multi-Grid on a sequence of meshes*; e 4) FT - *Discrete 3D fast Fourier Transform*. As três pseudo-aplicações são: 1) BT - *Block Tri-diagonal solver*; 2) SP - *Scalar Penta-diagonal solver*; e 3) LU - *Lower-Upper Gauss-Seidel solver*. Essas aplicações estão implementadas em um modelo híbrido de MPI e OpenMP.

O Adept Benchmark é utilizado para medir o desempenho e o consumo de energia de arquiteturas paralelas. Seu código é aberto e está dividido em 4 conjuntos: Nano, Micro, Kernel e Aplicação. O conjunto Micro, por exemplo, consiste de 12 aplicações sequenciais e paralelas com OpenMP, focando em aspectos específicos do sistema, como gerenciamento de processos, cache, entre outros. Por outro lado, o conjunto Kernel é o que mais se assemelha ao *benchmark* proposto neste trabalho, portanto ele será adotado para fins de comparação. Ele possui 10 aplicações implementadas de modo sequencial e paralela com OpenMP, MPI e uma delas em UPC (*Unified Parallel* C).

2.5.2 Contexto deste Trabalho

O *benchmarks* abordado neste trabalho consiste de 13 aplicações implementadas em C e suas complexidades variam de $O(n)$ à $O(n^3)$. As aplicações estão paralelizadas em 4 IPPs, sendo elas: PThreads, OpenMP, MPI-1 e MPI-2. Essas IPPs são o alvo deste trabalho por serem as mais difundidas no âmbito acadêmico e também por serem suportadas pela maior parte das arquiteturas multinúcleo, tanto embarcadas quanto de propósito geral. Portanto, o objetivo deste *benchmark* é fornecer ao usuário uma ferramenta que permita avaliar o desempenho e o consumo de energia de diferentes IPPs em arquiteturas

multinúcleo embarcadas e de propósito geral.

2.5.3 Comparação entre os Benchmarks

Através da identificação das principais características do *benchmark* discutido neste trabalho, foi possível realizar uma comparação em relação aos *benchmarks* relacionados. A Tabela 1 mostra todos os *benchmarks* apresentados neste trabalho e analisa-os de acordo com um determinado conjunto de critérios pertinentes a este trabalho.

Tabela 1 – Pontos de comparação entre os *benchmarks*

Critérios	EEMBC	MiBench	ParMiBench	SPEC	Linpack	NAS	Adept	Benchmark Proposto
Executa em qualquer arquitetura multinúcleo	X	X	X	X	X	X	X	X
Inclui sistemas embarcados	X	X	X	–	–	–	X	X
Avalia consumo de energia	X	–	–	X	–	–	X	X
Possui aplicações paralelizadas	–	–	X	–	X	X	X	X
Implementa as IPPs alvo	–	–	–	–	–	–	–	X
Possui códigos abertos à comunidade	–	X	X	–	X	X	X	X

Analisando a tabela nota-se que, embora todos os *benchmarks* possuam aplicações capazes de executar em diferentes arquiteturas multinúcleo, alguns deles não possuem um foco em sistemas embarcados (SPEC, Linpack e NAS), somente em computadores de propósito geral e supercomputadores. Também pode-se observar que apenas 3 dos *benchmarks* relacionados possuem aplicações que tornam possível realizar avaliações de consumo de energia, oferecendo ferramentas e instruções para a medição, sendo eles: EEMBC, SPEC e Adept.

Entre os *benchmarks* analisados, metade deles possuem um conjunto de aplicações paralelas. Por outro lado, somente o Adept utiliza diferentes IPPs, porém ainda assim não correspondem às IPPs alvo deste trabalho. Embora o NAS utilize MPI e OpenMP, seu modelo de programação híbrido não permite realizar essa comparação entre as IPPs. Por fim, a maioria dos *benchmarks* apresentados possuem código aberto, somente o EEMBC e o SPEC não se incluem nesse grupo, entretanto são oferecidas licenças acadêmicas nos dois casos.

2.6 Balanço do Capítulo

Este capítulo contextualizou os tópicos que servem de base para este trabalho. Primeiramente mostramos como que os *benchmarks* surgiram através da necessidade de ferramentas que executassem testes padronizados em diferentes sistemas. Testes com a finalidade de comparar e medir a eficiência desses sistemas. Também discutimos sobre alguns dos principais *benchmarks* voltados à sistemas paralelos de alto desempenho.

Na sequência mostramos uma visão geral sobre arquiteturas multinúcleo. Nessa parte, tratamos sobre o funcionamento de um ambiente multitarefa, incluindo como é realizada a comunicação entre os processadores através dos diferentes níveis de memória. Também mostramos que essa comunicação pode se tornar um potencial gargalo de desempenho em aplicações que possuem grande dependência de dados.

A seção 2.4 contextualizou a programação paralela através de quatro diferentes IPPs com comunicação por memória compartilhada (OpenMP e PThreads) e troca de mensagens (MPI 1 e 2) em processadores multinúcleo. Nesta parte mostramos como o OpenMP cria *threads* seguindo o modelo *fork/Join*, como o paralelismo é extraído através de diretivas de compilação e também como a sincronização ocorre através de *busy-waiting*. Em PThreads mostramos que o paralelismo é feito de forma explícita e a sincronização através de *mutex*. MPI também explora o paralelismo de forma explícita, porém vimos que faz-se necessário o uso de primitivas *send/receive* para comunicação. Ademais, MPI-2 adiciona a criação dinâmica de processos com relação à MPI-1.

Por fim, na seção 2.5 foram apresentados os principais trabalhos relacionados que foram encontrados através de um levantamento bibliográfico. Isso permitiu observar que, embora existam diversos *benchmarks* com propostas similares, nenhum deles possui todas as principais características do *benchmark* abordado neste trabalho. Com uma análise comparativa das características observou-se que o *benchmark* Adept é o que possui mais pontos em comum, porém com uma menor diversidade de aplicações implementadas em diferentes IPPs.

O próximo capítulo apresenta as 13 aplicações em suas versões sequenciais, com seus algoritmos e complexidades. Também são explicadas as abordagens utilizadas na paralelização das aplicações com as quatro IPPs.

3 *Benchmark* Estudado

Este capítulo apresenta as aplicações que compõem o *benchmark* detalhadamente, assim como os métodos e técnicas de paralelização adotados. Na seção seção 3.1 são apresentados os algoritmos sequenciais e suas principais características. As estratégias utilizadas na paralelização de cada um dos *benchmarks* estão detalhados na seção seção 3.2. Por fim, a seção 3.3 sumariza os principais aspectos abordados neste capítulo.

3.1 Apresentação das Aplicações

Existem poucos estudos que investigam eficiência energética de diferentes IPPs em sistemas embarcados e arquiteturas de propósito geral. Essas aplicações foram desenvolvidas com o propósito de estabelecer uma relação entre desempenho e consumo de energia nesses sistemas. As aplicações descritas aqui foram desenvolvidas por Lorenzon (2014), Lorenzon et al. (2015b), Lorenzon, Cera e Beck (2014), Lorenzon et al. (2015a), Lorenzon, Cera e Beck (2015) e nesta seção são apresentadas as principais características de cada uma. Cada subseção a seguir apresenta as aplicações em suas versões sequenciais e mostra detalhes sobre suas complexidades, seus algoritmos e fórmulas.

3.1.1 Cálculo do Pi

O Pi (π) é um valor irracional que estabelece uma relação numérica entre o perímetro de uma circunferência e seu diâmetro (ROY, 1990). Para realizar cálculos mais precisos, o Pi é comumente representado com 52 casas decimais. Contudo, pode-se aumentar essa precisão através de algoritmos computacionais. Existem diversos métodos para calcular o valor de Pi, estes envolvem aproximações, aproximações sucessivas e séries infinitas de somas, multiplicações e divisões. Nesta aplicação, foi utilizado o método de Gregory-Leibniz que é estabelecido pela equação 3.1 (ANDREWS; ROY, 1999).

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{\pi}{4} \tag{3.1}$$

Apesar do método de Gregory-Leibniz ser considerado ineficiente para poucas iterações, a precisão aumenta conforme aumenta-se as iterações de n, sendo que gera o Pi com uma precisão de 5 casas decimais após 500 mil iterações e de 10 casas decimais com 5 bilhões de iterações (BORWEIN; BORWEIN; DILCHER, 1989). A implementação desse algoritmo é bastante simples, inicia-se definindo o número de iterações a serem calculadas

e na sequência um laço aplica a equação 3.1. A saída consiste de um valor aproximado para Pi. Esse algoritmo é definido pelo pseudocódigo 1 e possui complexidade de $O(n)$.

Algorithm 1 Algoritmo do cálculo do Pi

```
1: Início
2: Define o número de iterações
3: for (Até atingir o número de iterações) do
4:     Aplica o método de Gregory-Leibniz
5: end for
6: Fim
```

3.1.2 Série Harmônica

A Série Harmônica é uma série finita que calcula a soma de precisão arbitrária depois do ponto decimal (GOLDSTON; YILDIRIM, 2001). Essa sequência matemática recebe esse nome por possuir proporções similares aos comprimentos de onda de uma corda a vibrar. Essa sequência diverge lentamente, conforme pode ser observado através da equação 3.2, onde n representa o valor da série harmônica a ser calculado.

$$\sum_{i=1}^{k} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots \tag{3.2}$$

Essa aplicação contém ainda um vetor que armazena a soma da precisão em cada i do algoritmo. Seu principal processamento consiste de sucessivas operações de divisão e cálculo do módulo de cada valor de i. A saída dessa aplicação consiste do somatório dessas operações. Esse algoritmo está descrito no pseudocódigo 2 e possui complexidade de $O(n * d)$, sendo n o número de iterações e d o tamanho do vetor.

Algorithm 2 Série Harmônica

```
1: Início
2: Define o número de iterações
3: for (Até atingir o número de iterações) do
4:     for (Percorre vetor) do
5:         Cálculo da Série Harmônica
6:         Armazena a soma da precisão a cada iteração
7:     end for
8: end for
9: for (Percorre vetor) do
10:    Ajuste da precisão decimal
11: end for
12: Fim
```

3.1.3 Dijkstra

O algoritmo de Dijkstra, concebido pelo cientista da computação holandês Edsger Dijkstra, soluciona o problema do caminho mais curto num grafo dirigido ou não dirigido com arestas de peso não negativo. Dado um vértice fonte no grafo, o algoritmo encontra o caminho com menor custo entre este vértice e qualquer outro vértice (DIJKSTRA, 1959).

Nossa implementação utiliza uma matriz de adjacência de tamanho $N \times N$. Considerando n sendo o número de vértices, a complexidade do algoritmo é de $O(n^2)$. Contudo, nossa aplicação considera o menor caminho de n para n vértices, elevando sua complexidade total para $O(n^3)$. A saída dessa aplicação é um vetor contendo a distância mínima entre cada um dos vértices como saída. Este algoritmo está representado no pseudocódigo 3.

Algorithm 3 Algoritmo de Dijkstra

```
1: Início
2: for (Percorre vértices) do
3:     for (Percorre vetor de distâncias) do
4:         Atribui a maior distância possível até todos os vértices
5:         Define cada vértice como não-visitado
6:     end for
7:     Atribui distância nula até o próprio vértice
8:     for (Percorre vértices) do
9:         for (Percorre vértices) do
10:            if (Se o vértice é vizinho e ainda não foi visitado) then
11:                Atualiza vértice corrente na matriz
12:            end if
13:        end for
14:        Define vértice corrente como visitado
15:        for (Percorre vetor de distâncias) do
16:            if (Verifica se é o menor caminho) then
17:                Atualiza caminho no vetor de soluções
18:            end if
19:        end for
20:    end for
21: end for
```

3.1.4 Método de Jacobi

O método de Jacobi é um método clássico que data do final do século XVIII. Técnicas iterativas são raramente utilizadas para solucionar sistemas lineares de pequenas dimensões, já que o tempo requerido para obter um mínimo de precisão ultrapassa o requerido pelas técnicas diretas como a Eliminação Gaussiana (BURDEN; FAIRES, 2003). Contudo, para sistemas grandes, com grande porcentagem de entradas nulas (sistemas esparsos), essas técnicas aparecem como alternativas mais eficientes. Considerando um sistema linear do tipo $Ax = b$, em que A é a matriz dos coeficientes $m \times n$, x é o vetor

de variáveis e b o vetor dos termos constantes. O objetivo do método é encontrar um resultado aproximado para x através da convergência dos vetores (PRESS et al., 2007). A equação 3.3 representa esse método.

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), i = 1, 2, \ldots, n. \tag{3.3}$$

O programa consiste de um laço de repetição em que a cada iteração percorre a matriz alvo e aplica o método de Jacobi. Devido ao algoritmo realizar as operações sobre uma única matriz, isso gera dependência de dados. Para contornar esse problema, é utilizada uma matriz auxiliar, que conterá o valor correto ao final de cada computação. Esse algoritmo está descrito no pseudocódigo 4 e possui complexidade de $O(n^3)$.

Algorithm 4 Método de Jacobi

```
1: Início
2: Define o número de iterações
3: for (Até atingir o número de iterações) do
4:     for (Percorre linhas da matriz) do
5:         for (Percorre colunas da matriz) do
6:             Aplica o método de Jacobi
7:             Armazena o resultado em uma matriz auxiliar
8:         end for
9:     end for
10:    Copia os dados da matriz auxiliar para a matriz solução
11: end for
12: Fim
```

3.1.5 Multiplicação de Matrizes

Existem diversos métodos para realizar a multiplicação de matrizes. O programa que utilizamos em nosso trabalho implementa o método mais comum que consiste da multiplicação dos elementos das linhas da matriz A pelos elementos das colunas da matriz B (PRESS et al., 2007). Essa técnica é expressa através da equação 3.4.

$$(AB)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj}. \tag{3.4}$$

O cálculo acontece através da multiplicação das linhas da matriz de entrada A pelas colunas da matriz de entrada B. O valor de cada computação é armazenado na respectiva posição na matriz de saída C. Esse algoritmo possui complexidade de $O(n^3)$ e é definido pelo pseudocódigo 5.

Algorithm 5 Multiplicação de Matrizes

```
1: Início
2: for (Percorre linhas) do
3:     for (Percorre colunas) do
4:         for (Percorre elementos) do
5:             Multiplica os elementos das linhas e colunas
6:             Armazena o resultado na matriz solução
7:         end for
8:     end for
9: end for
10: Fim
```

3.1.6 Similaridade entre Histogramas

Esse algoritmo consiste em reconhecer padrões entre histogramas de diferentes imagens. A comparação dos histogramas é feita através de medidas de similaridades calculadas pela distância de Hellinger, descrita nos termos do coeficiente de Bhattacharyya (OLIVEIRA; SCHARCANSKI, 2010). A equação da distância de Hellinger é definida em 3.5, onde d é a distância e $\rho[p,q]$ é o coeficiente de Bhattacharyya definido em 3.6. Nele p_n é o n-ézimo *bin* do histograma de níveis de cinza da região a ser comparada e q_n o n-ézimo *bin* do histograma calculado para uma região da imagem.

$$d = \sqrt{1 - \rho[p,q]} \tag{3.5}$$

$$\rho[p,q] = \sum_{n=1}^{m} \sqrt{p_n, q_n} \tag{3.6}$$

O programa percorre uma imagem pixel a pixel comparando o histograma dos pixeis adjacentes com o vetor de histogramas de outra imagem. Os histogramas utilizados para calcular a similaridade das regiões são extraídos de regiões retangulares da imagem em níveis de cinza, definidas por uma janela de $n \times m$ pixeis, utilizando 256 *bins*. Esses histogramas são normalizados pelo número total de pixeis contidos na região retangular. A saída consiste de uma nova imagem indicando quais pixeis da imagem são similares ao vetor de histogramas de entrada. Esse programa é definido pelo algoritmo 6 e sua complexidade é $O(n^2)$.

Para cada pixel da imagem, é definido um retângulo em torno dele (com as mesmas dimensões do retângulo que contém o objeto alvo) e extraído um histograma, o qual será comparado com o histograma da região a ser analisada. Dessa forma é possível calcular a similaridade de todos os pixeis da imagem com a região selecionada.

Algorithm 6 Similaridade entre Histogramas

```
1: Início
2: for (Percorre linhas da matriz de píxeis) do
3:     for (Percorre colunas da matriz de píxeis) do
4:         Cálculo do histograma em níveis de cinza da imagem
5:         Cálculo da similaridade entre histogramas
6:         Armazena o resultado na matriz de píxeis
7:     end for
8: end for
9: Fim
```

3.1.7 Produto Escalar

O produto escalar é uma operação entre dois vetores cujo o resultado é um número real (também chamado de escalar) (CALLIOLI; DOMINGUES; COSTA, 2007). Em nosso algoritmo calculamos o produto escalar entre uma sequência de valores ordenados e outra de ordenação inversa. Esse algoritmo pode ser visto no pseudocódigo e é definido pela equação 3.7. Sua complexidade é $O(n)$.

$$A \cdot B = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \tag{3.7}$$

Algorithm 7 Produto Escalar

```
1: Início
2: Define o número de iterações
3: for (Até atingir o numero de iterações) do
4:     Cálculo do produto escalar
5: end for
6: Fim
```

3.1.8 Jogo da Vida

O jogo da vida é um autômato celular desenvolvido pelo matemático John Horton Conway em 1970. O jogo foi criado de modo a utilizar regras simples para reproduzir as alterações e mudanças em grupos de seres vivos, podendo ser aplicado a diversas áreas da ciência (GARDNER, 1970). As regras definidas são aplicadas a cada nova geração, assim, a partir de uma imagem em um tabuleiro bidimensional definida pelo jogador, percebem-se mudanças a cada nova geração, variando entre padrões fixos e caóticos.

O jogo simula a evolução da vida em uma sociedade representada por uma estrutura bidimensional. A evolução é baseada em leis genéticas definidas por Conway, levando em consideração o estado das células vizinhas (GARDNER, 1970). O algoritmo consiste de percorrer posição por posição de uma matriz de tamanho $N \times N$ computando as leis

genéticas. Este processo é repetido até que a evolução da sociedade esteja satisfeita e a saída do algoritmo consiste de uma matriz contendo a sociedade evoluída após i gerações. A implementação desse programa é representada pelo algoritmo 8 e possui complexidade de $O(m * n^2)$, onde m é o número de evoluções da sociedade e n o tamanho da matriz.

Algorithm 8 Jogo da Vida

```
1: Início
2: Define o número de gerações
3: for (Até atingir o número de gerações) do
4:     for (Percorre linhas da matriz) do
5:         for (Percorre colunas da matriz) do
6:             Conta o número de células vivas
7:             if Existem 3 vizinhos vivos then
8:                 Continua viva na próxima geração
9:             else if Exitem 2 vizinhos vivos e a célula já está viva then
10:                Continua viva na próxima geração
11:            else
12:                Célula morre
13:            end if
14:        end for
15:    end for
16:    Atualiza os vizinhos na matriz solução
17: end for
18: Fim
```

3.1.9 Integração Numérica

A integral de uma função foi originalmente criada para determinar a área sob uma curva no plano cartesiano através de técnicas de aproximação. O processo de se calcular a integral de uma função é chamado de integração (STEWART, 2001). Nossa aplicação faz a integração da função $f(x)$ no intervalo entre a e b. O método básico envolvido nesta aproximação é chamado de quadratura numérica e é expresso pela fórmula 3.8, onde α_i são coeficientes reais (peso da função) e x_i, são pontos de $[a, b]$. O algoritmo 9, que possui complexidade de $O(n)$, representa essa aplicação.

$$\int_b^a f(x)dx \simeq \sum_{i=0}^{n} \alpha_i f(x_i) \tag{3.8}$$

Algorithm 9 Integração Numérica

```
Início
Define número de iterações
Define limites a e b
for (Até atingir o número de iterações) do
    Calcula f(x) de a até b
end for
Fim
```

3.1.10 Gram-Schmidt

O processo de Gram-Schmidt consiste de um método para ortonormalizar um conjunto de vetores em um espaço de produto interno, normalmente o espaço Euclidiano R^n (CHENEY; KINCAID, 2009). O processo de Gram–Schmidt recebe um conjunto finito e linearmente independente de vetores $S = v_1, \ldots, v_n$ e retorna um conjunto ortonormal $S' = u_1, \ldots, u_n$ que gera o mesmo subespaço S inicial. O processo realiza uma série de operações de projeção entre os vetores de entrada. Tais operações são definidas pela equação 3.9.

$$v = \frac{\langle v, v_1 \rangle}{\|v_1\|^2} v_1 + \cdots + \frac{\langle v, v_n \rangle}{\|v_n\|^2} v_n \tag{3.9}$$

O algoritmo possui uma matriz de vetores de entrada, e após sucessivas computações em n etapas, que são computadas uma após a outra, onde a computação de $n+1$ depende do resultado de n, uma nova matriz de saída é gerada. Esse algoritmo é representado pelo pseudocódigo 10 e sua complexidade é $O(n^3)$.

Algorithm 10 Processo de Gram-Schmidt

```
Início
for (Percorre linhas da matriz) do
    for (Percorre colunas da matriz) do
        Cálculo do produto interno
    end for
    for (Percorre colunas da matriz) do
        Cálculo da norma do vetor resultante
    end for
    for (Percorre colunas da matriz) do
        for (Percorre elementos da matriz) do
            Normaliza o vetor e armazena na matriz de soluções
        end for
    end for
end for
Fim
```

3.1.11 Ordenação Par-Ímpar

A ordenação Par-Ímpar é um algoritmo de ordenação que funciona através da comparação de todos os pares indexados (ímpar-par) de elementos adjacentes na lista e, se um par está na ordem errada (o primeiro é maior do que o segundo), os elementos são trocados. O próximo passo repete isso para os pares indexados (par-ímpar) de elementos adjacentes. Em seguida, ele alterna entre etapas de (ímpar-par) e (par-ímpar) até que a lista esteja ordenada (KNUTH, 1998). A saída desse algoritmo é o vetor de entrada ordenado. Sua implementação é representada pelo algoritmo 11 e se baseia no *bubble-sort*. Sua complexidade é $O(n^2)$.

Algorithm 11 Ordenação Par-Ímpar

```
1: Início
2: Define tamanho do vetor (n)
3: for (Percorre vetor) do
4:     for (De 0 até n − 1 passo 2) do
5:         if (Elemento na posição atual for maior que o elemento na posição sequinte) then
6:             Troca elementos de posição
7:         end if
8:     end for
9:     for (De 1 até n − 1 passo 2) do
10:        if (Elemento na posição atual for maior que o elemento na posição sequinte) then
11:            Troca elementos de posição
12:        end if
13:    end for
14: end for
15: Fim
```

3.1.12 Turing Ring

Alan Turing analisou a interação de dois produtos químicos em um anel de células usando duas equações diferenciais acopladas para descrever um sistema presa/predador (TURING, 1952). É um sistema espacial em que predadores e presas interagem em um mesmo local. O sistema simula a iteração e evolução entre presas e predadores através da utilização de duas equações diferenciais 3.10 e 3.11, onde $r_{X,Y}$ são as taxas de nascimento, $c_{a,b}$ representam interações locais e $\mu_{X,Y}$ são as taxas de migração entre células vizinhas. A evolução é definida de acordo com as células vizinhas (PAUDEL; AMARAL, 2011).

$$\frac{dX_i}{dt} = X_i(r_X + c_{XX}X_i + c_{XY}Y_i) + \mu_X(X_{i+1} + X_{i-1} - 2X_i) \tag{3.10}$$

$$\frac{dY_i}{dt} = Y_i(r_Y + c_{YX}X_i + c_{YY}Y_i) + \mu_Y(Y_{i+1} + Y_{i-1} - 2Y_i) \tag{3.11}$$

O algoritmo consiste de uma matriz de entrada contendo em cada posição o número de predadores e presas, que irão interagir durante n evoluções produzindo uma

sociedade (matriz) de saída. Esse algoritmo é representado pelo pseudocódigo 12 e possui complexidade de $O(m * n^2)$, onde m é o número de evoluções da sociedade e n o tamanho da matriz.

Algorithm 12 Turing Ring

```
1: Início
2: Define número de evoluções n
3: for (De 0 até n) do
4:     for (Percorre linhas da matriz) do
5:         for (Percorre colunas da matriz) do
6:             Simula a evolução aplicando as equações diferenciais
7:             Armazena a solução em uma matriz auxilar
8:         end for
9:     end for
10:    for (Percorre linhas da matriz) do
11:        for (Percorre colunas da matriz) do
12:            Armazena os dados da matriz auxiliar na matriz original
13:        end for
14:    end for
15: end for
16: Fim
```

3.1.13 Transformada Discreta de Fourier

A *Discrete Fourier Transform* (DFT) é uma função que mapeia um vetor de n números complexos para outro vetor de n números complexos. Os valores das frequências resultantes são múltiplos inteiros de uma frequência fundamental, cujo período corresponde ao comprimento do intervalo da amostragem (SMITH et al., 1997). Essa função é muito utilizada em processamento de sinais digitais e é definida pela equação 3.12, sendo que:

- N = Número de amostras;
- n = Amostra atual (de 0 até $N - 1$);
- x_n = Nível do sinal no instante de tempo n;
- k = Frequência atual (de 0 Hz até $N - 1$ Hz);
- X_k = Nível da frequência k no sinal.

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot \left(cos\left(2\pi k \frac{n}{N}\right) + j sin\left(2\pi k \frac{n}{N}\right) \right), \quad n \in \mathbb{Z} \tag{3.12}$$

O algoritmo possui dois cálculos principais que representam o somatório dos termos reais e imaginários que compõem a equação. Esses cálculos são realizados separadamente e somados no final, conforme é possível observar no pseudocódigo 13. Esse algoritmo possui complexidade de $O(n^2)$.

Algorithm 13 Transformada Discreta de Fourier

```
1: Início
2: for (Percorre vetor) do
3:     for (Percorre vetor) do
4:         Cálculo da parte real da função
5:     end for
6:     for (Percorre vetor) do
7:         Cálculo da parte imaginária da função
8:     end for
9:     Soma as duas partes e armazena o resultado no vetor solução
10: end for
11: Fim
```

3.2 Paralelização das Aplicações

Nesta seção mostraremos como foi feita a paralelização do conjunto de *benchmarks* de um modo geral. Detalhes individuais da implementação paralela de cada programa são abordados no Apêndice A. As versões paralelas foram desenvolvidas utilizando 4 IPPs diferentes: OpenMP, PThreads e MPI 1 e 2. A paralelização de um programa sequencial pode ocorrer de diversas formas e utilizar técnicas não adequadas pode impactar no desempenho da aplicação. Para minimizar esse problema, as implementações com as diferentes IPPs foram baseadas em indicações de Foster (1995), Wilson e Bal (1996), Butenhof (1997), Gropp, Lusk e Thakur (1999) e Rauber e Rünger (2010).

Rauber e Rünger (2010) propõem que a paralelização seja feita de forma sistemática, segundo ele, existem três etapas fundamentais para a paralelização de uma aplicação sequencial, que são:

- **Decomposição da Computação**: consiste em decompor um programa sequencial em diferentes instruções que possam ser executadas concorrentemente, ou seja, gerar tarefas suficientes para manter todos os núcleos do processador ocupados durante a execução do programa;
- **Atribuição das tarefas para processos/*threads***: é atribuir as tarefas de forma que haja um bom balanceamento de carga, isto é, cada processo ou segmento deve ter aproximadamente o mesmo número de cálculos a serem executados;
- **Mapeamento dos processos/*threads* em unidades físicas de processamento**: realizada pelo algoritmo de escalonamento do sistema operacional, o objetivo prin-

cipal da etapa de mapeamento é obter uma utilização equilibrada dos núcleos, mantendo a menor comunicação possível entre os processadores.

A etapa de decomposição da computação, também conhecida como particionamento, decompõe os dados e os cálculos relacionados em pequenos conjuntos. Segundo Foster (1995), essa decomposição dos dados deve ser a primeira abordagem utilizada quando tem-se grandes estruturas de dados como entrada ou saída de um problema, ou também dados que são acessados com grande frequência. Esse método é definido como a decomposição do domínio de um programa.

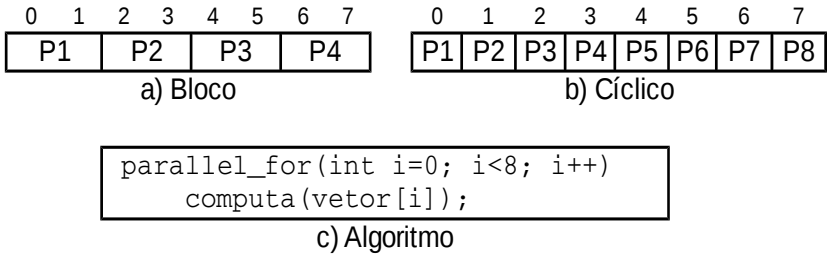
A decomposição da computação e atribuição das tarefas para processos/*threads* ocorreram de forma explícita na paralelização com PThreads e MPI 1 e 2, visando obter o melhor balanceamento da carga de trabalho. Também foram incluídas funções de troca de mensagens entre os processos, além da criação dinâmica de processos em MPI-2. Para a paralelização com OpenMP foram utilizados laços paralelos com granularidade fina e grossa. Segundo Foster (1995), Chapman, Jost e Pas (2008) e Rauber e Rünger (2010), esta técnica é mais apropriada para paralelizar aplicações que realizam cálculos iterativos e que percorrem estruturas de dados contíguos (ex. matriz, vetor, etc.). Para cada estrutura de dados adotou-se um modelo de paralelização específico.

3.2.1 Aplicações que utilizam vetores

As aplicações Ordenação Par-Impar e Série Harmônica utilizam estruturas de dados com uma única dimensão (vetores) para realizar a computação. Dessa forma, para paralelizar essas aplicações foi utilizada a decomposição de domínio em blocos de uma dimensão (1D), conforme apresentado na Figura 7. Neste modelo, cada *thread*/processo computa sobre diferentes elementos do vetor, que são distribuídos em forma de blocos de iterações (Figura 7a); ou cíclica (Figura 7b). A forma como as iterações são atribuídas entre as *threads* diferem. Enquanto na divisão por blocos as iterações são atribuídas de uma única vez, na cíclica elas são atribuídas conforme solicitação.

A Figura 7c apresenta um possível algoritmo para este modelo de decomposição. Nele, existe um laço `parallel_for`, que define que as iterações do laço `for` que percorre as posições do vetor, serão distribuídas entre as *threads*/processos e computadas de forma concorrente. Na paralelização com PThreads e MPI 1 e 2, foi utilizada a distribuição em blocos (Figura 7a) e realizado ajustes finos na divisão. Já na paralelização com OpenMP foi utilizada a distribuição em blocos (estática) para granularidade grossa e cíclica (dinâmica) para granularidade fina (Figura 7b) (LORENZON, 2014).

Figura 7 – Decomposição de domínio 1D

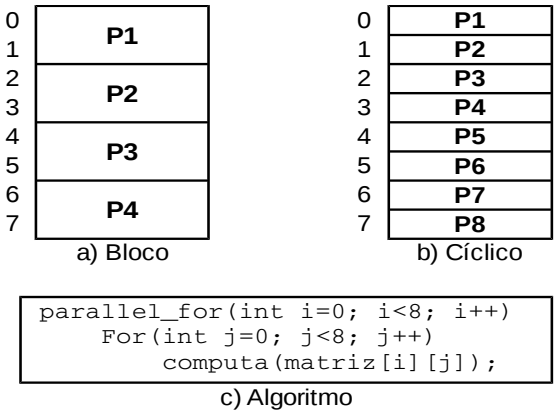


Fonte: Lorenzon (2014)

3.2.2 Aplicações que utilizam matrizes

Em nosso conjunto de *benchmarks*, a maioria das aplicações possuem matrizes como entrada ou saída de dados. As aplicações que manipulam esse tipo de estrutura de dados bidimensionais são: Dijkstra, Jacobi, Multiplicação de Matrizes, Similaridade entre Histogramas, Jogo da vida, Gram-Schmidt, Turing Ring e a Transformada Discreta de Fourier. Dessa forma, a decomposição de domínio em blocos de duas dimensões (2D), apresentada na Figura 8, foi a estratégia de paralelização utilizada. O algoritmo deste modelo é apresentado na Figura 8c. Ele mostra que o laço `for` a ser paralelizado é o mais externo (`parallel_for`), ou seja, o laço que controla a distribuição das linhas entre as *threads*/processos.

Figura 8 – Decomposição de domínio 2D



Fonte: Lorenzon (2014)

Assim como no caso anterior, na paralelização com PThreads, MPI-1 e MPI-2, foi usada a decomposição em bloco (Figura 8a) e foram realizados ajustes finos na paralelização. Contudo, na paralelização com OpenMP foi utilizada a distribuição em

blocos (estática) para granularidade grossa e cíclica (dinâmica) para granularidade fina (Figura 8b).

3.2.3 Aplicações que não operam sobre dados estruturados

O *benchmark* inclui também três aplicações que não computam sobre dados estruturados: Cálculo do Pi, Produto Escalar e Integração Numérica. Essas aplicações possuem uma implementação bastante simples, onde existe apenas um laço que realiza a computação principal. Nesses casos a paralelização ocorreu através da divisão do número de iterações dos laços pelo número de *threads*/processos de forma estática. Esse padrão de distribuição da carga da trabalho foi adotado para as quatro IPPs.

Ao final do laço, cada aplicação deve realizar uma redução para que os resultados parciais computados separadamente sejam unificados na solução final. Essa operação é realizada através do uso de variáveis do tipo `mutex` em PThreads ou com funções do tipo `reduce` nas demais IPPs.

3.3 Balanço do capítulo

Neste capítulo foram apresentadas as aplicações em suas versões sequenciais. Foram discutidos os principais detalhes de suas implementações, assim como suas complexidades e seus algoritmos. Também foram discutidas as estratégias de paralelização adotadas utilizando as quatro IPPs, onde mostramos que foram definidas diferentes políticas de distribuição de carga de trabalho de acordo com as características de cada algoritmo.

Para as aplicações que computam sobre dados estruturados, a distribuição de carga são muito semelhantes. Sendo que nas implementações com PThreads e MPI 1 e 2 a melhor estratégia é a decomposição em blocos, tanto para vetores quanto para matrizes. Já com OpenMP a distribuição de carga ocorreu de forma estática ou dinâmica, de acordo com a granularidade de cada aplicação

Cada aplicação possui suas próprias características e comportamentos variados. Através do estudo dessas características foi possível fazer uma diferenciação mais precisa entre as aplicações. Isso permitiu fazer uma classificação dessas aplicações segundo critérios relevantes ao nosso trabalho. O próximo capítulo mostra mais detalhes sobre essas diferentes classificações e seus critérios, incluindo o histórico de classificações utilizadas em estudos anteriores.

4 Classificação do *Benchmark*

As classificações atuais do conjunto de aplicações que compõem o *benchmark* já foram investigadas em trabalhos anteriores seguindo diferentes critérios (LORENZON, 2014) (LORENZON et al., 2015b) (LORENZON; CERA; BECK, 2014) (LORENZON et al., 2015a) (LORENZON; CERA; BECK, 2015). Nesses trabalhos, os autores classificaram as aplicações de diferentes formas, que evoluíram com os estudos mais recentes. Este trabalho surgiu com o objetivo de unificar esses estudos em um único trabalho. Porém, além das classificações já realizadas nos estudos prévios, neste trabalho analisamos uma série de fatores para comprovar que o conjunto de aplicações do *benchmark* possui diversidade o suficiente para ser considerado um *benchmark* para avaliar desempenho e consumo de energia.

Na seção 4.1 deste capítulo, faremos um levantamento do histórico de classificações utilizadas nos estudos anteriores, onde mostramos como o conjunto de aplicações do *benchmark* evoluiu no decorrer do tempo. A seção seguinte 4.2, apresenta a definição dos critérios considerados para organizar a classificação do *benchmark* neste trabalho. Por fim, na seção 4.3 é realizada uma revisão dos principais aspectos abordados neste Capítulo.

4.1 Histórico de Classificações

Parte do conjunto de aplicações que compõem nosso *benchmark* já foi utilizado com o propósito de analisar o desempenho e o consumo de energia em trabalhos anteriores. Nesses trabalhos, as aplicações foram classificadas de acordo com os seguintes critérios:

- Quantidade de acessos à memória - representa a quantidade de acessos que foram feitos à endereços de memória compartilhada e privada do processador, considerando leitura e escrita para cada aplicação;
- Dependência de dados - significa que pelo menos uma *thread* só poderá iniciar sua execução quando o resultado da computação de uma ou mais *threads* estiver finalizado, isso mostra a existência de comunicação entre as *threads*/processos;
- Pontos de sincronização - determinam que em certos momentos durante a execução de uma aplicação, todas as *threads*/processos precisarão serem sincronizadas antes de uma nova tarefa iniciar.
- TLP - mostra o quão ocupado fica o processador durante a execução de uma aplicação;

- Quantidade de operações de troca de dados - nas IPPs alvo, essas operações representam *barriers*, *locks*/*unlocks* e operações de criação e finalização de *threads*/processos.
- Taxa de comunicação - representa o volume de comunicação requerida pelas *threads*/processos durante a execução da aplicação. Essa taxa é dividida entre alta, média e baixa comunicação.
  - Alta comunicação: significa que existem trocas de dados constantes entre as *threads*/processos durante a execução;
  - Média comunicação: ocorre quando *threads*/processos em execução realizam acessos de leitura à endereços de memória compartilhada;
  - Baixa comunicação: a taxa de comunicação é considerada baixa, quando a comunicação ocorre apenas na distribuição da carga de trabalho e no retorno dos resultados no final da computação.

Para realizar a classificação através desses critérios, os autores coletaram dados utilizando ferramentas e interfaces de apoio, que fazem uso de contadores de *hardware* dentro de cada processador. As principais ferramentas utilizadas foram: *Performance Application Programming Interface* (PAPI), a qual disponibiliza uma estrutura para acessar os contadores de desempenho disponíveis na maioria dos processadores atuais (ICL, 2015) e Intel® Pin Tool, que permite coletar dados do processador em tempo de execução (INTEL, 2012). Essas ferramentas fornecem dados como: número de instruções executadas, número total de ciclos, acessos de leitura e escrita à endereços memória, entre outros.

4.1.1 *CPU-Bound* e *Memory-Bound*

No primeiro trabalho, Lorenzon, Cera e Beck (2014) classificaram as aplicações em duas categorias: *CPU-Bound* e *Memory-Bound*. *CPU-Bound* é quando o tempo de processamento depende mais do processador do que das entradas e saídas. Já o termo *Memory-Bound* refere-se a uma situação em que o tempo para completar um determinado problema computacional é decidida em primeiro lugar pela quantidade de memória necessária para armazenar dados (ABADI et al., 2005). Em outras palavras, o fator de limitação de resolução de um determinado problema é a velocidade de acesso à memória.

As aplicações foram primeiramente classificadas de acordo com o número de instruções executadas e quantidade de acessos à memória. As aplicações que fazem mais acessos a memória foram classificadas como *Memory-Bound*. Em contrapartida, as aplicações que fazem um maior uso do processador foram classificadas como *CPU-Bound*.

Nessa primeira versão do *benchmark* foram utilizadas 8 aplicações. Entre elas: Jogo da vida (*Game of Life* (GL)), Gram-Schmidt (GS), Decomposição-LU (LU) e Mul-

Figura 9 – Principais características da primeira classificação do *benchmark*.

Benchmark		Write (%)		Read (%)		Total (%)		Data Dependency	Sync. Points	TLP			Input Size
		Priv.	Shared	Priv.	Shared	Priv.	Shared			2	3	4	
MEM-B	GL	66.63	33.37	68.34	31.66	68.25	31.75	Yes	Yes	1.91	2.75	3.33	4,096×4,096
	GS	70.49	29.51	53.98	46.02	56.01	43.99	Yes	Yes	1.99	2.98	3.84	2,048×2,048
	LU	99.42	0.58	61.69	38.31	63.05	36.95	Yes	Yes	1.98	2.93	3.66	2,048×2,048
	MM	50.40	49.60	58.88	41.12	57.98	42.02	No	Yes	2.00	2.99	3.87	2,048×2,048
CPU-B	PI	99.99	0.01	96.30	3.70	98.51	1.49	No	No	2.00	3.00	3.92	4 billions
	MS	99.99	0.01	95.80	4.20	97.21	2.79	No	No	1.70	2.77	3.74	1,024×768
	DJ	99.99	0.01	71.94	28.06	73.86	26.14	No	No	2.00	2.96	3.86	2,048×2,048
	HS	99.99	0.01	87.17	12.83	89.90	10.10	No	No	1.96	2.90	3.70	1,920×1,080

Fonte: Lorenzon, Cera e Beck (2014)

tiplicação de Matrizes (MM) eram consideradas *Memory-Bound*. Entre as *CPU-Bound* estavam: Cálculo do Pi (PI), Conjunto de Mandelbrot (*Mandelbrot Set* (MS)), Dijkstra (DJ) e Similaridade entre Histogramas (HS). A Figura 9 mostra essas aplicações classificadas de acordo com os critérios que foram utilizados.

Os critérios utilizados nesta etapa foram: quantidade de acessos à memória, dependência de dados, quantidade de pontos de sincronização e o TLP de cada aplicação. Para o primeiro critério, foram coletados individualmente a quantidade de escrita e a quantidade de leitura na memória, tanto para memória privada quanto para memória compartilhada. Além disso, a tabela mostra o total de acessos de leitura e escrita a essas memórias.

Dependência de dados também foi um critério estudado nesse trabalho, isso indica que existem *threads* durante a execução que necessitam aguardar a finalização de outras *threads* para executar. Da mesma forma, foi considerada a existência de pontos de sincronização nas aplicações, que indica que em determinado momento da execução todas as *threads*/processos precisam ser sincronizadas. Por fim, a tabela mostra o TLP das aplicações, considerando a média entre as IPPs e executando com 2, 3 e 4 *threads*, sendo que 4 era o limite imposto pelo processador. Mais detalhes sobre o TLP são apresentados na subseção 4.2.1.

A coleta desses dados utilizados para definir cada critério foi feita através da ferramenta PAPI e Pin Tool, que permitiram a obtenção das seguintes informações: quantidade de acessos à memória (*cache* e principal), número de instruções executadas e, por fim, a quantidade de ciclos do processador que cada aplicação utilizou para executar. Cada valor representa a média de um total de 10 execuções para cada aplicação.

Figura 10 – Principais características da segunda classificação do *benchmark*.

Benchmarks		Write (%)		Read (%)		Total (%)		Communication Rate	TLP				Input Size
		Private	Shared	Private	Shared	Private	Shared		2	3	4	8	
CPU-B	Calculation of the PI	99.99	0.01	96.30	3.70	98.51	1.49	Low	2.00	3.00	3.92	7.89	4 billions
	Harmonic Series	99.99	0.01	93.21	6.71	94.91	5.09	Low	2.00	2.98	3.92	7.86	100000 elem.
WMEM-B	Dijkstra	99.99	0.01	71.94	28.06	73.86	26.14	Medium	2.00	2.96	3.86	7.68	2048 x 2048
	Jacobi	92.99	7.01	54.64	45.36	58.16	41.84	Medium	2.00	2.99	3.88	7.70	2048 x 2048
	LU-Decomposition	99.42	0.58	61.69	38.31	63.05	36.95	Medium	1.98	2.93	3.76	7.59	2048 x 2048
	Matrix Multiplication	50.40	49.60*	58.88	41.12	57.98	42.02	Medium	2.00	2.99	3.87	7.69	2048 x 2048
	Similarity of Histograms	99.99	0.01	87.17	12.83	89.90	10.10	Medium	1.96	2.90	3.80	7.62	1920 x 1080
MEM-B	Game of Life	66.63	33.37	68.34	31.66	68.25	31.75	High	1.91	2.75	3.33	7.15	4096 x 4096
	Gram Schmidt	70.49	29.51	53.98	46.02	56.01	43.99	High	1.99	2.98	3.74	7.34	2048 x 2048
	Odd-Even Sort	58.56	41.44	50.39	49.61	52.02	47.98	High	1.99	2.86	3.39	7.19	150000 elem.
	Turing-Ring	59.93	40.07	82.24	17.76	69.73	30.27	High	2.00	2.97	3.73	7.29	2048 x 2048

Fonte: Lorenzon, Cera e Beck (2015)

4.1.2 *CPU Bound*, *Weakly Memory-Bound* e *Memory-Bound*

Em um segundo momento, Lorenzon, Cera e Beck (2015) incluíram nos critérios anteriores a taxa de comunicação entre as *threads*/processos. Isso tornou possível realizar uma divisão mais precisa do conjunto de aplicações do *benchmark*. Com isso, entre as classificações *CPU-Bound* e *Memory-Bound* foi adicionada a classificação *Weakly Memory-Bound*. Essa nova classificação representa aplicações que fazem um maior uso do processador e que também possuem muitos acessos de leitura/escrita à endereços de memória compartilhados entre as *threads*, mesmo possuindo pouca dependência de dados.

Também ocorreram alterações do conjunto de aplicações. Com a nova configuração foram adicionados os algoritmos: Série Harmônica, Método de Jacobi, Ordenação Par-Ímpar e *Turing-Ring*. Por outro lado, a aplicação Conjunto de Mandelbrot foi retirada do *benchmark* por apresentar falhas de implementação e de balanceamento de carga. Afinal, processadores embarcados possuem uma pilha de recursividade menor e as operações de ponto flutuante são emuladas pelo sistema operacional, gerando *overhead* extra. Já em processadores de propósito geral a pilha de recursividade é maior e as operações de ponto flutuante são realizadas em *hardware*. Essas características influenciam na comparação das IPPs entre os processadores. A Figura 10 mostra a configuração dessa segunda fase do *benchmark*, já incluindo a divisão de acordo com a taxa de comunicação entre *threads*/processos.

Os critérios utilizados nesta classificação foram: quantidade de acessos à memória, taxa de comunicação e o TLP de cada aplicação. Para o primeiro critério, foi utilizado o mesmo padrão de informações apresentado na subseção 4.1.1. O segundo critério foi acrescentado com o objetivo de quantificar a comunicação realizada entre *threads*/processos das aplicações. Indiferente da IPP utilizada, operações de comunicação são feitas em regiões de memória compartilhada. Geralmente essas regiões ficam mais longe do processador aumentando o tempo de acesso e o consumo de energia em relação às memórias mais próximas do processador (registradores e *caches* L1 e L2). Portanto, viu-se a necessidade

de identificar a taxa de comunicação de cada aplicação.

Para tentar antecipar futuros cenários, o ambiente de execução foi alterado. Nesse trabalho foram utilizados processadores para sistemas embarcados e de propósito geral de 8 núcleos. Dessa forma, o TLP das aplicações foi medido utilizando 2, 3, 4 e 8 *threads*.

As informações necessárias para classificar as aplicações foram coletadas através da ferramenta PAPI e Pin Tool. Essas informações consistem do número de instruções executadas; quantidade de acessos à endereços de memória compartilhados e privados; e número total de ciclos do processador.

4.1.3 Alta e Baixa Demanda de Comunicação

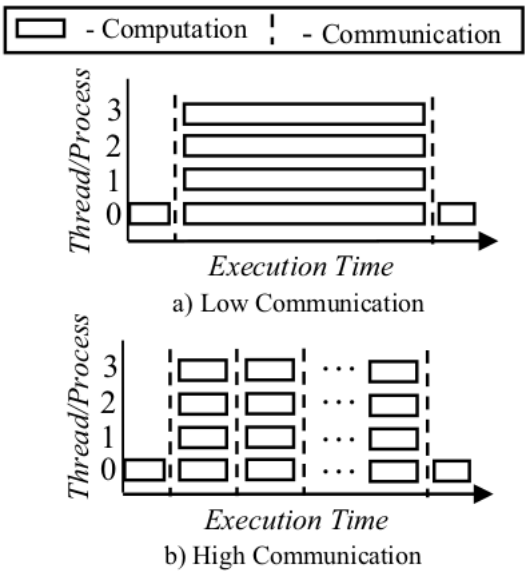
Nessa terceira fase de classificações do *benchmark*, além dos critérios de classificação anteriores, Lorenzon et al. (2015b) dividiram as aplicações de acordo com a sua demanda de comunicação. Foram consideradas a quantidade de operações de troca de dados efetuadas (*barriers*, *locks*/*unlocks* e operações de criação e finalização de *threads*) de acordo com o número de *threads*/processos utilizados. As aplicações foram divididas entre alta demanda de comunicação (*High Communication* (HC)) e baixa demanda de comunicação (*Low Communication* (LC)).

As aplicações LC realizam pouca comunicação entre *threads*/processos. Nesse caso, a comunicação ocorre na distribuição da carga de trabalho e no retorno dos resultados que é realizado no final da computação. Por outro lado, aplicações HC realizam muitas operações de comunicação para garantir uma distribuição adequada de dados entre as *threads*/processos. Esse cenário pode ser visto na Figura 11.

As aplicações utilizadas nessa fase do estudo permaneceram inalteradas. Nenhuma aplicação foi adicionada ou removida do *benchmark*. A Figura 12 apresenta esse conjunto de aplicações divididas de acordo com sua demanda de comunicação.

Nesse estudo foi considerado um critério principal que consiste na quantidade de operações para troca de dados entre *threads*/processos. Essas operações representam *barriers*, *locks*/*unlocks* e operações de criação e finalização de *threads*/processos. Foram consideradas execuções com 2, 3, 4 e 8 *threads*/processos. Esses dados foram coletados utilizando a ferramenta Pin Tool e Intel® VTune™ Amplifier XE (INTEL, 2016), que permite identificar pontos de sincronização em tempo de execução.

Figura 11 – Comportamento da comunicação das aplicações.



Fonte: Lorenzon et al. (2015b)

Figura 12 – Principais características da terceira classificação do *benchmark*.

Benchmarks		Operations exchange data (total per nº of threads/processes)				Input Size
		2	3	4	8	
LC	Calc. of the Number PI	4	6	8	16	4 billions
	Harmonic Series	8	12	16	32	100000 elem.
	Dijkstra	4	6	8	16	2048 x 2048
	Similarity of Histograms	4	6	8	16	1920 x 1080
	Matrix Multiplication	4	6	8	16	2048 x 2048
HC	Jacobi	4004	6006	8008	16016	2048 x 2048
	LU-Decomposition	8192	12288	16398	32776	2048 x 2048
	Game of Life	414	621	1079	1625	4096 x 4096
	Gram-Schmidt	3009277	4604284	6385952	12472634	2048 x 2048
	Odd-Even Sort	300004	450006	600008	1200016	150000 elem.
	Turing Ring	16000	24000	32000	64000	2048 x 2048

Fonte: Lorenzon et al. (2015b)

4.2 Definição de Critérios de Classificação

A escolha dos critérios utilizados para classificar o *benchmark* evoluíram permitindo um maior refinamento na classificação das aplicações. Até o estudo mais recente utilizando o conjunto de aplicações como um *benchmark*, possuíamos os seguintes critérios: (*i*) número de acessos à memória privada e compartilhada; (*ii*) pontos de sincronização entre *threads*/processos; (*iii*) TLP; (*iv*) dependência de dados e (*v*) operações de troca de dados.

Tabela 2 – Organização do *benchmark* atual de acordo com a comunicação entre *threads*/processos.

Comunicação	**Aplicações**	**Total de operações de troca de dados por *threads*/processos**				**Tamanho da Entrada**
		2	**3**	**4**	**8**	
Alta Comunicação (HC)	Jogo da Vida	414	621	1079	1625	4096 x 4096
	Gram-Schmidt	3009277	4604284	6385952	12472634	2048 x 2048
	Método de Jacobi	4004	6006	8008	16016	2048 x 2048
	Ordenação Par-Ímpar	300004	450006	600008	1200016	150000
	Turing Ring	16000	24000	32000	64000	2048 x 2048
Baixa Comunicação (LC)	Similaridade entre Histogramas	4	6	8	16	1920 x 1080
	Multiplicação de Matrizes	4	6	8	16	2048 x 2048
	Dijkstra	4	6	8	16	2048 x 2048
	Série Harmônica	8	12	16	32	100000
	Cálculo do Pi	4	6	8	16	4 bilhões
	Produto escalar	4	6	8	16	15 bilhões
	Integração Numérica	4	6	8	16	1 bilhão
	Transformada Discreta de Fourier	4	6	8	16	32368

Com o avançar do estudo, foram adicionadas ao *benchmark* as aplicações DFT, Integração Numérica e Produto Escalar, sendo que as duas últimas pertencem ao grupo de aplicações que não operam sobre dados estruturados (LORENZON; CERA; BECK, 2016). Em contrapartida, a aplicação Decomposição-LU foi temporariamente retirada do conjunto, pois a implementação paralela apresentou instabilidades quando utilizados números ímpares de *threads*/processos para a divisão da carga de trabalho. Após algumas modificações, essa aplicação poderá ser novamente incluída no conjunto que compõe o *benchmark*. A Tabela 2 mostra o conjunto atual de aplicações deste trabalho. Eles estão classificados de acordo com a demanda de comunicação (HC e LC).

Analisando todos os estudos anteriores, juntamente com cada classificação proposta nos diferentes trabalhos, é possível notar que as aplicações ainda estão classificadas em grandes blocos heterogêneos, similares entre si. O resultado da primeira classificação em *CPU-Bound* e *Memory-Bound* assemelha-se à mais recente classificação, que é de acordo com a quantidade de operações de troca de dados entre *threads*/processos. Nesses casos, aplicações HC tendem a ser *Memory-Bound*, enquanto que aplicações LC tendem a ser *CPU-Bound*. Entretanto, considerando a segunda classificação, que incluiu *Weakly Memory-Bound*, foi possível notar que algumas aplicações LC também fazem uma maior quantia de acessos à memória.

O fato de terem sido considerados valores que representam a média entre as IPPs, não torna possível notar a forma com que a variação das IPPs influenciou na classificação proposta em cada trabalho. Algumas análises, como a proporção de uso de CPU em relação ao uso de memória de cada aplicação, por IPP, ou mesmo o impacto de cada IPP no TLP das aplicações, não podem ser feitas baseadas nos dados dos trabalhos anteriores. Dessa forma, por se tratar de um *benchmark* cujo o principal objetivo é analisar o impacto das IPPs em arquiteturas multinúcleo, foram levantadas novos critérios de classificação,

baseados na variação das IPPs, para refinar ainda mais o conjunto de aplicações. Esses critérios são: TLP(por IPP), demanda de CPU e demanda de memória.

4.2.1 *Thread-Level Parallelism*

A métrica TLP é utilizada conforme definida pelos autores em (BLAKE et al., 2010) e (GAO et al., 2014). Ela é usada para medir o nível de concorrência da aplicação e a taxa de utilização dos processadores durante a execução da aplicação. Quanto mais perto o valor é do número de *threads*, maior é o nível de exploração do TLP fornecido pela aplicação. Por exemplo, um TLP de 4 para 4 *threads* significa que durante a execução da aplicação, as quatro *threads* estiveram executando concorrentemente durante 100% do tempo. O modelo de comunicação de cada IPP, assim como a forma de implementação de cada aplicação paralela, pode impactar diretamente no TLP, isso pode ser visto nos trabalhos anteriores, onde aplicações *CPU-Bound* possuem TLP ligeiramente superior às *Memory-Bound*.

$$TLP = \frac{\sum_{i=1}^{n} c_i i}{1 - c_0} \tag{4.1}$$

A Equação 4.1 mostra como o valor de TLP é obtido, onde c_i é a fração de tempo que i núcleos estão executando concorrentemente i *threads*, n é o número de núcleos, e c_0 é a fração de tempo ocioso do sistema (nenhuma *thread* está executando a aplicação). O tempo de execução pode ser obtido através da divisão do número de ciclos utilizados pela *thread* mais lenta pela frequência do processador. Para obter os dados direto do contador de *hardware*, foi utilizado o Intel® Parallel Studio XE (INTEL, 2015), que possui ferramentas como o Intel® VTune™ Amplifier XE que capturam esses dados e calcula o TLP automaticamente para cada aplicação.

4.2.2 Demanda de CPU e de Memória

A demanda de CPU juntamente com a demanda de memória são critérios avaliados em conjunto. Nos estudos anteriores, as aplicações estavam classificadas entre *CPU-Bound* e *Memory-Bound* de acordo com a quantidade de acessos de leitura/escrita à memória compartilhada. Porém esse tipo de dado não indica o quanto de CPU foi realmente utilizado por determinada aplicação. Uma aplicação que realiza muitos acessos à memória compartilhada também poderia ter um alto uso de CPU. No caso oposto, uma aplicação com poucos acessos à memória e previamente classificada como *CPU-Bound*, também poderia fazer menor uso de CPU em relação à outra aplicação classificada como *Memory-Bound*.

Analisando esses possíveis cenários chegou-se à conclusão de que essa classificação entre *CPU/Memory-Bound* precisa ser melhor investigada. Os dados coletados nos estudos anteriores também representam uma média entre as IPPs nesse caso. Portanto, é feita uma análise por IPP mostrando o impacto de cada uma no uso de CPU e memória.

Para obter os dados direto do contador de *hardware*, foi utilizada a ferramenta Intel® VTune™ Amplifier XE que captura esses dados em tempo de execução e estima automaticamente a porcentagem de uso de CPU e de uso de memória para cada aplicação, considerando as características de cada IPP.

4.3 Balanço do Capítulo

Neste capítulo apresentamos todo o histórico de classificações das aplicações. Inicialmente o *benchmark* era classificado apenas entre aplicações *CPU-Bound* e *Memory-Bound*. Em estudos posteriores essa classificação refinou-se mais, onde as aplicações passaram a ser classificadas também em *Weakly Memory-Bound*. Após essa fase, os estudos seguintes classificaram essas aplicações de acordo com a comunicação entre as *threads*/processos dessas aplicações. Essa nova classificação identificou aplicações com um alto uso de comunicação (HC) e aplicações com um baixo uso de comunicação (LC).

Essas classificações ocorreram de acordo com dados obtidos seguindo alguns critérios considerados relevantes para o contexto desse estudo. Em um primeiro momento foram considerados para cada aplicação: o total de acesso à endereços de memória privados e compartilhados, tanto para leitura quanto para escrita de dados; a existência ou não de pontos de sincronização entre as *threads*/processos; a dependência de dados e, por fim, o TLP das aplicações. Em trabalhos seguintes, foi adicionada à esses critérios a taxa de comunicação de cada aplicação, divididas entre alta, média e baixa comunicação.

Na sequência, foram apresentados os novos critérios utilizados para refinar ainda mais a classificação das aplicações, tendo como ideia principal a variação das IPPs. Esses critérios incluem: TLP, uso de CPU e uso de memória. Através da ferramenta Intel® Parallel Studio XE é feita a coleta dos dados definidos por esses critérios.

No próximo capítulo são apresentados os resultados, assim como a análise dos mesmos. Também é apresentado o ambiente de execução, juntamente de uma melhor definição das ferramentas que foram utilizadas na coleta dos dados.

5 Resultados

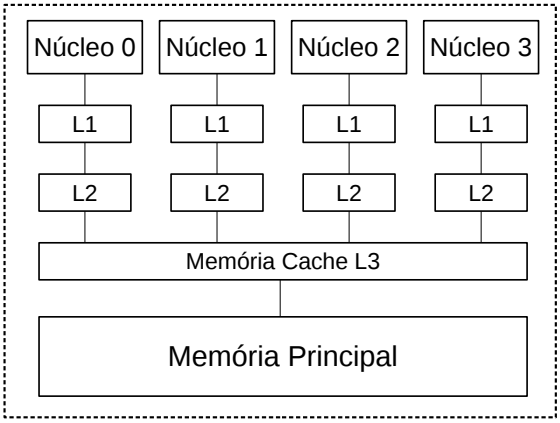
Neste capítulo apresentamos todos os resultados obtidos através das execuções e análises realizadas em cima das aplicações apresentadas. Essas execuções foram feitas em duas arquiteturas diferentes, que são apresentadas na seção 5.1, juntamente das ferramentas utilizadas para a extração dos dados. Em sequência, são apresentados na seção 5.2 os primeiros resultados, que se referem às estruturas de dados utilizadas e complexidade das aplicações. Na seção 5.3 são apresentados os resultados de TLP para cada aplicação, fazendo uma comparação entre as IPPs. Por fim, a seção 5.4 mostra os resultados de uso de CPU e memória de cada aplicação para cada IPP.

5.1 Ambiente de Execução

5.1.1 Arquiteturas Utilizadas

Os experimentos foram realizados em dois computadores. O primeiro é equipado com um processador Intel® Core™ i7-4700MQ com 4 núcleos físicos e 4 virtuais operando na frequência padrão de 2.4 GHz e com frequência turbo de 3.4 GHz para até 4 núcleos simultâneos. Seu sistema de memória é formado por três níveis de cache: 4 caches L1 de dados com 32 KB e 4 de instruções também com 32 KB para cada núcleo; 4 caches L2 de 256 KB igualmente para cada núcleo e 6 MB de cache L3 compartilhada entre todos os núcleos. A organização hierárquica dessa arquitetura é representada na Figura 13. A memória principal (RAM) tem tamanho de 8 GB e tecnologia DDR3. O sistema operacional é o Linux Ubuntu 16.04.4 com o compilador GCC 5.4.0.

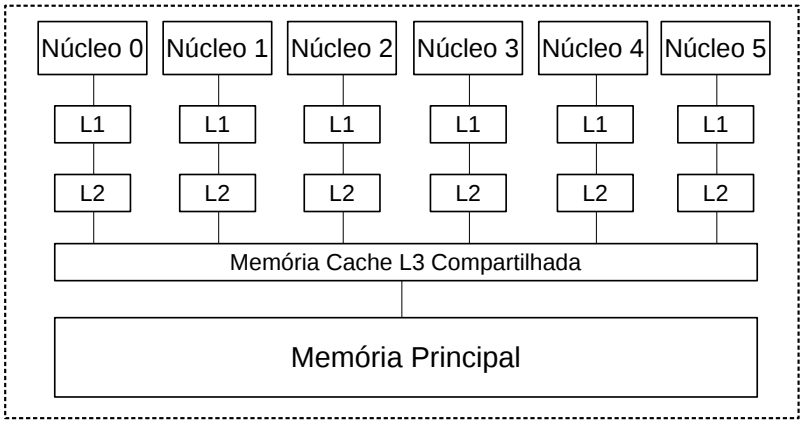
Figura 13 – Organização hierárquica de memória do Intel® Core™ i7-4700QM.



Fonte: O próprio autor.

O segundo computador é equipado com um processador Intel® Xeon® E5-2609 v3 com 6 núcleos físicos operando na frequência de 1.9 GHz. Seu sistema de memória é formado por três níveis de cache: 6 caches L1 de dados com 32 KB e 6 de instruções também com 32 KB para cada núcleo; 6 caches L2 de 256 KB igualmente para cada núcleo e 15 MB de cache L3 compartilhada entre todos os núcleos. A organização hierárquica dessa arquitetura é representada na Figura 14. A memória principal (RAM) tem tamanho de 16 GB e tecnologia DDR4. O sistema operacional é o Linux Ubuntu 14.04.5 com o compilador GCC 4.8.4. Em ambas arquiteturas foi utilizado o OpenMPI versão 1.10.2 e OpenMP versão 4.5.

Figura 14 – Organização hierárquica de memória do Intel® Xeon® E5-2609 v3.



Fonte: O próprio autor.

5.1.2 Extração dos Dados

5.1.2.1 Ferramentas Utilizadas

A análise das aplicações e das IPPs foi realizada através de dados obtidos com o Intel® VTune™ Amplifier 2017, que é uma ferramenta para a análise de desempenho de software para arquiteturas de 32 e 64-bit. Ele pode ser utilizado através de interface gráfica e linha de comando em sistemas operacionais Linux ou Microsoft Windows e faz parte do conjunto de ferramentas para análise e desenvolvimento de programas paralelos da Intel, chamado de Parallel Studio.

O Intel® Parallel Studio é um conjunto de ferramentas de desenvolvimento de software criado pela Intel. Ele facilita o desenvolvimento de código nativo em Windows e Linux em C, C++ e Fortran para computação paralela. Além de incluir o VTune, esse conjunto é composto também pelo Intel® Advisor, Intel® Inspector, além de disponibilizar os compiladores e bibliotecas de compilação da Intel.

5.1.2.2 Coleta dos Dados

Cada tipo de coleta precisa ser regulada de acordo com a IPP utilizada. Para a análise de aplicações com PThreads, podem ser utilizadas as configurações padrão do software. Para coleta de dados de aplicações em OpenMP, algumas opções específicas para essa IPP devem ser utilizadas. Já a coleta de dados com MPI só pode ser realizada através da linha de comando. Nas demais IPPs o número de threads pode ser passado como parâmetro em um campo específico. No entanto, a implementação das aplicações em MPI não permite definir o número de processos dessa forma. Assim, as execuções em MPI foram feitas através da linha de comando, sendo que o resultado salvo pode ser visualizado na interface gráfica do VTune posteriormente.

Para este trabalho, foram coletados dados de TLP e uso de CPU e memória. O TLP é calculado automaticamente pelo VTune, a métrica chamada *Average CPU Usage* utiliza o cálculo da Equação 4.1 visto anteriormente para computar o TLP. Os dados de uso de CPU e memória foram obtidos através da opção de coleta chamada *HPC-Performance*.

Para o TLP as aplicações foram executadas com 2 e com 4 *threads*/processos com o processador Intel® Xeon® E5-2609 v3. Apesar da possibilidade de se executar com até 6 *threads*/processos nessa arquitetura, as aplicações estão melhor otimizadas para divisão da carga de trabalho de acordo com uma progressão geométrica de razão 2.

A coleta do uso de CPU e memória foi realizada utilizando o processador Intel® Core™ i7-4700MQ, pois o processador Xeon E5-2609 v3 não possuí suporte e nem os *drivers* necessários para a coleta de dados de memória exigidos pelo VTune. Foram realizados alguns testes onde foi observado que a variação no número de *threads*/processos não afetava o resultado final de cada aplicação, portando para os dados de uso de CPU e memória decidiu-se utilizar apenas 4 *threads*/processos.

Os resultados apresentados neste capítulo são a média de 10 execuções desconsiderando os valores extremos. Para todos os cenários o desvio padrão foi abaixo de 1% o que justifica o baixo número de execuções. A Tabela 3 mostra o tamanho das entradas utilizadas para cada aplicação, assim como a respectiva sigla utilizada para identificar cada aplicação nas seções seguintes.

5.1.2.3 Problemas Encontrados

Diversos problemas e desafios tiveram que ser resolvidos ou contornados para alcançar os objetivos deste trabalho. A falta de documentação das aplicações dificultou muito a análise de seus algoritmos, equações envolvidas e complexidades.

Para a execução das aplicações, não havia nenhum tipo de instrução em relação à parâmetros de execução, identificação dos argumentos de entrada, comandos de compila-

Tabela 3 – Tamanho das entradas das aplicações.

Aplicações	Siglas	Tamanho das Entradas
Cálculo do PI	PI	4 bilhões de iterações
Integração Numérica	IN	1 bilhão de iterações
Produto Escalar	PE	15 bilhões de iterações
Ordenação Par-Ímpar	OE	Vetor com 150000 elementos
Série Harmônica	HS	Vetor com 100000 elementos
Transf. Discreta de Fourier	DFT	Vetor com 32768 elementos
Jogo da vida	GL	Matriz 2048 × 2048
Gram–Schmidt	GS	Matriz 2048 × 2048
Método de Jacobi	JA	Matriz 2048 × 2048
Turing ring	TR	Matriz 2048 × 2048
Dijkstra	DJ	Matriz 2048 × 2048
Multiplicação de Matrizes	MM	Matriz 2048 × 2048
Similaridade de Histogramas	SH	Matriz 1920 × 1080

ção e comandos de execução com as diferentes IPPs. Além disso, os códigos apresentavam alguns problemas em relação a variáveis e argumentos de entrada.

A primeira ferramenta ser explorada foi o Pin Tool da Intel. Com essa ferramenta pretendia-se coletar *trace* de memória, para poder quantificar o uso de memória por *threads*/processos. Entretanto os *trace* gerado por essa ferramenta era muito grande, impossibilitando a análise dos dados. Um *parser* em C foi utilizado na tentativa de tratar esses dados, porém para analisar o *trace* de um programa com uma matriz pequena de 128×128 o demorava cerca de 1 hora. Dessa forma, após muitas tentativas, impossibilitou-se o uso dessa ferramenta para esse propósito.

Além dos problemas com as aplicações e com as ferramentas, complicações nas arquiteturas utilizadas atrasaram as análises diversas vezes. Inicialmente, pretendia-se executar os testes em uma *workstation* com processador de 32 núcleos. Os resultados iniciais foram perdidos devido à uma falha nos discos rígidos e a máquina permaneceu fora de funcionamento até a conclusão deste trabalho. Além disso, a outra *workstation*, onde os dados de TLP foram coletados, não oferece suporte do processador para análises de uso de memoria. Então novamente precisou-se refazer parte do trabalho em outra máquina.

Com as coletas utilizando o Vtune surgiram mais problemas. O comando `mpirun` do pacote libopenmpi instalado na máquina não retornava dados com o Vtune. Após algum tempo descobriu-se que devia ser usado o `mpirun` disponibilizado pelo próprio Parallel Studio. Além disso, os programas em MPI-1 compilados com o `mpicc` do pacote libopenmpi retornavam dados inconsistentes. Novamente foi necessário utilizar o `mpicc` disponibilizado pelo Parallel Studio. De qualquer forma, todos os desafios foram superados ou contornados.

5.2 Complexidade das Aplicações

Neste trabalho foi feita uma análise detalhada do código sequencial das aplicações juntamente com um estudo bibliográfico para estimar a complexidade de cada aplicação. A análise de complexidade de um algoritmo é geralmente utilizada para obter uma estimativa do tempo de execução em função do tamanho dos dados de entrada. O resultado é normalmente expresso utilizando a notação *Big O*. Isso é útil para comparar algoritmos, especialmente quando uma grande quantidade de dados deve ser processada. São necessárias estimativas mais detalhadas para a comparação de algoritmos quando a quantidade de dados é pequena, embora o tempo de execução não se torne um problema nesses casos.

Com o estudo das complexidades concluído, foi possível criar a Tabela 4, que relaciona a complexidade das aplicações com as estruturas de dados que cada uma utiliza. Nessa tabela pode-se observar que dados de entrada não estruturados estão estritamente relacionados à $O(n)$, a menor entre as complexidades das aplicações do *benchmark*. Essas três aplicações realizam somente cálculos iterativos usando um único laço em seu algoritmo.

Entre as três aplicações que utilizam vetores, encontrou-se duas complexidades distintas: $O(n * d)$ e $O(n^2)$. A aplicação Ordenação Par-Ímpar possui complexidade de $O(n * d)$ pois possui laços aninhados independentes, onde a quantidade de iterações pode variar em função de d. As outras duas aplicações possuem dois laços aninhados simples.

A maior parte das aplicações do *benchmark* computam sobre matrizes, que podem ser chamadas de estruturas de dados bidimensionais. Com essa estrutura de dados estão relacionadas as complexidades: $O(n^2)$, $O(m*n^2)$ e $O(n^3)$. Similar à Ordenação Par-Ímpar, as aplicações Jogo da Vida e Turing Ring possuem três laços aninhados, sendo que em ambos os casos um deles varia suas iterações em função de m. As demais aplicações que utilizam matrizes possuem laços aninhados simples.

Essa análise de complexidade foi feita baseada nas aplicações sequenciais. A complexidade de aplicações seriais se baseia nas operações aritméticas do algoritmo, enquanto a análise da complexidade de aplicações paralelas se baseia principalmente no tempo de execução, porém diversos outros fatores influenciam nessa análise das aplicações paralelas, como balanceamento de carga e modelo de paralelização, conforme Kruskal, Rudolph e Snir (1990) e Mikloško e Kotov (1984) explicam em seus trabalhos.

O estudo das complexidades sequenciais mostrou que o conjunto de aplicações do *benchmark* variam de $O(n)$ à $O(n^3)$, com diversas outras complexidades intermediárias. Dessa forma, é possível concluir que o *benchmark* possui diversidade o suficiente nesse aspecto para avaliar de diferentes formas o desempenho de diferentes arquiteturas.

Tabela 4 – Dados obtidos através da análise das aplicações sequenciais.

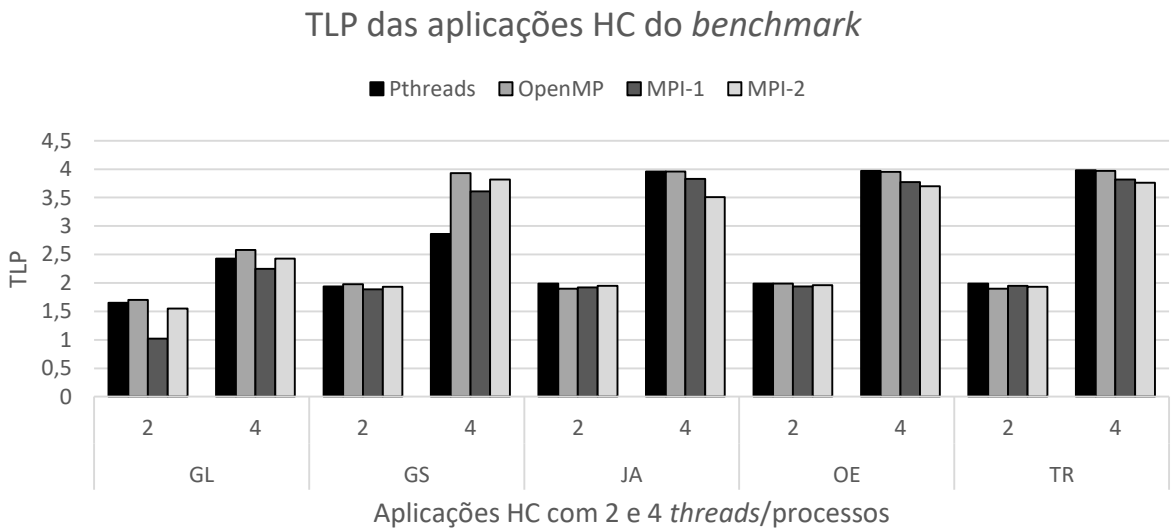
Estruturas de dados	**Aplicações**	**Complexidades**
Dados não estruturados	Cálculo do Pi Produto Escalar Integração Numérica	$O(n)$
Vetores	Série Harmônica	$O(n * d)$
	Ordenação Par-Ímpar Transf. Discreta de Fourier	$O(n^2)$
Matrizes	Similaridade entre Histogramas	$O(n^2)$
	Jogo da Vida Turing Ring	$O(m * n^2)$
	Dijkstra Método de Jacobi Multiplicação de Matrizes Gram-Schmidt	$O(n^3)$

5.3 TLP

Nesta seção são apresentados os resultados de TLP das aplicações. Os gráficos estão organizados em dois grupos, de acordo com a quantidade de comunicação das aplicações (HC e LC). Na Figura 15 são exibidos os resultados de TLP para as aplicações HC do *benchmark*, variando as IPPs e a quantidade de *threads*/processos entre 2 e 4.

Entre as 5 aplicações HC do *benchmark*, 3 delas tiveram um comportamento se-

Figura 15 – Gráfico com o TLP das aplicações HC

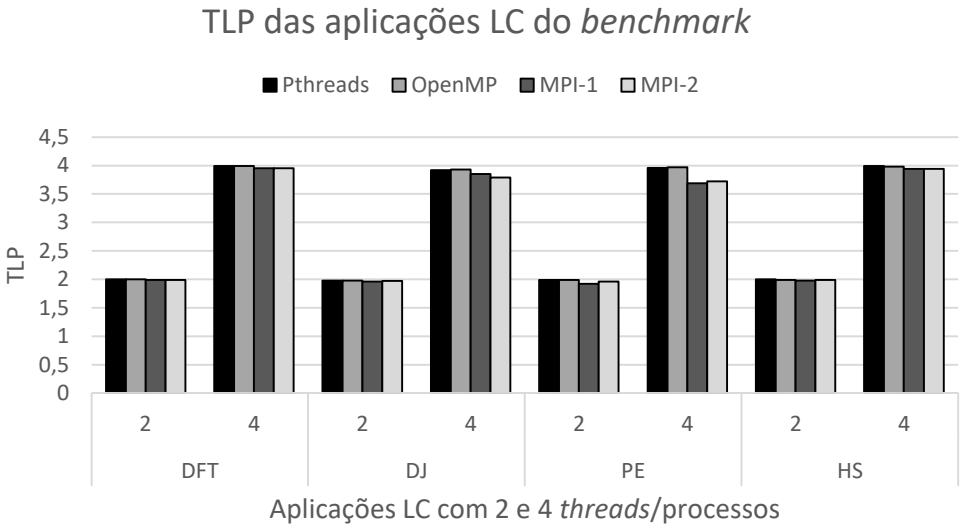


melhante, sendo elas: JA, OE e TR. Considerando esses três casos, o TLP com 4 *threads*/processos seguiu o mesmo padrão, sendo praticamente idêntico entre OE e TR, com MPI-2 tendo uma pequena queda no TLP na aplicação JA. Considerando as execuções dessas 3 aplicações com 2 *threads*/processos é possível notar que, embora PThreads tenha um TLP mais alto em todos os casos, o segundo TLP mais alto se alterna entre OpenMP, MPI-1 e MPI-2.

Nas duas primeiras aplicações (GL e GS), ocorre um comportamento semelhante das IPPs entre elas, porém com padrão distinto do padrão das demais aplicações. Nessas duas aplicações, a paralelização com OpenMP atingiu um melhor TLP em todos os casos. Já PThreads, que se destacou positivamente com JA, OE e TR, teve TLP abaixo de 3 executando GS com 4 *threads*, ficando cerca de 1 ponto abaixo das demais IPPs. Nos demais casos, MPI-1 mostrou TLP inferior às demais IPPs.

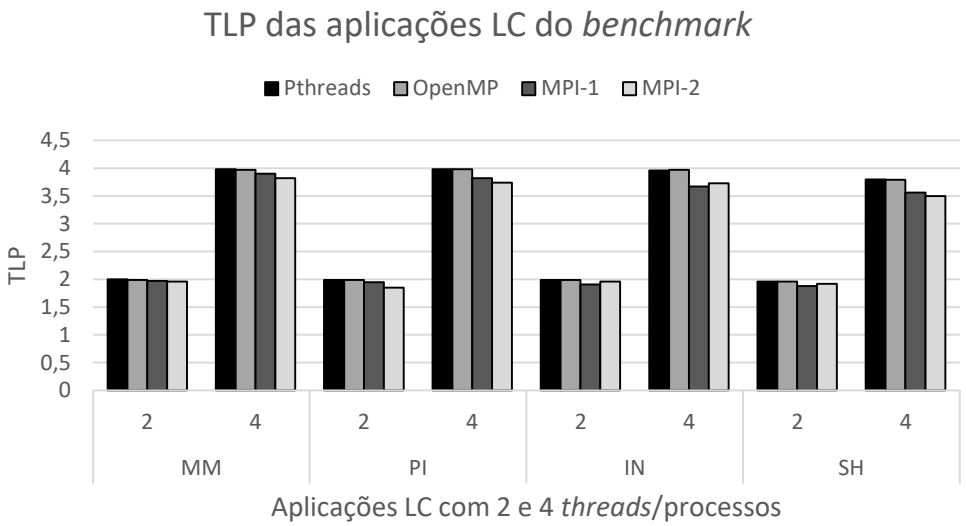
Entre todas as aplicações HC analisadas, GL teve menor TLP em todos os casos. O TLP dessa aplicação foi o menor em relação às outras aplicações em todos os casos analisados, sendo que na execução com 4 *threads*/processos o TLP foi quase metade do ideal. Nas demais aplicações, o TLP ficou próximo do ideal (2 e 4 para 2 e 4 *threads*/processos, respectivamente).

Figura 16 – Gráfico com o TLP das aplicações LC



Na Figura 16 e Figura 17, são exibidos os resultados de TLP para as aplicações LC, sendo elas: DFT, DJ, PE, HS, MM, PI, IN, SH. No geral, o TLP dessas aplicações ficou próximo do ideal para a maioria dos casos. Um comportamento semelhante do MPI 1 e 2 pode ser observado nas aplicações PE, PI e IN nas execuções com 4 processos. Nos três casos, o desempenho dessas IPPs ficou inferior em relação aos demais casos.

Figura 17 – Gráfico com o TLP das aplicações LC



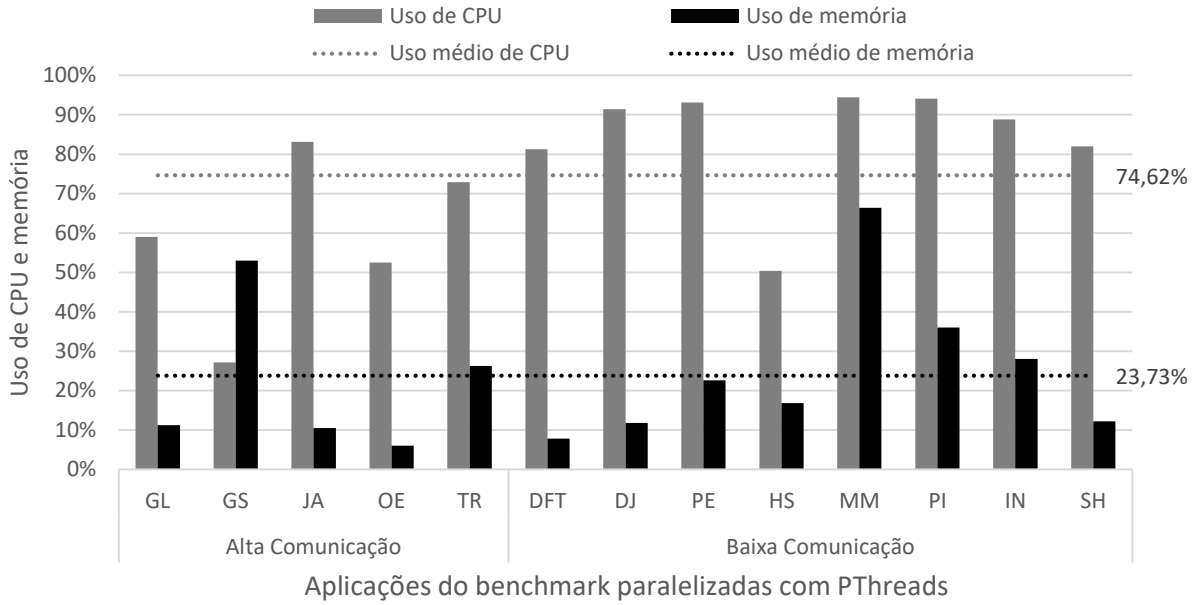
O comportamento dessas IPPs ocorre na execução com dois processos dessas aplicações também, porém de forma menos acentuada.

No geral, PThreads e OpenMP obtiveram TLP mais alto em todos os casos de aplicações LC analisados, seguido de MPI-1 e MPI-2. Esse resultado evidencia o impacto de diferentes IPPs no grau de paralelismo. Entre essas aplicações, SH possui o menor TLP para todas IPPs analisadas. O restante das aplicações possuem TLP próximo do ideal, o que indica que o grau de paralelismo explorado por essas aplicações é alto o suficiente para que componham um *benchmark* com o propósito de analisar o desempenho e consumo de energia de arquiteturas multinúcleo.

5.4 Uso de CPU e Memória por IPP

Esta seção são apresentados os resultados de uso de CPU e uso de memória das aplicações do *benchmark*. Inicialmente os resultados são apresentados individualmente por IPP. A estrutura da apresentação desses resultados nas subseções seguintes seguirá o mesmo formato para cada IPP: o primeiro gráfico exibe o uso de CPU e memória, dividindo as aplicações entre Alta Comunicação (HC) e Baixa Comunicação (LC); o segundo gráfico exibe essas informações em ordem decrescente de uso de CPU; o terceiro gráfico exibe as informações em ordem decrescente de uso de memória. Os gráficos também incluem a média do uso de CPU e memória. Essa média é uma média simples, calculada considerando os valores todas as aplicações.

Figura 18 – Gráfico do uso de CPU e memória por aplicação utilizando PThreads.



5.4.1 PThreads

Conforme a Figura 18 apresenta, as aplicações em PThreads apresentam uma grande distribuição no uso de CPU e memória. Em relação ao uso de CPU, ele varia de cerca de 95% com MM a menos de 30% com GS. Sendo GS a única aplicação que fez mais uso de memória do que CPU, sendo essa diferença aproximadamente o dobro. Embora MM faça o maior uso de CPU, essa aplicação também faz o maior uso de memória entre as demais aplicações, com mais de 65% de uso.

A distribuição do uso de memória em PThreads em relação ao uso de CPU, mostra-se bastante diversificada, com aplicações possuindo alto e baixo uso de memória e também alto, médio e baixo uso de CPU. Nessa IPP, o uso de CPU foi em média cerca de 50% maior que o uso de memória das aplicações.

A Tabela 5 mostra uma relação das aplicações paralelizadas com PThreads divididas em relação à média do uso de CPU e memória. Um alto uso significa um uso acima da média, assim como um baixo uso significa um uso abaixo da média.

Analisando os gráficos ordenados por CPU (Figura 19) e memória (Figura 20), é possível observar que em PThreads a maioria das aplicações estão acima da média em relação ao uso de CPU e abaixo da média em relação ao uso de memória. Entretanto, a tabela mostra que existem aplicações com: alto uso de CPU e alto uso de memória (MM, PI e IN); alto uso de CPU e baixo uso de memória (JA, DFT, DJ, PE e SH); baixo uso de CPU e baixo uso de memória (GL, OE e HS); e baixo uso de CPU e alto uso de memória

Figura 19 – Gráfico do uso de CPU e memória por aplicação utilizando PThreads.

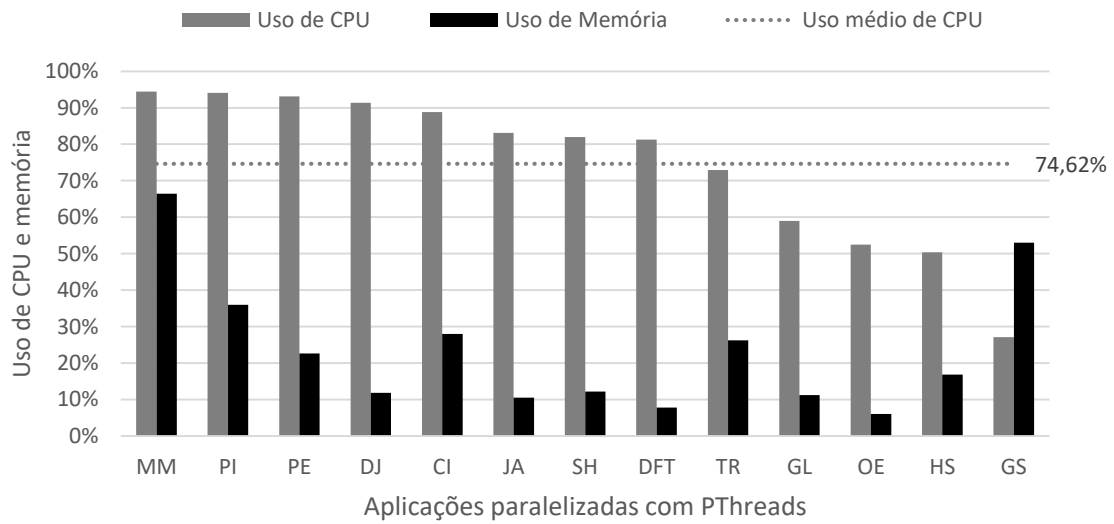


Figura 20 – Gráfico do uso de CPU e memória por aplicação utilizando PThreads.

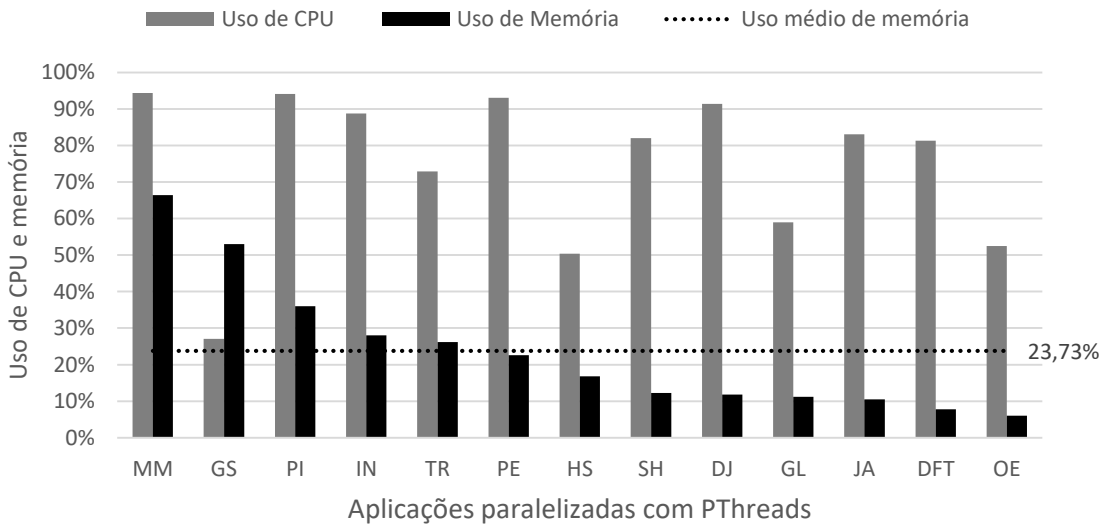


Tabela 5 – Visão geral do uso de CPU e memória das aplicações com PThreads.

Uso de CPU	Aplicação	Uso de memória
Alto	MM PI IN	Alto
	JA DFT DJ PE SH	Baixo
Baixo	GL OE HS	
	TR GS	Alto

(TR e GS). Portanto pode-se concluir que o conjunto de aplicações implementadas com PThreads possui uma boa distribuição de aplicações com alto e baixo uso de CPU e memória.

5.4.2 OpenMP

Figura 21 – Gráfico do uso de CPU e memória por aplicação utilizando OpenMP.

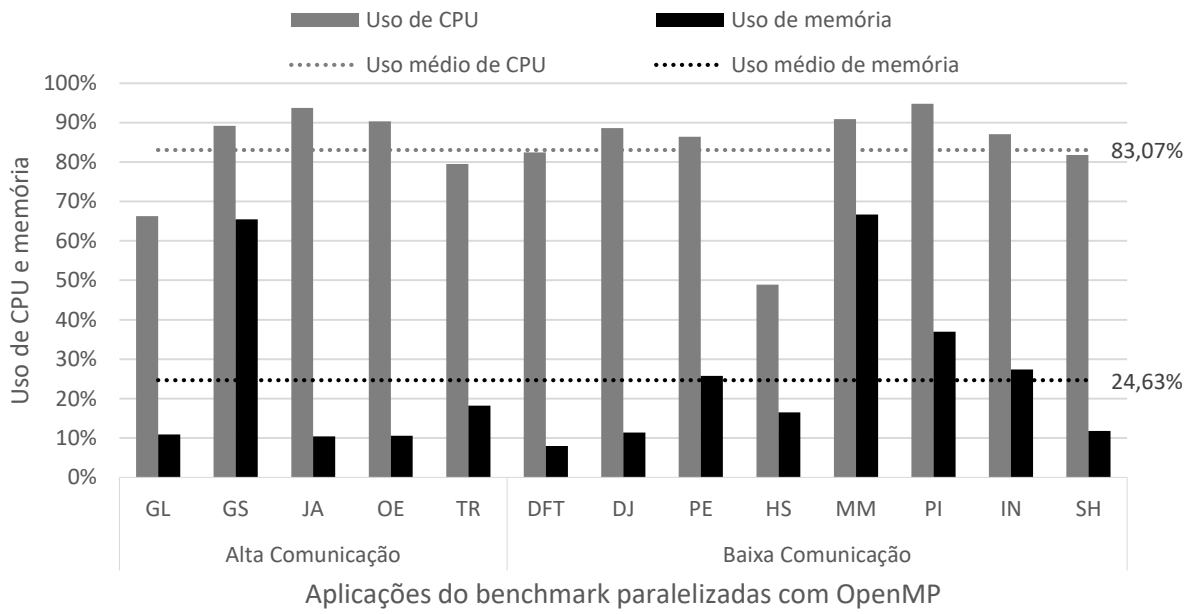


Figura 22 – Gráfico do uso de CPU e memória por aplicação utilizando OpenMP.

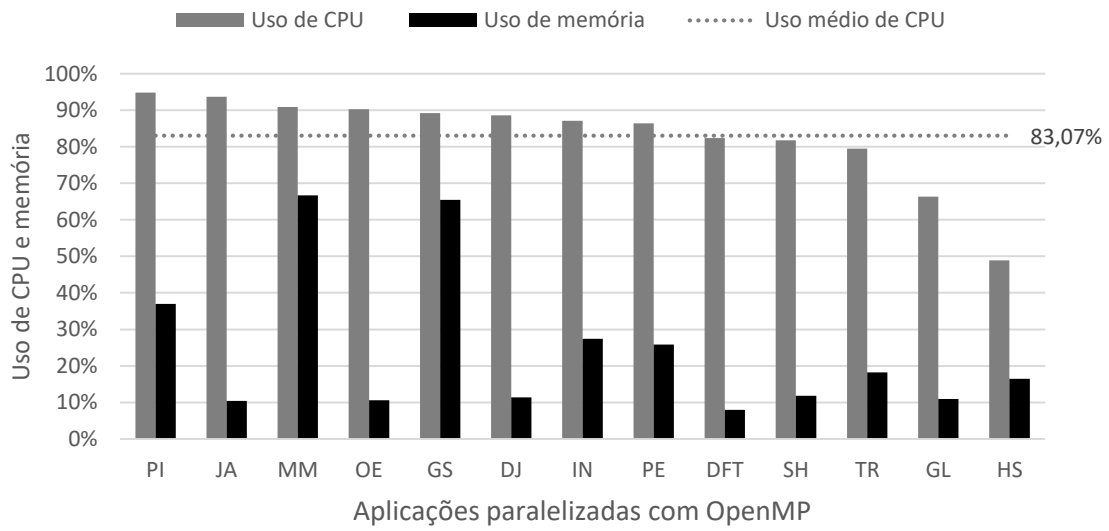
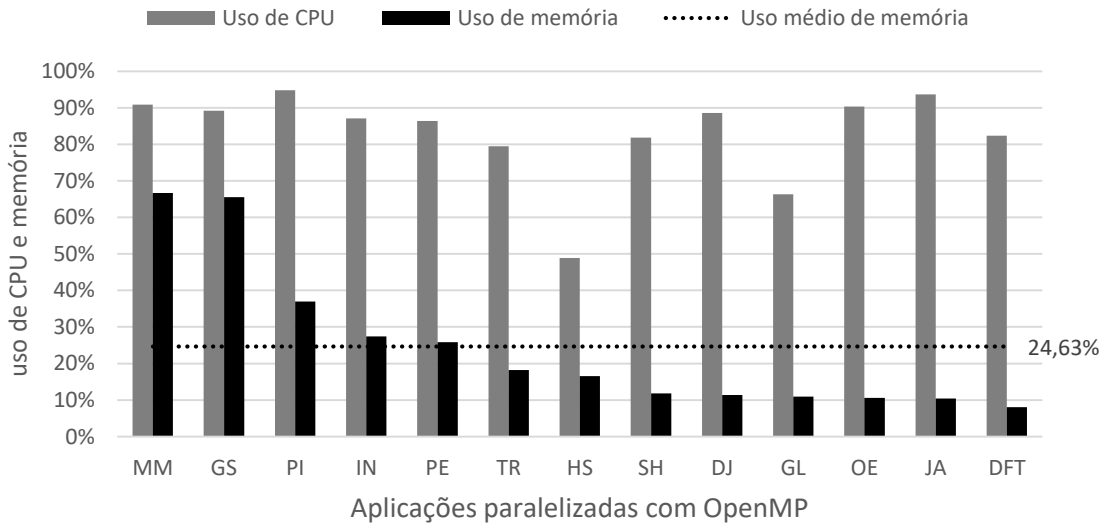


Figura 23 – Gráfico do uso de CPU e memória por aplicação utilizando OpenMP.



As aplicações em OpenMP também apresentam uma grande distribuição no uso de CPU e memória, conforme pode observado na Figura 21. Em relação ao uso de CPU, ele varia de cerca de 95% com PI a menos de 50% com HS. Ao contrário do cenário visto em PThreads, nesta IPP nenhuma aplicação teve uso de memória superior ao uso de CPU. Assim como em PThreads, em OpenMP, MM também faz o maior uso de memória entre as demais aplicações, com mais de 65% de uso, porém não faz o maior uso de CPU.

A distribuição do uso de memória em OpenMP em relação ao uso de CPU, mostra-se bastante semelhante ao cenário visto em PThreads, com aplicações possuindo alto, médio e baixo uso de memória e também alto, médio e baixo uso de CPU. Nessa IPP, o

Tabela 6 – Visão geral do uso de CPU e memória das aplicações com OpenMP.

Uso de CPU	Aplicação	Uso de memória
Alto	PI PE MM IN GS	Alto
	DJ OE JA	Baixo
Baixo	DFT SH TR GL HS	

uso de CPU foi em média cerca de 60% maior que o uso de memória das aplicações, cerca de 10% a mais em relação ao resultado visto em PThreads.

A Tabela 6 mostra uma relação das aplicações paralelizadas com OpenMP divididas em relação à média do uso de CPU e memória. Analisando os gráficos ordenados por CPU (Figura 22) e memória (Figura 23), é possível observar que em OpenMP a maioria das aplicações estão acima da média em relação ao uso de CPU e abaixo da média em relação ao uso de memória. O mesmo cenário visto em PThreads. Entretanto, a tabela mostra que não existem aplicações em OpenMP com baixo uso de CPU e alto uso de memória, o que evidencia que a escolha da interface de programação interfere na utilização de CPU e memória.

A tabela ficou organizada da seguinte forma: aplicações com alto uso de CPU e alto uso de memória (PI, PE, MM, IN e GS); alto uso de CPU e baixo uso de memória (DJ, OE e JA); e baixo uso de CPU e baixo uso de memória (DFT, SH, TR, GL e HS).

Pode-se concluir que o conjunto de aplicações implementadas com OpenMP também possui uma boa distribuição de aplicações com alto e baixo uso de CPU e memória. Entretanto, nenhuma aplicação fez mais uso de memória em relação à CPU com essa IPP. Dessa forma, talvez seja necessário investigar uma aplicação que atenda à essa demanda.

5.4.3 MPI-1

De acordo com o gráfico da Figura 24, observa-se que as aplicações em MPI-1 apresentam uma grande distribuição no uso de memória e pouca variação em relação ao uso de CPU. Excetuando-se as aplicações GL e JA, que possuem uso de CPU abaixo de 30%, todas as demais aplicações fazem uso de CPU próximo à média. Em compensação

Figura 24 – Gráfico do uso de CPU e memória por aplicação utilizando MPI-1.

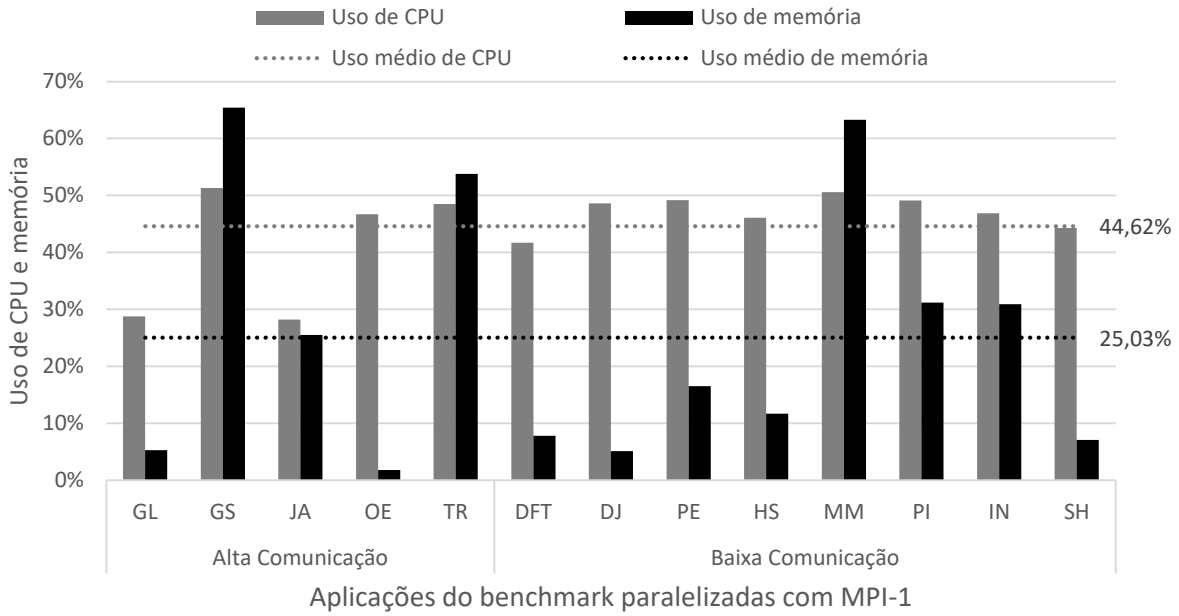
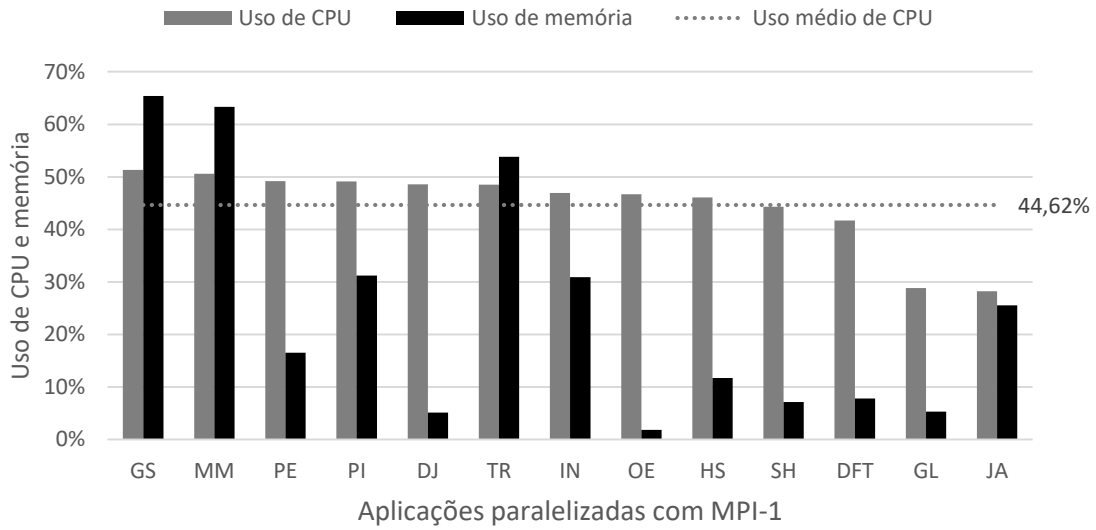


Figura 25 – Gráfico do uso de CPU e memória por aplicação utilizando MPI-1.



à lacuna de aplicação com uso de memória maior que uso de CPU visto em OpenMP, em acmpi-1 três aplicações preenchem essa lacuna (GS, MM e TR). A aplicação GS possui tanto o maior uso de CPU, quanto o maior uso de memória nessa IPP.

A distribuição do uso de memória em MPI-1 não se assemelha aos cenários vistos anteriormente, onde as aplicações possuíam alto, médio e baixo uso de CPU. Nessa IPP, o uso de CPU é mais uniforme entre as aplicações. Isso representa cerca de apenas 20% a

Figura 26 – Gráfico do uso de CPU e memória por aplicação utilizando MPI-1.

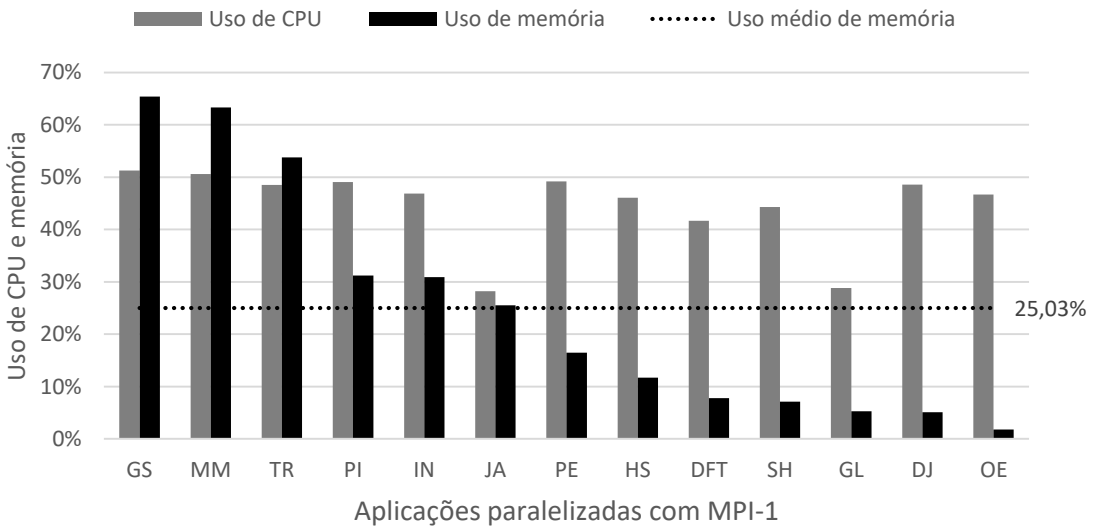


Tabela 7 – Visão geral do uso de CPU e memória das aplicações com MPI-1.

Uso de CPU	**Aplicação**	**Uso de memória**
Alto	PE DJ OE HS	Baixo
	PI MM IN GS TR	Alto
Baixo	JA	
	SH GL DFT	Baixo

mais que o uso de memória das aplicações, contra os 60% de diferença visto em OpenMP.

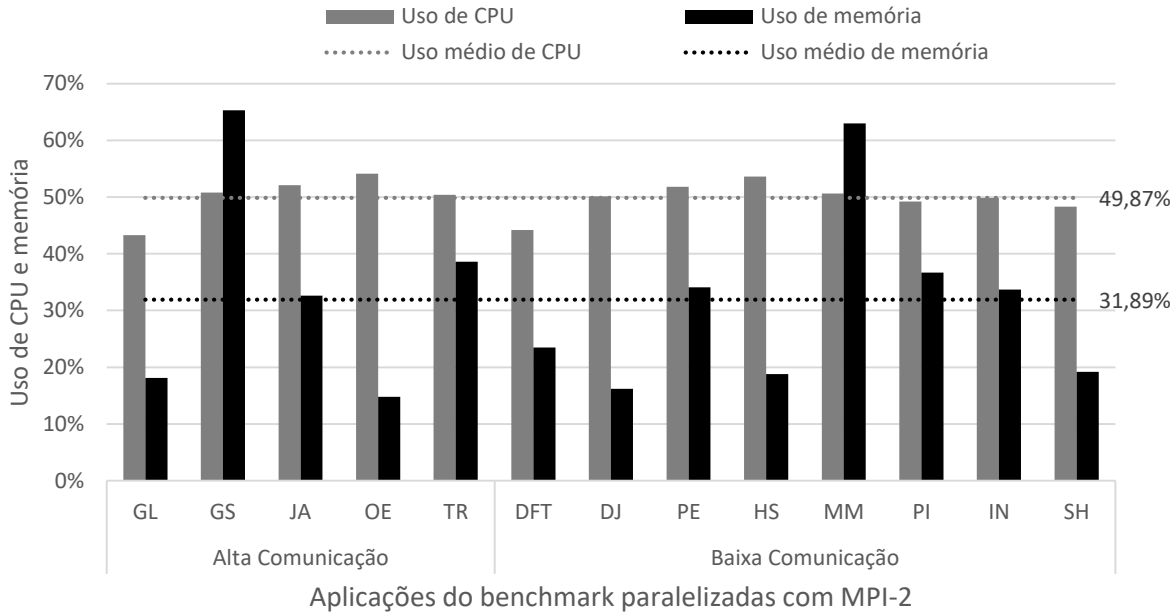
A Tabela 7 mostra uma relação das aplicações paralelizadas com MPI-1 divididas em relação à média do uso de CPU e memória. Analisando os gráficos ordenados por CPU (Figura 25) e memória (Figura 26), é possível observar que em MPI-1 a maioria das aplicações estão acima da média em relação ao uso de CPU, porém apresenta uma maior variação e um aumento no uso de memória em relação às IPPs vistas anteriormente. A tabela mostra que todas as combinações foram preenchidas com MPI-1. A tabela ficou organizada da seguinte forma: aplicações com alto uso de CPU e alto uso de memória (PI, MM, IN, GS e TR); alto uso de CPU e baixo uso de memória (PE, DJ, OE e HS); baixo

uso de CPU e alto uso de memória (JA); e baixo uso de CPU e baixo uso de memória (SH, GL e DFT).

Pode-se concluir que o conjunto de aplicações implementadas com MPI-1 também possui uma boa distribuição de aplicações com alto e baixo uso de memória. Entretanto, ocorre pouca variação em relação ao uso de CPU. De modo geral, as aplicações possuem uma grande variação no uso de memória em relação à média e uma boa variação considerando o uso de CPU e memória de cada aplicação.

5.4.4 MPI-2

Figura 27 – Gráfico do uso de CPU e memória por aplicação utilizando MPI-2.



As aplicações em MPI-2 apresentam resultados muitos semelhantes aos observados com MPI-1, onde há uma grande distribuição no uso de memória e pouca variação em relação ao uso de CPU. Todas as aplicações fazem uso de CPU próximo à média (aproximadamente 50%), conforme pode ser visto naFigura 27. Com relação às aplicações com uso de memória superior ao uso de CPU, duas aplicações se encaixam nessa categoria: GS e MM.

A distribuição do uso de CPU em MPI-2 é similar ao visto em MPI-1, porém é ainda mais uniforme, com cerca de apenas 10% de variação entre o menor e o maior uso de memória. A diferença entre a média de uso de CPU em relação ao uso de memória representa menos 20% nessa IPP, valor similar ao observado em MPI-1.

Figura 28 – Gráfico do uso de CPU e memória por aplicação utilizando MPI-2.

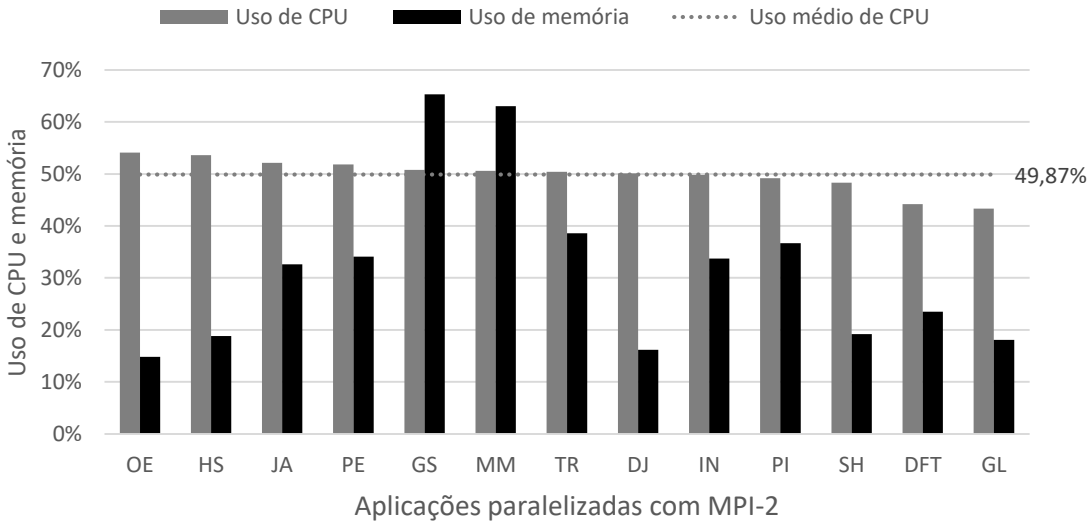
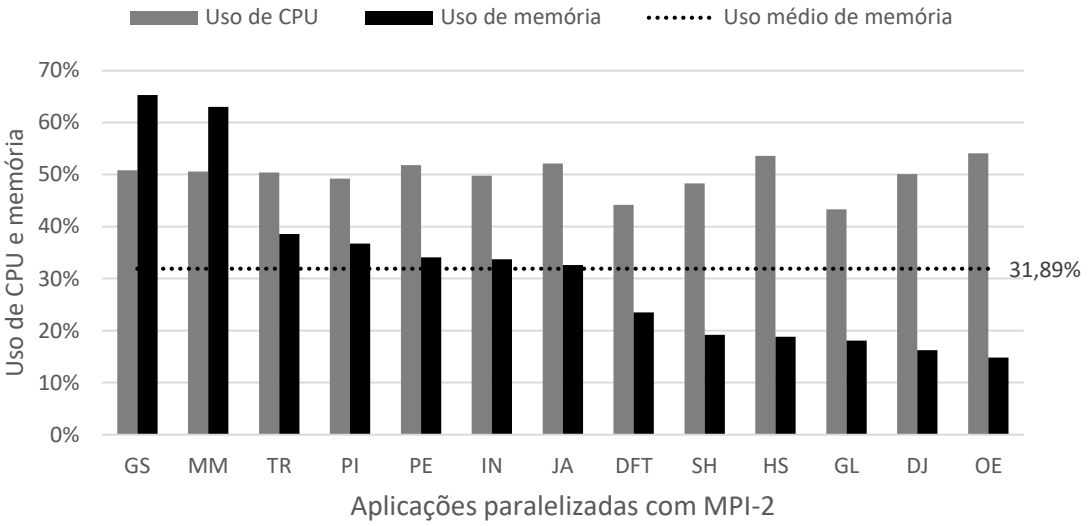


Figura 29 – Gráfico do uso de CPU e memória por aplicação utilizando MPI-2.



A Tabela 8 mostra uma relação das aplicações paralelizadas com OpenMP divididas em relação à média do uso de CPU e memória. Analisando os gráficos ordenados por CPU (Figura 28) e memória (Figura 29), nota-se que em MPI-2 as aplicações possuem uma distribuição mais uniforme no uso de CPU em relação às demais IPPs, apresentando variação no uso de memória mais acentuado, semelhante à MPI-1. A tabela mostra que com MPI-2 todas as combinações também foram preenchidas, da mesma forma observada em PThreads e MPI-1. A tabela ficou organizada da seguinte forma: aplicações com alto uso de CPU e alto uso de memória (MM, PI e IN); alto uso de CPU e baixo uso de memória (JA, SH, PE, DJ e DFT); baixo uso de CPU e alto uso de memória (TR e GS);

Tabela 8 – Visão geral do uso de CPU e memória das aplicações com MPI-2.

Uso de CPU	Sigla	Uso de memória
Alto	JA PE GS MM TR	Alto
	OE HS DJ	Baixo
Baixo	SH DFT GL	
	IN PI	Alto

e baixo uso de CPU e baixo uso de memória (GL, OE e HS).

Pode-se concluir que o conjunto de aplicações implementadas com MPI-2 também possui uma boa distribuição de aplicações com alto e baixo uso de memória de forma similar à MPI-1. Também é a IPP que apresenta a menor variação em relação ao uso de CPU. Entretanto, considerando as médias de uso de CPU e memória, conclui-se que as aplicações possuem variação o suficiente para compor o *benchmark*.

Analisando as tabelas das 4 IPPs em conjunto, foi possível identificar alguns padrões. Entre as aplicações com alto uso de CPU e alto uso de memória, as aplicações MM, PI e IN ficaram nessa mesma categoria em todas as 4 IPPs. Já a aplicação DJ, em todas as IPPs ficou na categoria de alto uso de CPU e baixo uso de memória. Por fim, a aplicação GL apresentou um baixo uso de CPU e baixo uso de memória em todos os casos.

Como forma de entender essas aplicações que ficaram nos extremos de alto e baixo uso de CPU, foram coletados dados por *thread* para as aplicações MM e GL. A Figura 31 ilustra o comportamento entre as *threads* da aplicação MM, que faz um alto uso de CPU. Já a Figura 30 mostra o comportamento das *threads* da aplicação GL, com baixo uso de CPU.

Conforme é possível observar, o comportamento entre as *threads* de ambas as aplicações diferem muito. Com MM, assim que a carga de trabalho é divida, é utilizado praticamente 100% da CPU, até o final da computação. Por outro lado, as 4 *threads* da aplicação GL em nenhum momento utilizam 100% da CPU simultaneamente de forma contínua. Isso também impacta diretamente no TLP dessa aplicação, pois indica que em determinados trechos não é explorado todo o paralelismo que a arquitetura permite. Com

Figura 30 – Uso de CPU por *thread* da aplicação GL em PThreads.

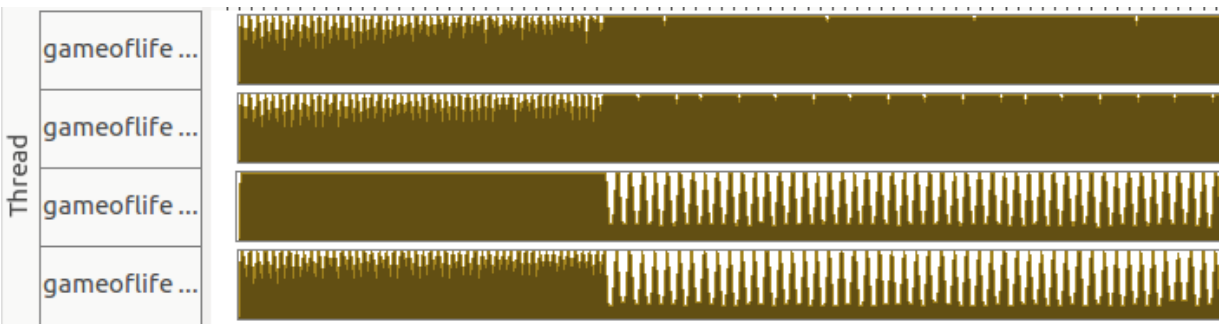


Figura 31 – Uso de CPU por *thread* da aplicação MM em PThreads.

isso, podemos concluir que a existência de trechos que não são altamente paralelizáveis impacta negativamente no TLP de uma aplicação, de forma similar à Lei de Amdahl (HILL; MARTY, 2008), que calcula o impacto de trechos inerentemente sequenciais na paralelização de algoritmos, indicando qual o desempenho máximo que pode ser obtido em determinada aplicação.

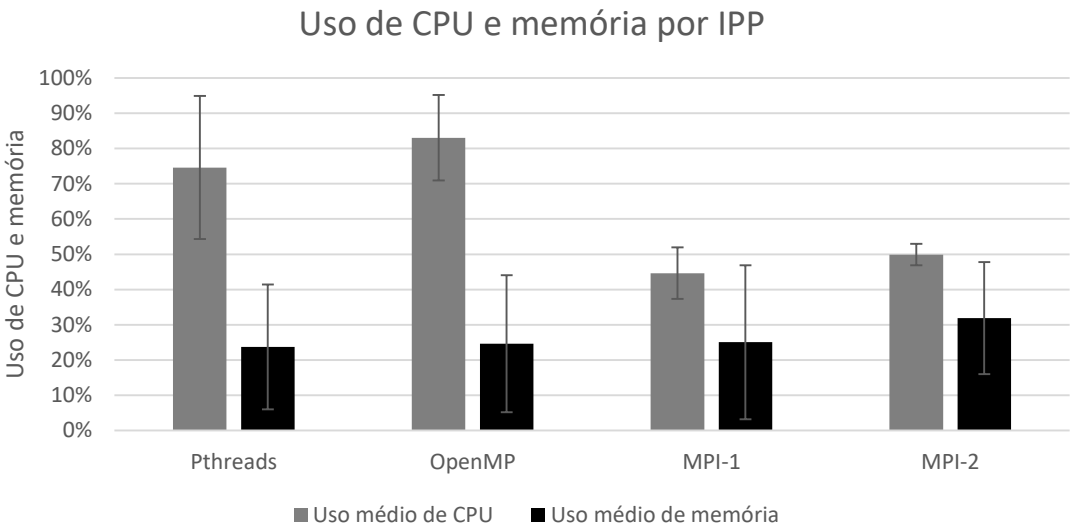
5.4.5 Comparação entre as IPPs

A Figura 32 contém o gráfico com a média de uso de CPU e memória entre as aplicações paralelizadas com cada IPP. O gráfico também inclui o desvio padrão para cada caso analisado.

Na média geral das aplicações do *benchmark* por IPP, pode-se notar que ambas interfaces, PThreads e OpenMP, possuem aplicações com um alto uso de CPU, até cerca de 95% de uso em alguns casos. A maioria das aplicações com PThreads ficou acima dos 75% de uso de CPU, enquanto a maioria das aplicações com OpenMP ficou acima dos 83%.

Em ambas IPPs com MPI o uso de CPU ficou muito inferior em relação à PThreads

Figura 32 – Gráfico da média de uso de CPU e memória por IPP



e OpenMP, cerca de 30% a menos em média. Nos dois casos, o desvio padrão indica que a maioria das aplicações estão próximas da média com MPI 1 e 2, ao contrário de PThreads, onde a porcentagem do uso de CPU varia bastante entre as aplicações.

Em relação ao uso de memória o cenário se inverte, porém com resultados mais próximos entre todas as IPPs. Aplicações em MPI 1 e 2 usam um pouco mais de memória em média, quando comparando com PThreads e OpenMP. De modo geral, as aplicações com MPI-2 apresentaram um maior uso de memória em média, sendo a única IPP em que a maioria das aplicações estão acima da média do uso de memória. Mais da metade das aplicações com essa IPP usaram pelo menos 30% de memória.

Essa diferença no uso de CPU e memória dessas IPPs pode ser explicado no contexto de *threads* e processos. *Threads* costumam ser denominadas como um tipo de processo mais leve para o sistema, enquanto os processos são mais pesados. Uma *thread* compartilha com outras *threads* a sua área de código, de dados e os recursos do sistema operacional. O compartilhamento extensivo faz com que a mudança (escalonamento) e a criação de *threads*, feitas pela CPU seja mínima, se comparada com o chaveamento de contexto entre processos pesados. Embora o chaveamento de contexto de uma *thread* necessite da mudança (chaveamento) de um conjunto de registradores, nenhum processamento relacionado ao gerenciamento de memória é feito. Dessa forma, as *threads* conseguem fazer um melhor uso de CPU, enquanto os processos não conseguem usar a CPU de forma tão eficiente e aumentam o uso de memória devido às trocas de contexto.

Desconsiderando essa diferença entre threads e processos, pode-se concluir baseado nos dados coletados que as aplicações paralelizadas com OpenMP utilizam, em média, mais CPU para computar do que PThreads. Por outro lado, as aplicações paralelizadas

com PThreads são as que fazem em média menor uso de memória. Com MPI-1 o uso médio de memória das aplicações ficou abaixo de 50% na maioria dos casos. MPI-2 possui aplicações que usam em média mais memória entre as IPPs analisadas, esse aumento no consumo em relação a MPI-1 provavelmente está relacionado ao custo da criação dinâmica de processos do MPI-2.

A análise dos resultados mostrou que as aplicações possuem diferentes características e que essas características variam quando são utilizadas diferentes IPPs. Para todas as interfaces existem aplicações que exploram os recursos da arquitetura com maior ou menor intensidade. O estudo das aplicações sequenciais mostrou que as aplicações estão bastante diversificadas com relação às complexidades. Com o TLP foi possível observar que as aplicações na maioria dos casos exploraram o paralelismo das arquiteturas ao máximo, que é o que espera-se de um *benchmark* para avaliar arquiteturas paralelas. Por fim, com o uso de CPU e uso de memória, foi possível notar como as características das aplicações e o seu uso dos recursos do sistema flutuam de acordo com a IPP utilizada.

6 Conclusão

O objetivo inicial deste trabalho envolvia fazer análises e testes em um conjunto de aplicações que estava sendo utilizado para avaliar desempenho e consumo de energia de diferentes interfaces de programação paralela em arquiteturas multinúcleo. Com o resultado dessas análises, esperava-se obter dados que comprovassem que o conjunto de aplicações em questão possuía características suficientes para ser considerado um *benchmark* paralelo. Para atender esses objetivos, primeiramente foram descritos os algoritmos base de cada aplicação. Através desses algoritmos, as aplicações foram classificadas de acordo com as estruturas de dados de entrada, sendo elas: dados não estruturados (algoritmos puramente iterativos), vetores e matrizes. Em conjunto à esses dados, identificou-se a complexidade desses algoritmos. Foram encontradas complexidades mínimas de $O(n)$ e máximas de $O(n^3)$, com uma variedade de complexidade intermediarias. Essa foi a primeira classificação.

Em um segundo momento, para atender a proposta de ser um *benchmark* de aplicações paralelas, foi investigado o grau de paralelismo de cada aplicação, o TLP. Classificando as aplicações de acordo como TLP, foi possível observar o comportamento das diferentes Interfaces de Programação Paralela e qual o impacto de cada uma no desempenho dessas aplicações. No geral, as IPPs que utilizam *threads* (PThreads e OpenMP) para a paralelização atingiram maiores graus de paralelismo em relação às aplicações que utilizam processos (MPI 1 e 2). Esse TLP mostrou níveis similares entre aplicações que possuem uma baixa demanda de comunicação e níveis mais voláteis para algumas aplicações com alta demanda de comunicação. No geral, as aplicações possuem TLPs próximos do ideal, mostrando que podem ser utilizadas para extrair todo o potencial de uma arquitetura multinúcleo.

A terceira etapa, envolveu investigar o impacto de cada IPP no uso de CPU e memória das aplicações. Foram coletados e analisados os dados individuais de todas as aplicações com cada umas das IPPs. Na sequência, as aplicações foram categorizadas entre aplicações que fazem alto e baixo uso de CPU e memória. Isso gerou quatro categorias: alto uso de CPU e alto uso de memória; alto uso de CPU e baixo uso de memória; baixo uso de CPU e alto uso de memória; baixo uso de CPU e baixo uso de memória. Para cada IPP, foi criada uma tabela classificando as aplicações de acordo com essas categorias. De modo geral, as aplicações com todas as IPPs mostraram ter diversidade o suficiente no uso dos recursos (CPU e memória) para preencher as diferentes categorias.

A maior contribuição deste trabalho em relação aos estudos anteriores está na análise do impacto de cada IPP na classificação das aplicações, onde foi possível observar

que existem, de fato, diferentes comportamentos de uma mesma aplicação ao variar a IPP. Fazendo uma análise geral, é possível afirmar que as aplicações possuem diversidade o suficiente em relação aos dados e estruturas de dados utilizadas, complexidade, uso dos recursos do sistema, e exploram altos níveis de paralelismo. Portanto conclui-se que esse conjunto de aplicações pode realmente ser utilizado como um *benchmark* pra avaliar desempenho e consumo de energia de diferentes interfaces de programação paralela em arquiteturas multinúcleo.

Como trabalhos futuros, o primeiro passo será organizar o conjunto de aplicações para que seja possível a sua disponibilização à comunidade científica. Esse trabalho envolve realizar toda a documentação de cada aplicação, revisar os códigos, disponibilizar manuais e arquivos de configuração do *benchmark* para que seja possível realizar análises mais precisas e configurar diferentes tipos de entradas para as aplicações. Além disso, podem ser investigadas novas aplicações para diversificar ainda mais o *benchmark*, como por exemplo, um aplicação com baixo uso de CPU e alto uso de memória em OpenMP, para preencher a lacuna que ficou aberta nessa categoria.

Referências

ABADI, M. et al. Moderately hard, memory-bound functions. *ACM Transactions on Internet Technology (TOIT)*, ACM, v. 5, n. 2, p. 299–327, 2005. Citado na página 56.

ALMASI, G. Review of "highly parallel computing". *IBM Syst. J.*, IBM Corp., Riverton, NJ, USA, v. 29, n. 1, p. 165–166, jan. 1990. ISSN 0018-8670. Citado na página 26.

ANDREWS, G. E.; ROY, R. *Special Functions*. Cambridge, UK; New York, NY, USA: Cambridge University Press, 1999. ISBN 9781107266865 1107266866 9781107325937 1107325935. Citado na página 41.

ASANOVIC, K. et al. *The landscape of parallel computing research: A view from berkeley*. [S.l.], 2006. Citado 2 vezes nas páginas 24 e 26.

BAILEY, D. H. et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, SAGE Publications, v. 5, n. 3, p. 63–73, 1991. Citado na página 38.

BLAKE, G. et al. Evolution of thread-level parallelism in desktop applications. In: ACM. *ACM SIGARCH Computer Architecture News*. [S.l.], 2010. v. 38, n. 3, p. 302–313. Citado na página 62.

BORWEIN, J. M.; BORWEIN, P. B.; DILCHER, K. Pi, euler numbers, and asymptotic expansions. *The American Mathematical Monthly*, Mathematical Association of America, v. 96, n. 8, p. pp. 681–687, 1989. ISSN 00029890. Citado na página 41.

BURDEN, R. L.; FAIRES, J. D. Análise Numérica, Pioneira Thomson Learnign. *São Paulo*, 2003. Citado na página 43.

BUTENHOF, D. R. *Programming with POSIX threads*. [S.l.]: Addison-Wesley Professional, 1997. Citado 2 vezes nas páginas 31 e 51.

CALLIOLI, C. A.; DOMINGUES, H. H.; COSTA, R. C. F. *Álgebra linear e aplicações*. [S.l.]: Atual, 2007. Citado na página 46.

CAPPELLO, F. et al. Toward exascale resilience. *International Journal of High Performance Computing Applications*, SAGE Publications, 2009. Citado na página 19.

CHAPMAN, B.; JOST, G.; PAS, R. V. D. *Using OpenMP: portable shared memory parallel programming*. [S.l.]: MIT press, 2008. v. 10. Citado na página 52.

CHENEY, W.; KINCAID, D. Linear algebra: Theory and applications. *The Australian Mathematical Society*, p. 654, 2009. Citado na página 48.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik*, v. 1, p. 269–271, 1959. Citado na página 43.

FELDMAN, M. *HPC Challenge Awards Competition*. 2012. Disponível em: <https://www.hpcwire.com/2012/01/23/intel_snaps_up_infiniband_technology_product_line_from_qlogic/>. Citado na página 19.

FOSTER, I. *Designing and building parallel programs*. [S.l.]: Addison Wesley Publishing Company, 1995. Citado 3 vezes nas páginas 27, 51 e 52.

GAO, C. et al. A study of thread level parallelism on mobile devices. In: IEEE. *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. [S.l.], 2014. p. 126–127. Citado na página 62.

GARDNER, M. The Fantistic Combinatons of John Conway's New Solitaire Game Life. *Scientific American*, p. 223,120, 1970. Citado na página 46.

GOLDSTON, D.; YILDIRIM, C. Y. *On the second moment for primes in an arithmetic progression*. 2001. 85–104 p. Citado na página 42.

GRAMA, A. *Introduction to parallel computing*. [S.l.]: Pearson Education, 2003. Citado na página 31.

GRAY, J. *Benchmark handbook: for database and transaction processing systems*. [S.l.]: Morgan Kaufmann Publishers Inc., 1992. Citado 2 vezes nas páginas 23 e 24.

GROPP, W.; LUSK, E.; THAKUR, R. *Using MPI-2: Advanced features of the message-passing interface*. [S.l.]: MIT press, 1999. Citado 4 vezes nas páginas 27, 32, 35 e 51.

GUTHAUS, M. R. et al. Mibench: A free, commercially representative embedded benchmark suite. In: IEEE. *Workload Characterization, IEEE Workshop on*. [S.l.], 2001. p. 3–14. Citado na página 37.

HENNESSY, J. L.; PATTERSON, D. A. *Computer architecture: a quantitative approach*. [S.l.]: Elsevier, 2011. Citado na página 25.

HILL, M. D.; MARTY, M. R. Amdahl's law in the multicore era. *IEEE Computer*, v. 41, p. 33–38, 2008. Citado na página 83.

ICL. *PAPI Counter Interfaces*. 2015. Disponível em: <icl.cs.utk.edu/projects/papi/wiki/PAPIC:Overview>. Citado na página 56.

INTEL. *Intel® Pin - A Dynamic Binary Instrumentation Tool*. 2012. Disponível em: <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>. Citado na página 56.

INTEL. *Intel® Parallel Studio XE*. 2015. Disponível em: <https://software.intel.com/en-us/intel-parallel-studio-xe>. Citado na página 62.

INTEL. *Intel® VTune™ Amplifier 2017*. 2016. Disponível em: <https://software.intel.com/en-us/intel-vtune-amplifier-xe>. Citado na página 59.

IQBAL, S.; LIANG, Y.; GRAHN, H. Parmibench - an open-source benchmark for embedded multiprocessor systems. *Computer Architecture Letters*, v. 9, n. 2, p. 45–48, Feb 2010. ISSN 1556-6056. Citado na página 37.

JI, J.; WANG, C.; ZHOU, X. System-level early power estimation for memory subsystem in embedded systems. In: IEEE. *Fifth IEEE International Symposium on Embedded Computing*. [S.l.], 2008. p. 370–375. Citado na página 19.

KNUTH, D. E. *The art of computer programming: sorting and searching*. [S.l.]: Pearson Education, 1998. v. 3. Citado na página 49.

KORTHIKANTI, V. A.; AGHA, G. Towards optimizing energy costs of algorithms for shared memory architectures. In: ACM. *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. [S.l.], 2010. p. 157–165. Citado 3 vezes nas páginas 19, 25 e 26.

KRUSKAL, C. P.; RUDOLPH, L.; SNIR, M. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, Elsevier, v. 71, n. 1, p. 95–132, 1990. Citado na página 69.

LORENZON, A. F. *Avaliação do desempenho e consumo energético de diferentes interfaces de programação paralela em sistemas embarcados e de propósito geral*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, 2014. Citado 6 vezes nas páginas 20, 30, 41, 52, 53 e 55.

LORENZON, A. F.; CERA, M. C.; BECK, A. C. S. Performance and energy evaluation of different multi-threading interfaces in embedded and general purpose systems. *Journal of Signal Processing Systems*, Springer, v. 80, n. 3, p. 295–307, 2014. Citado 4 vezes nas páginas 41, 55, 56 e 57.

LORENZON, A. F.; CERA, M. C.; BECK, A. C. S. On the influence of static power consumption in multicore embedded systems. In: IEEE. *Circuits and Systems (ISCAS), 2015 IEEE International Symposium on*. [S.l.], 2015. p. 1374–1377. Citado 3 vezes nas páginas 41, 55 e 58.

LORENZON, A. F.; CERA, M. C.; BECK, A. C. S. Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy. *Journal of Parallel and Distributed Computing*, Elsevier, v. 95, p. 107–123, 2016. Citado na página 61.

LORENZON, A. F.; CERA, M. C.; ROSSI, F. D. Analysing the impact of mpi-2 dynamic process creation to the game of life problem. In: IEEE. *Computer Systems (WSCAD-SSC), 2012 13th Symposium on*. [S.l.], 2012. p. 133–140. Citado na página 35.

LORENZON, A. F. et al. The influence of parallel programming interfaces on multicore embedded systems. In: IEEE. *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*. [S.l.], 2015. v. 2, p. 617–625. Citado 2 vezes nas páginas 41 e 55.

LORENZON, A. F. et al. Optimized use of parallel programming interfaces in multithreaded embedded architectures. In: *VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on*. [S.l.: s.n.], 2015. p. 410–415. Citado 4 vezes nas páginas 41, 55, 59 e 60.

MAILLARD, N.; CERA, M. C. Message-passing interface avançado. *10th Escola Regional de Alto Desempenho – SBC*, 2010. Citado na página 34.

MIKLOŠKO, J.; KOTOV, V. E. Complexity of parallel algorithms. In: *Algorithms, software and hardware of parallel computers*. [S.l.]: Springer, 1984. p. 45–63. Citado na página 69.

OLIVEIRA, A. de; SCHARCANSKI, J. Vehicle counting and trajectory detection based on particle filtering. In: *Graphics, Patterns and Images (SIBGRAPI), 2010 23rd SIBGRAPI Conference on*. [S.l.: s.n.], 2010. p. 376–383. Citado na página 45.

OpenMP ARB. The OpenMP® API specification for parallel programming. 2015. Disponível em: <http://openmp.org/wp/>. Citado na página 28.

PAUDEL, J.; AMARAL, J. N. Using the Cowichan Problems to Investigate the Programmability of X10 Programming System. In: *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*. New York, NY, USA: ACM, 2011. (X10 '11), p. 4:1–4:10. ISBN 978-1-4503-0770-3. Citado na página 49.

PETITET, A. Hpl-a portable implementation of the high-performance linpack benchmark for distributed-memory computers. 2004. Disponível em: <http://www.netlib.org/benchmark>. Citado na página 37.

POLETTI, F. et al. Energy-efficient multiprocessor systems-on-chip for embedded computing: Exploring programming models and their architectural support. *Computers, IEEE Transactions on*, IEEE, v. 56, n. 5, p. 606–621, 2007. Citado na página 32.

POOVEY, J. et al. A benchmark characterization of the eembc benchmark suite. *International Symposium on Microarchitecture*, IEEE, 2009. Citado na página 36.

PRESS, W. H. et al. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3. ed. [S.l.]: Cambridge University Press, 2007. ISBN 0521880688. Citado na página 44.

RAUBER, T.; RÜNGER, G. *Parallel programming: For multicore and cluster systems*. [S.l.]: Springer Science & Business Media, 2010. Citado 7 vezes nas páginas 23, 26, 27, 28, 30, 51 e 52.

REVISTABW. *OpenMP: Uso da biblioteca e construtores*. 2015. Disponível em: <http://www.revistabw.com.br/revistabw/openmp-biblioteca-construtores/>. Citado na página 29.

ROY, R. The Discovery of the Series Formula for pi by Leibniz, Gregory and Nilakantha. *Mathematics Magazine*, Mathematical Association of America, v. 63, n. 5, p. pp. 291–306, 1990. ISSN 0025570X. Citado na página 41.

SMITH, S. W. et al. The scientist and engineer's guide to digital signal processing. California Technical Pub. San Diego, 1997. Citado na página 50.

SPEC. *Standard Performance Evaluation Corporation*. 2016. <https://www.spec.org/>. Acessado: 08-01-2016. Citado 2 vezes nas páginas 21 e 24.

STEWART, J. Cálculo, vol. 1. *Pioneira Thomson Learning*, 2001. Citado na página 47.

TANENBAUM, A. S. et al. *Operating systems: design and implementation*. [S.l.]: Prentice-Hall Englewood Cliffs, NJ, 1987. v. 2. Citado na página 32.

TERUEL, X. et al. Openmp tasking analysis for programmers. In: IBM CORP. *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*. [S.l.], 2009. p. 32–42. Citado na página 28.

TURING, A. M. The chemical basis of morphogenesis. *Phil. Trans. Roy. Soc.*, v. 237, p. 37, 1952. Citado na página 49.

WEHNER, M.; OLIKER, L.; SHALF, J. A real cloud computer. *Spectrum, IEEE*, IEEE, v. 46, n. 10, p. 24–29, 2009. Citado na página 19.

WILSON, G. V.; BAL, H. E. Using the cowichan problems to assess the usability of orca. *Parallel & Distributed Technology: Systems & Applications, IEEE*, IEEE, v. 4, n. 3, p. 36–44, 1996. Citado na página 51.

Apêndices

APÊNDICE A – ESTRATÉGIA DE PARALELIZAÇÃO

A.1 Cálculo do Pi

A implementação paralela da aplicação do Cálculo do Pi com OpenMP ocorre através da inserção da diretiva `#pragma omp parallel for schedule` entre as linhas 2 e 3 do pseudocódigo 1. Os valores contidos internamente à diretiva `schedule` definem como as iterações serão atribuídas entre as *threads* na região paralela. Para que cada *thread* possa controlar suas próprias iterações sem interferência das demais, é necessário definir a váriável de iteração do laço como privada. Ao final da região paralela é necessário fazer uma redução para que a *thread* mestre faça a soma do valor de Pi calculado por cada *thread*.

Em Pthreads, a paralelização é feita através da divisão do número total iterações entre o número de *threads* de forma estática. Por exemplo, considerando que o número de iterações a ser calculado seja igual a 100 e que existem 4 *threads*, a $thread_0$ computará as iterações de 0 até 24; a $thread_1$, as iterações de 25 até 49 e assim consecutivamente com a $thread_2$ e a $thread_3$, mantendo o mesmo tamanho na distribuição dos blocos que executarão em paralelo. Também são utilizadas variáveis do tipo *mutex* ao final do laço `for` para fazer a redução.

A paralelização com MPI-1 difere da implementação com PThreads apenas através da inserção da função `MPI_Reduce()` ao final do laço principal. Essa função substitui o *mutex* de PThreads e tem o mesmo objetivo da diretiva `reduce` do OpenMP. Por fim, MPI-2 difere de MPI-1 apenas por realizar a criação dinâmica de processos.

A.2 Série Harmônica

A paralelização da aplicação que calcula a Série Harmônica com OpenMP ocorre através da divisão do número de iterações do laço mais externo entre as **threads**. Para tal, foi inserida a diretiva `#pragma omp parallel for schedule` na linha 3 do pseudocódigo 14. As variáveis de iteração dos dois laços (linhas 4 e 5) devem ser definidas como privadas. Foi necessário realizar algumas modificações na estrutura do algoritmo. Isso se deve ao fato de que ao final do laço principal na linha 11, deve ser feita a redução dos valores internos ao vetor, pois cada *thread* computa valores separados das posições do vetor. Como o OpenMP não possui operações do tipo *reduce* para vetores, então redução é feita

através da diretiva de seção crítica `#pragma omp critical` para cada *thread* atualizar o valor final do vetor na série.

Algorithm 14 Série Harmônica

```
1: Início
2: Define o número de iterações
3: #pragma omp parallel for schedule
4:     for (Até atingir o número de iterações) do
5:         for (Percorre vetor) do
6:             Cálculo da Série Harmônica
7:             Armazena a soma da precisão a cada iteração em um vetor auxiliar
8:         end for
9:     end for
10: end pragma
11: #pragma omp critical
12:     for (Percorre vetor) do
13:         Redução dos valores internos ao vetor
14:     end for
15: end pragma
16: for (Percorre vetor) do
17:     Ajuste da precisão decimal
18: end for
19: Fim
```

Na implementação com PThreads, busca-se obter um bom balanceamento de carga através da divisão das iterações do laço mais externo do cálculo principal. Para fazer a redução ao final do trecho paralelo utilizam-se variáveis do tipo *mutex* para possibilitar que cada *thread* atualize o valor final do vetor de saída.

Com MPI-1, a divisão da carga de trabalho ocorre da mesma forma que em Pthreads. Contudo, MPI fornece redução de valores em vetores através da função `MPI_Reduce`. Portanto, ela foi utilizada para a atualização do vetor de saída.

Por fim, em MPI-2, os processos foram criados dinamicamente e devido a não possuir operações do tipo reduce entre intercomunicadores, cada processo "filho" retornou ao processo "pai" os dados computados (através das funções `MPI_Send - MPI_Recv`) e o processo pai realizou a soma destes valores em um novo vetor.

A.3 Dijkstra

A paralelização do algoritmo do Caminho Mínimo de Dijkstra se deu através da divisão do número de vértices a serem computados. Para tal, são divididas as iterações do laço da linha 2 do pseudocódigo 3. Para a implementação paralela com o OpenMP foi inserida a diretiva `#pragma omp parallel for schedule` antes desse laço principal. Os valores contidos internamente à diretiva `schedule` definem como as iterações serão

atribuídas entre as *threads* na região paralela. A variável de iteração do laço deve ser definida como privada, para que cada *thread* possa controlar e computar suas próprias iterações sem a interferência das demais.

A paralelização em Pthreads ocorre através da divisão do número de vértices a serem procurados pelo número de *threads*. Por exemplo, considerando que o número de vértices seja igual a 100 e que existem 4 *threads*, a $thread_0$ computará as iterações de 0 até 24; a $thread_1$, as iterações de 25 até 49 e assim consecutivamente com a $thread_2$ e a $thread_3$, mantendo o mesmo tamanho na distribuição dos blocos que executarão em paralelo.

MPI-1 possui uma distribuição da carga de trabalho no mesmo formato utilizado Pthreads. No entanto, ao final do laço for da linha 2, é necessário unificar o resultado obtido pela computação de cada processo. Para tal, cada processo envia seus resultados para o processo principal através das diretivas `MPI_Send/MPI_Recv`. Por fim, na implementação em MPI-2, é implementada a criação dinâmica de processos com relação a paralelização em MPI-1.

A.4 Método de Jacobi

A paralelização desse algoritmo se deu através da divisão das iterações do laço da linha 4 (pseudocódigo 4) entre as *threads*/processos. A implementação paralela com OpenMP ocorre através do uso da diretiva `#pragma omp parallel` entre as linhas 2 e 3, onde as variáveis utilizadas para percorrer a matriz devem ser privadas; `#pragma omp for schedule` entre as linhas 3 e 4 para dividir as iterações entre as *threads*; e `#pragma omp master` na linha 10, para que a troca de dados entre as duas matrizes seja feita por apenas pela thread principal.

Com Pthreads, as iterações do laço da linha 4 foram divididas entre as *threads* buscando obter o melhor balanceamento da carga de trabalho. Essa divisão das iterações representa dividir o número de linhas da matriz.

A distribuição da carga de trabalho entre os processos com MPI-1 seguiu o mesmo padrão que Pthreads. Inicialmente o processo P_0 particiona os dados que cada processo irá computar e envia essa informação através das funções `MPI_Send/MPI_Irecv` na linha 3. Após cada processo computar sua parcela de dados, eles precisam atualizar seus dados com relação aos demais processos utilizado a função `MPI_Bcast` na linha 10.

A mesma divisão da carga de trabalho em MPI-1 foi aplicada em MPI-2. Porém, para atualizar os dados entre todos os processos foi necessário trocar a função `MPI_Bcast` por diretivas de comunicação ponto a ponto `MPI_Send/MPI_Irecv`. Isto porque, de acordo com o relacionamento hierárquico gerado pela criação de processos, não é possível realizar

tais comunicações coletivas.

A.5 Multiplicação de Matrizes

A implementação paralela desta aplicação deu-se através da divisão das iterações do laço `for` que percorre as linhas da matriz, conforme o pseudocódigo 5. Para tal, a implementação paralela com o OpenMP ocorreu utilizando a diretiva `#pragma omp parallel for schedule` antes do laço mais externo. Assim, as iterações serão divididas entre as *threads* conforme o `schedule` definido. Todas as variáveis utilizadas nos laços que percorrem a matriz devem ser definidas como privadas. As variáveis privadas são necessárias para que uma *thread* não interfira na computação das demais.

A paralelização com Pthreads, ocorreu através da divisão da altura da matriz entre as *threads*. Por exemplo, considerando que a altura da matriz seja igual a 1024 e que existem 4 *threads*. A $thread_0$ computará as iterações de 0 até 255; a $thread_1$, as iterações de 256 até 511 e assim consecutivamente com a $thread_2$ e a $thread_3$, mantendo o mesmo tamanho na distribuição dos blocos que executarão em paralelo.

MPI-1 possui uma divisão de carga de trabalho igual à aplicada em Pthreads, porém existem duas comunicações adicionais. A primeira comunicação consiste em o processo P_0 enviar as porções da matriz A (`MPI_Send/MPI_Irecv`) que cada processo irá computar e a matriz B (`MPI_Bcast`). No momento em que os processos finalizam a computação da sua parcela de trabalho, ambos retornam ao processo P_0 (que também está computando) o resultado de sua computação. Para implementar essas comunicações foram utilizadas as funções `MPI_Send` e `MPI_Recv`. Caso a divisão do número de linhas da matriz pelo número de processos seja exata, as comunicações são através das funções `MPI_Gatter` e `MPI_Scatter`. Por fim, na implementação com MPI-2 os processos são criados dinamicamente.

A.6 Similaridade entre Histogramas

A paralelização desse algoritmo se deu através da divisão das iterações do laço mais externo (pseudocódigo 6) entre as *threads*/processos. A implementação paralela com OpenMP ocorre através do uso da diretiva `#pragma omp parallel private` entre as linhas 1 e 2, sendo que as variáveis utilizadas para percorrer a matriz de píxeis devem ser privadas; e, ainda entre as linhas 1 e 2, `#pragma omp for schedule` para dividir as iterações entre as *threads*.

Com Pthreads, as iterações desse laço são divididas entre as *threads* buscando obter o melhor balanceamento da carga de trabalho. Essa divisão das iterações representa dividir o número de linhas da matriz.

Em MPI-1 é implementado o mesmo padrão de distribuição de carga feito em PThreads. Inicialmente o processo P_0 envia a parcela de dados que cada processo irá computar através das funções `MPI_Send/MPI_Irecv` antes da linha 2. Após cada processo computar sua parcela de dados, eles atualizam seus dados com relação aos demais processos utilizando o `MPI_Bcast`.

Na paralelização com MPI-2 utilizou-se a mesma divisão da carga de trabalho que em MPI-1 e PThreads. Entretanto, ao invés da utilização do `MPI_Bcast` para atualizar os dados entre todos os processos, foram utilizadas as diretivas de comunicação ponto a ponto `MPI_Send/MPI_Irecv`. Isto porque, de acordo com o relacionamento hierárquico gerado pela criação de processos, não é possível realizar tais comunicações coletivas.

A.7 Produto Escalar

A paralelização desse algoritmo deu-se através da divisão das iterações do seu único laço. A implementação paralela com OpenMP ocorre através da inserção da diretiva `#pragma omp parallel for schedule` entre as linhas 2 e 3 do pseudocódigo 7. A váriável de iteração do laço deve ser difinida como privada, para que cada *thread* possa controlar suas próprias iterações sem interferência das demais. Ao final da região paralela é necessário fazer uma redução para que a *thread* mestre faça a soma do valor calculado por cada *thread*.

Em Pthreads, a paralelização é feita através da divisão do número total iterações entre o número de *threads* de forma estática. Essa divisão das iterações representa dividir o número de linhas da matriz. Variáveis do tipo *mutex* devem ser usadas ao final do laço para fazer a redução.

A paralelização com MPI-1 difere da implementação com PThreads apenas através da inserção da função `MPI_Reduce()` ao final do laço principal. Essa função substitui o *mutex* de PThreads e tem o mesmo objetivo da diretiva `reduce` do OpenMP. Por fim, MPI-2 difere de MPI-1 apenas por realizar uma criação dinâmica de processos.

A.8 Jogo da Vida

A paralelização ocorreu através da divisão das iterações do laço da linha 4 (pseudocódigo 8) entre as *threads*/processos. A dependência de dados existente devido a computação em uma única matriz foi resolvida adicionando uma matriz auxiliar, que conterá o valor correto ao final de cada computação. No final da computação do laço na linha 7, a matriz solução é atualizada com uma troca de dados entre as duas matrizes, a qual deve ser executada por um único processo.

Na paralelização com OpenMP foi utilizada a diretiva `#pragma omp parallel` entre as linhas 2 e 3; `#pragma omp for schedule` para dividir as iterações do laço `for` da linha 4 entre as *threads*; e `#pragma omp master` antes da linha 16.

Com Pthreads, as iterações do laço for da linha 4 foram divididas entre as *threads* buscando obter o melhor balanceamento da carga de trabalho. Essa divisão das iterações representa dividir o número de linhas da matriz. A troca de dados entre a matriz auxiliar e a matriz solução é executada pela $thread_0$, enquanto as demais *threads* ficam aguardando em uma barreira.

O mesmo padrão de distribuição de carga de trabalho utilizado em PThreads foi implementado com MPI-1. Inicialmente o processo P_0 envia a parcela de dados que cada processo irá computar através das funções `MPI_Send/MPI_Irecv` antes da linha 4. Também, após cada processo computar sua parcela de dados, eles precisam atualizar seus dados com relação aos demais processos. Para isto, foi utilizado o `MPI_Bcast` na linha 7.

Na paralelização com MPI-2 utilizou-se a mesma divisão da carga de trabalho que em MPI-1 e PThreads. Contudo, ao invés da utilização do `MPI_Bcast` para atualizar os dados entre todos os processos, foi utilizado as diretivas de comunicação ponto a ponto `MPI_Send/MPI_Irecv`. Isto porque, de acordo com o relacionamento hierárquico gerado pela criação de processos, não é possível realizar tais comunicações coletivas.

A.9 Integração Numérica

A paralelização ocorreu através da divisão das iterações do laço da linha 4 (pseudocódigo 9) entre as *threads*/processos. A implementação paralela com OpenMP ocorre através do uso da diretiva `#pragma omp parallel private` entre as linhas 3 e 4, sendo que a variável de iteração e a variável que armazena o resultado temporariamente devem ser privadas; ainda entre as linhas 3 e 4, é utilizada a diretiva `#pragma omp for schedule` para dividir as iterações entre as *threads*. Ao final da região paralela é necessário fazer uma redução para que a *thread* mestre faça a soma do valor calculado por cada *thread*.

Em Pthreads, a paralelização é feita através da divisão do número total de iterações entre o número de *threads* de forma estática. Essa divisão das iterações representa dividir o número de linhas da matriz. Variáveis do tipo *mutex* são utilizadas ao final do laço para fazer a redução.

Com MPI-1 a paralelização difere da implementação com PThreads apenas através da inserção da função `MPI_Reduce()` ao final do laço principal. Essa função substitui o *mutex* de PThreads e temo mesmo objetivo da diretiva `reduce` do OpenMP. Por fim, MPI-2 difere de MPI-1 apenas por realizar uma criação dinâmica de processos.

A.10 Gram-Schmidt

Para paralelizar este algoritmo, as iterações dos laços `for` das linhas 3, 6, e 9 (pseudocódigo 10) foram divididas entre as *threads*/processos. Os cálculos são computados utilizando uma variável temporária, porém nos laços das linhas 3 e 9 ocorrem alterações nessa variável, que será utilizada nos laços `for` seguintes. Portanto, ao final destes laços `for` é necessário atualizar a variável temporária em todas as *threads*/processos através da operação de redução. As variáveis de iteração dos laços das linhas 3 e 6 devem ser definidas como privadas. Para evitar problemas de consistência dos dados, a execução de cada laço só pode começar após a execução do laço anterior concluir. Portanto, é necessário utilizar barreiras temporais ao fim destes laços `for`.

A paralelização com OpenMP ocorre através do uso da diretiva `#pragma omp parallel for private reduction` no laço `for` da linha 3 e 9. No laço da linha 6 foi inserida a mesma diretiva sem a operação de redução. No OpenMP, já existe uma barreira temporal ao final de cada laço paralelizado.

Com Pthreads, as iterações destes laços foram divididas entre as *threads* buscando obter o melhor balanceamento da carga de trabalho. Essa divisão das iterações representa dividir o número de linhas da matriz. Como barreira temporal, foi utilizada a função `pthread_barrier_wait` e o mutex foi utilizado para fazer as operações de redução na variável temporária.

Nas implementações com MPI foi utilizada a mesma distribuição da carga de trabalho que com Pthreads. No entanto, após cada laço `for`, foi necessário incluir as diretivas de comunicação coletivas (`MPI_Allreduce`) entre todos os processos. A diferença do MPI-1 para o MPI-2 é a presença da criação dinâmica de processos, além da utilização de diretivas de comunicação ponto a ponto (`MPI_Send/MPI_Irecv`) no MPI-2.

A.11 Ordenação Par-Ímpar

A paralelização dessa aplicação se deu através da divisão das iterações dos laços das linhas 4 e 9 (pseudocódigo 11) entre as *threads*/processos. No entanto, de acordo com a característica do algoritmo, a computação do laço da linha 9 só poderá iniciar após a finalização por completo da computação do laço da linha 4. Para tanto, é necessário inserir uma barreira antes da linha 4. O mesmo é necessário para não iniciar a computação do laço da linha 4 antes da conclusão do laço da linha 9.

Com OpenMP a paralelização ocorre através da inserção das diretivas `#pragma omp parallel private` antes da linha 3, para criar as *threads* uma única vez; e `#pragma omp for schedule` antes dos laços das linhas 3 e 5. Nota-se que todas as variáveis de iteração devem ser definidas como privadas. Ao final de cada laço paralelo, as *threads*

sincronizam, portanto não é necessária a inserção de diretivas explicitas de sincronização.

Em Pthreads, as iterações dos laços das linhas 4 e 9 são divididas entre as *threads* buscando obter o melhor balanceamento da carga de trabalho. Para realizar a barreira entre as *threads*, é utilizada a função `pthread_barrier_wait()`, antes do laço das linhas 4 e 9. Assim, a computação destes laços sempre irá iniciar de forma sincronizada.

A paralelização com MPI-1 é feita da mesma forma que Pthreads. No entanto, inicialmente P_0 envia para os demais processos (utilizando `MPI_Send/MPI_Irecv`) a parcela de dados que cada um irá computar conforme atribuído na divisão da carga de trabalho. Após a computação do laço da linha 4 e 9 são inseridas funções de comunicação (troca dos dados de borda através de `MPI_Send/MPI_Irecv`) entre os processos para ambos processadores computarem sobre os dados atualizados. Já em MPI-2, adicionou-se a criação dinâmica de processos à paralelização com MPI-1.

A.12 Turing Ring

A paralelização é feita através da divisão das iterações dos laços for das linhas 4 e 10 (pseudocódigo 12) entre as *threads*/processos. No entanto, de acordo com a característica do algoritmo, a computação do laço da linha 10 só poderá iniciar após a finalização por completo da computação do laço da linha 4. Para isso, é necessário inserir uma barreira antes da linha 10. O mesmo é necessário para não iniciar a computação do laço for da linha 4 antes da conclusão do laço da linha 10.

Com OpenMP a paralelização ocorre utilizando as diretivas: `#pragma omp parallel private` antes da linha 3, para criar as *threads* uma única vez; e `#pragma omp for schedule` antes dos laços das linhas 4 e 10. As variáveis de iteração de todos os laços devem ser definidas como privadas. Ao final de cada laço paralelo, as *threads* sincronizam, portanto não é necessária a inserção de diretivas explicitas de sincronização.

Com Pthreads, as iterações dos laços das linhas 4 e 10 foram divididas entre as *threads* buscando obter o melhor balanceamento da carga de trabalho. Para realizar a barreira entre as *threads*, é utilizada a função `pthread_barrier_wait()`, antes do laço das linhas 4 e 10. Assim, a computação destes laços sempre irá iniciar de forma sincronizada.

A paralelização com MPI-1 ocorre da mesma forma que foi feita em Pthreads. No entanto, existe a necessidade de atualizar a parcela de dados computados em cada processo. Assim, após a computação do laço da linha 10 devem ser inseridas funções de comunicação (`MPI_Send/MPI_Irecv`) entre os processos. Em MPI-2 a criação dinâmica de processos e a divisão da carga de trabalho e comunicação ocorre da mesma forma aplicada no MPI-1.

A.13 Transformada Discreta de Fourier

A paralelização ocorreu através da divisão das iterações do laço mais externo na linha 2 (pseudocódigo 9) entre as *threads*/processos. A implementação paralela com OpenMP ocorre através do uso da diretiva `#pragma omp parallel private` antes da linha 2, sendo que a variável de iteração ser definida como privada; ainda antes da linha 2, é utilizada a diretiva `#pragma omp for schedule` para dividir as iterações entre as *threads*.

A paralelização com PThreads é feita através da divisão do número total iterações entre o número de *threads* de forma estática. Por exemplo, considerando que o tamanho do vetor de entrada seja igual a 100 e que existem 4 *threads*, a $thread_0$ computará as iterações de 0 até 24; a $thread_1$, as iterações de 25 até 49 e assim consecutivamente com a $thread_2$ e a $thread_3$, mantendo o mesmo tamanho na distribuição dos blocos que executarão em paralelo.

O mesmo padrão de distribuição de carga implementado em PThreads é aplicado em MPI-1. Entretanto, a parcela de dados computados em cada processo deve ser atualizada. Para isso, após a computação do laço na linha 10 devem ser inseridas funções de comunicação (`MPI_Send/MPI_Irecv`) entre os processos. Em MPI-2 a criação dinâmica de processos e a divisão da carga de trabalho e comunicação ocorre da mesma forma aplicada no MPI-1.

Índice

CPU, 18, 19, 22, 23, 57–59, 61

DFT, 45, 57

FIFO, 31

HC, 55, 57–59, 61
HPC, 22
HPL, 22, 23

ILP, 26
IPP, 18–22, 26, 34–37, 47, 50, 52, 54

LC, 55, 57, 58, 60, 61

MPI, 9, 26, 31–36, 47–50, 73–81

NAS, 22, 23

OpenMP, 9, 26–30, 35, 47–50, 60, 73–81

PAPI, 52, 53, 55, 58, 60, 61
PThreads, 26, 29, 30, 35, 47–50, 73, 74, 77, 78, 81

SPEC, 22

TCC, 65, 66
TLP, 19, 26, 51, 53–58, 61