UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ARTHUR FRANCISCO LORENZON

Avaliação do Desempenho e Consumo Energético de Diferentes Interfaces de Programação Paralela em Sistemas Embarcados e de Propósito Geral

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Antonio Carlos Schneider Beck Filho
Co-orientador: Prof. Dr. Márcia Cristina Cera

Porto Alegre
2014

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Lorenzon, Arthur Francisco

Avaliação do Desempenho e Consumo Energético de Diferentes Interfaces de Programação Paralela em Sistemas Embarcados e de Propósito Geral / Arthur Francisco Lorenzon. – 2014.

166 f.

Orientador: Antonio Carlos Schneider Beck Filho; Co-orientadora: Márcia Cristina Cera.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2014.

1. Programação Paralela. 2. Sistemas Embarcados 3. Propósito Geral 4. Eficiência Energética. Beck, Antonio Carlos Schneider. II. Cera, Márcia Cristina. III. Avaliação de Desempenho e Energia de Diferentes Interfaces de Programação Paralela em Sistemas Embarcados e de propósito Geral.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitor de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento

Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb

Coordenador do PPGC: Prof. Luigi Carro

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

À minha Família e

À memória de meu padrinho Reimar.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a Deus por mais esta conquista.

Agradeço a meu orientador, Prof. Antonio Carlos S. Beck por ter se mostrado um excelente orientador durante este período, pelos ensinamentos, e também por ter acreditado em mim. Espero ter conseguido retribuir à ótima orientação.

Agradeço a minha coorientadora, Prof.ª. Márcia C. Cera, que acompanha minha trajetória desde a graduação. Tenha certeza que o aprendizado que tive com a senhora durante o período de graduação foi de suma importância durante este período do mestrado.

Agradeço aos funcionários, corpo docente e colaboradores do PPGC e INF da UFRGS e ao CNPq pelo suporte financeiro. Estendo meus agradecimentos aos colegas do Laboratório de Sistemas Embarcados (LSE-UFRGS) pelos momentos compartilhados de saberes e descontração: Anderson, Andrws, Gabriel, Girão, Jeferson, Jorge, Paolo, Paulo, Rafael, Ronaldo, Tiago, Thiago e Ulisses.

Agradeço aos amigos de Porto Alegre que estiveram presentes durante este período e pelo apoio prestado nos momentos de dificuldade: Ademir, Adriano, Arthur, Cassiano, Fábio, Larissa, Marcelo, Matheus, Mônia e Vinicius. Também aos meus amigos e colegas de graduação: Brandon, Henrique, Jaline, Jean, Sander e Thiarles. Todos vocês, sem exceção serão lembrados pelo apoio.

Por último, mas não menos importante, agradeço a minha família (Osvaldo, Senilda, Allyne, Jeferson e Matheus) por todo o apoio incondicional prestado durante mais esta etapa. Vocês foram de extrema importância, fornecendo o suporte necessário nos momentos de dificuldade. Peço desculpas pelas ausências nos momentos importantes.

RESUMO

Nos sistemas computacionais atuais, enquanto é necessário explorar a disponibilidade de múltiplos núcleos, também é obrigatório consumir menos energia. Para acelerar o processo de desenvolvimento de aplicações paralelas e o tornar mais transparente ao programador, Interfaces de Programação Paralela (IPPs) são largamente utilizadas. Entretanto, cada IPP implementa diferentes formas para trocar dados usando regiões compartilhadas da memória. Estas regiões são, geralmente, mais distantes do processador do que regiões privadas da memória e, por consequência, possuem maior tempo de acesso e consumo de energia. Ademais, o sistema de memória dos processadores embarcados é diferente em hierarquia, tamanho, tempo de acesso, consumo de energia, etc., quando comparado aos processadores de propósito geral. Assim, considerando o cenário supracitado, com diferentes IPPs sendo utilizadas em sistemas *multicore* com diferentes requisitos, neste trabalho será mostrado que cada interface possui comportamento diferente em termos de desempenho, consumo de energia e *Energy-Delay Product* (EDP), e que este comportamento varia de acordo com a característica da aplicação e o processador utilizado (propósito geral ou embarcado). Por exemplo, Pthreads consome 8% menos energia que o melhor caso de OpenMP; 12% menos que MPI-1; e 8% menos que MPI-2, considerando todos os *benchmarks* no processador Intel Core i7 (propósito geral). Em contrapartida, no processador ARM Cortex-A9 (sistema embarcado), o melhor caso com OpenMP consumiu 2% menos energia que Pthreads; 6% menos que MPI-1; e 15% menos que MPI-2, para o mesmo conjunto de *benchmark*s.

Palavras-chave: Sistemas Embarcados, Processadores de Propósito Geral, Programação Paralela, Eficiência Energética

AVALIAÇÃO DO DESEMPENHO E CONSUMO ENERGÉTICO DE DIFERENTES INTERFACES DE PROGRAMAÇÃO PARALELA EM SISTEMAS EMBARCADOS E DE PROPÓSITO GERAL

ABSTRACT

In current computer systems, while it is necessary to exploit the availability of multiple cores, it is also mandatory to consume less energy. To accelerate the development of parallel applications and to make it more transparent to the programmer, Parallel APIs (Application Programming Interfaces) are widely used. However, each Parallel API implements different ways to exchange data using shared memory regions. These regions are generally more remote than the private ones, and therefore have greater access time and energy consumption. Furthermore, the memory system of embedded processors is different with regard to hierarchy, size, access time, energy consumption, etc., when compared to general purpose processors. Thus, considering the above scenario, with different Parallel APIs being used in multicore systems with different requirements, this work will show that each interface has different behavior in terms of performance, energy consumption and Energy-Delay Product (EDP), and that this behavior varies according to the characteristic of the application and the processor employed (general purpose or embedded). For example, as a result of this work, we have observed that Pthreads consumes 8% less energy than the best case of OpenMP; 12% less than MPI-1; and 8% less than MPI-2, considering all benchmarks on the Intel Core i7 (general purpose). In contrast, in the ARM Cortex-A9 processor (embedded system), the best case with OpenMP consumed 2% less energy than Pthreads; 6% less than MPI-1; and 15% less than MPI-2 for the same benchmarks set.

Keywords: Embedded Systems, General Purpose Processors, Parallel Programming, Energy Efficiency

LISTA DE FIGURAS

Figura 1.1- Comportamento da Paralelização em termos de Desempenho e Energia.............22
Figura 1.2 - Diferentes Organizações do Sistema de Memória...23
Figura 2.1- Exemplo de uma Arquitetura Multicore 28
Figura 2.2 - Organização e Características do Processador Intel Core i7 30
Figura 2.3- Organização e Característica do Processador ARM Cortex-A9...........................32
Figura 2.4 - Comunicação através de Variáveis Compartilhadas...34
Figura 2.5 - Exemplo de Troca de Dados em Memória Compartilhada.................................34
Figura 2.6 Exemplos de Operações de Comunicação 35
Figura 2.7 - Exemplo do Modelo Fork-Join do OpenMP..38
Figura 2.8 - Exemplo de Atribuição de Iterações ás Threads...39
Figura 2.9 - Exemplo de Comunicação entre Processos MPI-1 43
Figura 2.10 - Relacionamento Hierárquico resultante da Criação de um Único Processo.......45
Figura 4.1 - Decomposição de Domínio 1D...65
Figura 4.2 - Decomposição de Domínio 2D...66
Figura 4.3 - Organização Hierárquica dos Processadores Alvo 68
Figura 5.1 - Resultados CPU-B 75
Figura 5.2 - Resultados WMEM-B..77
Figura 5.3 - Resultados MEM-B 79
Figura 5.4 - Resultado de Speedup nas Aplicações CPU-B 83
Figura 5.5 - Resultado de Speedup nas Aplicações WMEM-B 84
Figura 5.6 - Resultado de Speedup nas Aplicações MEM-B 85
Figura 5.7 - Resultados de Eficiência do EDP – Aplicações CPU-B.......................................91
Figura 5.8 - Resultados de Eficiência do EDP – Aplicações WMEM-B 92
Figura 5.9 - Resultados de Eficiência do EDP – Aplicações MEM-B 93
Figura 5.10 - Resultados de Eficiência/Escalabilidade do Desempenho..................................95
Figura 5.11 - Resultado Médio de Eficiência/Escalabilidade do Desempenho por IPP...........96
Figura 5.12 - Resultados de Eficiência/Escalabilidade da Energia 98
Figura 5.13 - Resultado Médio de Eficiência/Escalabilidade da Energia por IPP 99
Figura 5.14 - Resultados de Eficiência/Escalabilidade do EDP 100
Figura 5.15 - Resultados Médio de Eficiência/Escalabilidade do EDP 102
Figura 5.16 – Melhor Eficiência do EDP em cada Processador...103
Figura 6.1 – Texas OMAP 5..108

Figura 6.2 - Exemplo de combinação de uma CPU com o Intel Xeon Phi 109
Figura 6.3 – Exemplo de Combinação de uma CPU com uma GPU 110
Figura 6.4 – Exemplo de SoC mobile heterogêneo 111
Figura 6.5 - Redução do Consumo de Energia em Arquiteturas Heterogêneas 112
Figura 6.6 – Exemplos de topologia NoC 113
Figura A.1 Algoritmo Sequencial Calculo do Pi 126
Figura A.2 Algoritmo Sequencial Conjunto de Mandelbrot 127
Figura A.3 Algoritmo Sequencial Série Harmônica 128
Figura A.4 Algoritmo Sequencial Dijkstra 129
Figura A.5 Algoritmo Sequencial Similaridade entre Histogramas 130
Figura A.6 Algoritmo Sequencial Decomposição-LU 131
Figura A.7 Algoritmo Sequencial Método de Jacobi 133
Figura A.8 Algoritmo Sequencial Multiplicação de Matriz 135
Figura A.9 Algoritmo Sequencial Jogo da Vida 136
Figura A.10 Algoritmo Sequencial Gram-Schmidt 138
Figura A.11 Algoritmo Sequencial Ordenação Par-Impar 140
Figura A.12 Algoritmo Sequencial Turing Ring 141
Figura B.1 – Resultados Cálculo do Número Pi (escala logarítmica) 142
Figura B.2 – Resultados Cálculo do Número Pi (escala logarítmica) - Continuação 143
Figura B.3 – Resultados Conjunto de Mandelbrot 144
Figura B.4 – Resultados Conjunto de Mandelbrot - Continuação 145
Figura B.5 – Resultados Série Harmônica 146
Figura B.6 – Resultados Série Harmônica - Continuação 147
Figura B.7 – Resultados Dijkstra 148
Figura B.8 – Resultados Dijkstra - Continuação 149
Figura B.9 – Resultados Similaridade entre Histogramas 150
Figura B.10 – Resultados Similaridade entre Histogramas - Continuação 151
Figura B.11 – Resultados Decomposição-LU 152
Figura B.12 – Resultados Decomposição-LU - Continuação 153
Figura B.13 – Resultados Método de Jacobi 154
Figura B.14 – Resultados Método de Jacobi - Continuação 155
Figura B.15 – Resultados Multiplicação de Matriz 156
Figura B.16 – Resultados Multiplicação de Matriz - Continuação 157
Figura B.17 – Resultados Jogo da Vida 158

Figura B.18 – Resultados Jogo da Vida - Continuação 159
Figura B.19 – Resultados Gram-Schmidt 160
Figura B.20 – Resultados Gram-Schmidt - Continuação 161
Figura B.21 – Resultados Ordenação Par-Impar 162
Figura B.22 – Resultados Par-Impar - Continuação 163
Figura B.23 – Resultados Turing Ring 164
Figura B.24 – Resultados Turing Ring - Continuação 165

LISTA DE TABELAS

Tabela 4.1 - Classificação dos Benchmarks 61
Tabela 4.2- Principais características dos processadores 67
Tabela 4.3- Dados de energia de cada processador 70
Tabela 5.1- Tamanho de Entrada de cada Benchmark 73
Tabela 5.2 - Diferença no Número de Instruções Executadas (baseline Core i7) 76
Tabela 5.3 – Tempo Total de Execução (em segundos) 81
Tabela 5.4 - Diferença no Número de Ciclos Executados (baseline Core i7) 82
Tabela 5.5 - Consumo Total de Energia (em Joules) 87
Tabela 5.6 - EDP Total (em Joules x segundos) – em milhares 90

SUMÁRIO

RESUMO **9**
ABSTRACT **11**
LISTA DE FIGURAS **13**
LISTA DE TABELAS **16**
SUMÁRIO **17**
1 INTRODUÇÃO **21**
1.1 Objetivos **23**
1.2 Organização do Texto **25**
2 PROGRAMAÇÃO PARALELA E ARQUITETURAS MULTICORE **27**
2.1 Arquiteturas *Multicore* **28**
2.1.1 Processadores de Propósito Geral 29
2.1.2 Processadores de Baixo Consumo Energético 31
2.2 Modelos de Comunicação **33**
2.2.1 Variáveis Compartilhadas 33
2.2.2 Trocas de Mensagens 35
2.3 Interfaces de Programação Paralela **36**
2.3.1 OpenMP 37
2.3.2 POSIX Threads 39
2.3.3 *Message-Passing Interface* 41
2.3.3.1 MPI-1 42
2.3.3.2 MPI-2 44
2.4 Conclusão do Capítulo **46**
3 TRABALHOS CORRELATOS **47**
3.1 Comparação entre Interfaces de Programação Paralela **47**
3.1.1 Facilidade de Programação 47
3.1.2 Controle e Flexibilidade 50
3.1.3 Custo Extra Imposto pela Paralelização 52
3.1.4 Desempenho 54
3.1.5 Consumo de Energia 56
3.2 Contexto desta Dissertação **58**
4 METODOLOGIA **60**
4.1 Conjunto de *Benchmarks* **60**
4.1.1 *CPU-Bound* (CPU-B) 61
4.1.2 *Weakly Memory-bound* (WMEM-B) 62
4.1.3 *Memory-bound* (MEM-B) 63
4.2 Paralelização do Conjunto de *Benchmark* **64**
4.3 Ambiente de Execução **66**
4.3.1 Arquiteturas Alvo 66
4.3.2 Extração de Dados e Métricas Analisadas 68
5 RESULTADOS **73**
5.1 Número de Instruções Executadas e Acessos a Memória de Dados **74**

5.1.1 Análise das Aplicações CPU-B 74
5.1.2 Análise das Aplicações WMEM-B 76
5.1.3 Análise das Aplicações MEM-B 78
5.1.4 Análise Crítica 80
5.2 Desempenho 80
5.2.1 Análise das Arquiteturas 80
5.2.2 Análise das Interfaces de Programação Paralela 82
5.3 Consumo de Energia 86
5.3.1 Análise das Arquiteturas 86
5.3.2 Análise das Interfaces de Programação Paralela 88
5.4 *Energy-Delay Product* 90
5.4.1 Análise das Arquiteturas 90
5.4.2 Análise das Interfaces de Programação Paralela 91
5.5 Escalabilidade 93
5.5.1 Desempenho 94
5.5.2 Consumo de Energia 97
5.5.3 *Energy-Delay Product* 99
5.5.4 Análise Crítica 102
6 CONCLUSÕES E TRABALHOS FUTUROS 105
6.1 Trabalhos Futuros 106
6.1.1 Expansão do Conjunto de *Benchmark* 106
6.1.2 Análise de Diferentes Compiladores 107
6.1.3 Impacto das Boas Práticas de Programação 107
6.1.4 Investigar o Impacto do uso de Comunicações Coletivas 107
6.1.5 Arquiteturas Heterogêneas 107
6.1.5.1 Intel Xeon Phi 109
6.1.5.2 Unidades de Processamento Gráfico 110
6.1.5.3 Tecnologia big.LITTLE 111
6.1.5.4 Dynamic Voltage and Frequency Scaling - DVFS 112
6.1.5.5 Network-on-Chips 112
REFERÊNCIAS 114
APÊNDICE A – ESTRATÉGIA DE PARALELIZAÇÃO 126
A.1 Cálculo do Pi 126
A.2 Conjunto de Mandelbrot 127
A.3 Série Harmônica 128
A.4 Dijkstra 129
A.5 Similaridade entre Histogramas 130
A.6 Decomposição-LU 130
A.7 Método de Jacobi 133
A.8 Multiplicação de Matrizes 135
A.9 Jogo da Vida 136
A.10 Gram-Schmidt 138
A.11 Ordenação Par-Impar 140
A.12 Turing Ring 141

APÊNDICE B – RESULTADOS OBTIDOS EM CADA APLICAÇÃO............142
B.1 Cálculo do Número Pi 142
B.2 Conjunto de *Mandelbrot*............144
B.3 Série Harmônica 146
B.4 *Dijkstra*............148
B.5 Similaridade entre Histogramas............150
B.6 Decomposição-LU............152
B.7 Método de Jacobi............154
B.8 Multiplicação de Matriz............156
B.9 Jogo da Vida............158
B.10 Gram-Schmidt 160
B.11 Ordenação Par-Impar............162
B.12 *Turing Ring*............164

1 INTRODUÇÃO

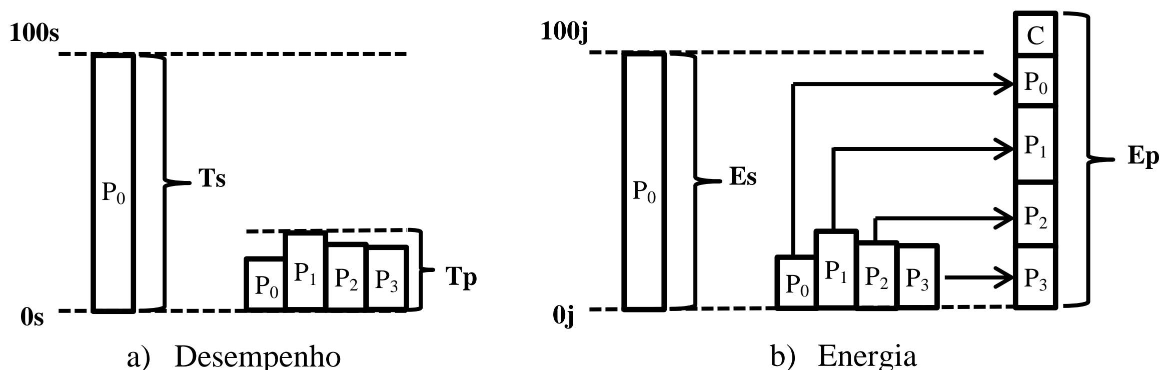
Nos últimos anos, com o aumento da complexidade das aplicações para sistemas embarcados, que demandam maior eficiência computacional; e a chegada da computação *exascale*, com poder de processamento 100 vezes maior que os computadores atuais (CAPPELLO, 2009), surge a necessidade de aumentar o desempenho com o menor impacto possível no consumo de energia. Enquanto que a maioria dos sistemas embarcados é representada pelos dispositivos móveis e dependentes de bateria (*smartphones, tablets*, etc.), os computadores *exascale* poderão chegar ao consumo de energia que corresponde à potência fornecida por uma usina nuclear de médio porte (WEHNER, 2009). Portanto, o desafio não deve ser somente aumentar o desempenho, mas também consumir o mínimo possível de energia.

Neste ímpeto de aumento de desempenho, observa-se a popularização de sistemas embarcados que utilizam múltiplos núcleos (*multicore*). A computação paralela explora o uso de múltiplos processadores para executar partes diferentes de um mesmo programa simultaneamente. Todavia, para que esta cooperação ocorra, os processadores deverão trocar informações em determinado momento da execução. Esta troca ocorre através do acesso simultâneo a regiões compartilhadas da memória. Entretanto, este acesso se dá em regiões da memória que estão hierarquicamente mais distantes do processador (i.e.: 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, 2010).

Desta forma, enquanto a paralelização possibilita aumentar o desempenho dos sistemas computacionais, a necessidade de comunicação entre os processadores pode levar a um maior consumo de energia. Por exemplo, o subsistema de memória pode ser responsável por até 80% do consumo total de energia dos sistemas embarcados (JI, 2008). Para melhor exemplificar esta situação, a Figura 1.1 apresenta uma análise do comportamento de desempenho e consumo de energia considerando a execução paralela de uma aplicação em um processador *multicore* com quatro núcleos (P_0, P_1, P_2 e P_3).

Na maioria das vezes, uma aplicação paralelizada obtém ganho de desempenho com relação à versão sequencial, pois o tempo total de computação é distribuído entre os processadores. Por exemplo, na Figura 1.1a, 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

Figura 1.1- Comportamento da Paralelização em termos de Desempenho e Energia



comportamento não acontece no consumo de energia (Figura 1.1b), 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.

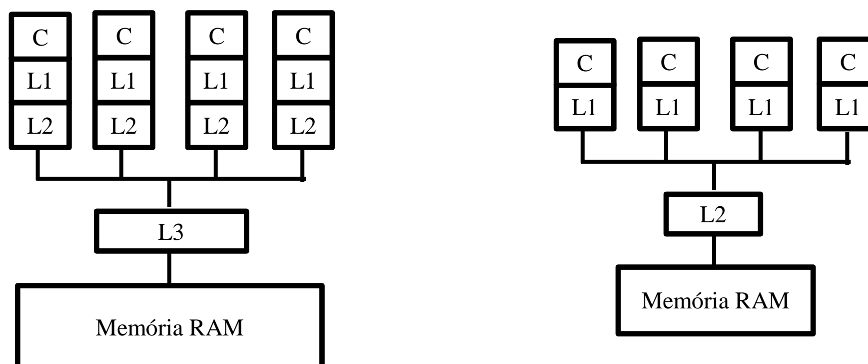
Assim, o ganho de desempenho e o consumo de energia podem variar dependendo de fatores como:

- Arquitetura do processador e organização hierárquica da memória: processadores *multicore* de propósito geral possuem arquitetura e sistema de memória maior e mais robusto que processadores *multicore* para sistemas embarcados. A Figura 1.2 mostra um exemplo, em que o processador para propósito geral Intel Core i7 (Figura 1.2a) possui três níveis de *cache* (L1 e L2 privadas para cada *core*, e L3 compartilhada entre todos os *cores*) e memória principal de 8 GB, enquanto que o processador para sistemas embarcados ARM Cortex-A9 (Figura 1.2b) possui apenas dois níveis de *cache* (L1 privada para cada *core* e L2 compartilhada entre todos os *cores*) e memória principal de apenas 1 GB.

- Modelo de comunicação: a comunicação nos processadores *multicore* pode ocorrer acessando a memória compartilhada entre todos os processadores de duas formas: variáveis compartilhadas, baseada na existência de uma memória global que pode ser acessada por todos os processadores e a troca de dados acontece através de acessos simultâneos a estas regiões de memória; e troca de mensagens, em que os processadores, em alto nível, não compartilham o mesmo endereço de memória e a comunicação se dá através do uso de operações de envio e recebimento de mensagens.

- Interfaces de Programação Paralela (IPP): utilizadas para facilitar a tarefa de comunicação e exploração do paralelismo, além de diminuir o tempo de desenvolvimento de

Figura 1.2 - Diferentes Organizações do Sistema de Memória



a) Intel Core i7 b) ARM Cortex-A9

aplicações paralelas e o tornar menos propenso a erros. No entanto, cada uma delas possui suas particularidades com relação à forma de exploração do paralelismo e comunicação. Por exemplo, enquanto em OpenMP o paralelismo é explorado através da inserção de diretivas de compilação no código sequencial e comunicação é dada através de variáveis compartilhadas; em MPI, o paralelismo é explícito, ou seja, o programador é responsável por dividir e atribuir a carga de trabalho entre os processadores, além de prever a comunicação através de troca de mensagens.

- Características da aplicação: aplicações com uso intensivo da CPU e com pouca comunicação entre os processadores podem apresentar melhor desempenho e utilização da memória *cache* e, assim, obter consumo de energia similar ao consumo da versão sequencial. Por outro lado, aplicações com grande quantidade de comunicação/sincronização geralmente consomem mais energia que a versão sequencial (PORTERFIELD et al., 2013).

1.1 Objetivos

Considerando o cenário apresentado na Seção anterior, infere-se que embora a paralelização permita ganhos de desempenho, pode levar a um maior consumo energético, já que, para que a cooperação ocorra, faz-se necessário 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 Interfaces de Programação

Paralela, as quais cada uma tem suas particularidades em termos de sincronização e comunicação.

Assim, este trabalho irá mostrar que a eficiência das aplicações paralelas muda conforme a Interface de Programação Paralela utilizada, o grau de paralelismo explorado, a característica da aplicação e o tipo de processador utilizado (por exemplo: propósito geral ou sistema embarcado). Fatores como o número total de acessos a memória e o comportamento da aplicação irão influenciar no tempo de execução, consumo de energia e *Energy-Delay Product* (EDP). Isto significa que o programador deve estar ciente de que a melhor Interface de Programação Paralela a ser utilizada não é só dependente do tipo de aplicação, mas também do processador. Portanto, os objetivos deste trabalho correspondem a:

- Analisar o comportamento de diferentes Interfaces de Programação Paralela em termos de instruções executadas e acessos a memória de dados considerando processadores embarcados e de propósito geral.

- Analisar o impacto no desempenho, consumo de energia e eficiência energética de diferentes Interfaces de Programação Paralela considerando processadores embarcados e de propósito geral;

- Considerando o melhor desempenho obtido em cada Interface de Programação Paralela em cada processador, verificar se a escalabilidade (i.e.: comportamento quando o número de processadores aumenta em relação à execução sequencial) em termos de desempenho, consumo de energia e eficiência energética é a mesma em ambos os tipos de processadores.

- Mostrar que diferentes Interfaces de Programação Paralela possuem diferentes impactos na eficiência energética e que o impacto pode variar conforme o nicho de aplicação, a arquitetura e a organização utilizada.

Para tanto, implementou-se um conjunto de doze aplicações classificadas em três grupos de acordo com a necessidade de comunicação, pontos de sincronização e dependência de dados. Todas elas foram implementadas com três populares Interfaces de Programação Paralela: OpenMP, POSIX Threads, MPI (-1 e -2). Foram utilizados quatro processadores com diferentes características: Intel Core i7 e Core2Quad representando os processadores de propósito geral; e Intel Atom e ARM Cortex-A9 representando os processadores para sistemas embarcados.

1.2 Organização do Texto

Esta dissertação está organizada como segue:

No Capítulo 2, as arquiteturas *multicore* são discutidas com ênfase na tarefa de comunicação entre os processos e nas principais diferenças entre processadores de propósito geral e de sistemas embarcados. A seguir são apresentados os modelos de comunicação utilizados pela programação paralela nestas arquiteturas. Também são contextualizadas as principais características das Interfaces de Programação Paralela alvo deste trabalho.

O Capítulo 3 mostra os trabalhos desenvolvidos que são relacionados a esta dissertação. Eles são divididos em trabalhos que avaliam a facilidade do desenvolvimento de códigos paralelos; controle e flexibilidade na exploração do paralelismo; custo extra, imposto pelo gerenciamento (criação/finalização) de *threads*/processos e comunicação entre os mesmos; desempenho obtido com cada Interface de Programação Paralela; e os trabalhos relacionados ao consumo de energia de tais Interfaces. Ademais, o escopo deste trabalho é apresentado, destacando as principais contribuições com relação aos trabalhos já realizados.

O Capítulo 4 descreve a metodologia utilizada no desenvolvimento deste trabalho. Inicialmente, o conjunto de *benchmarks* escolhido é apresentado, juntamente com as principais características de cada aplicação. A seguir, a estratégia de paralelização adotada em cada aplicação para cada Interface de Programação Paralela é discutida. Contextualiza-se o ambiente de execução, destacando as principais características de cada processador utilizado. Também são apresentadas as métricas utilizadas na análise: desempenho, consumo de energia, *energy-delay product* e escalabilidade.

Os resultados obtidos são apresentados e discutidos no Capítulo 5. Para tanto, inicialmente são apresentados os resultados para o número de instruções executadas e a quantidade de acessos à memória. A seguir o desempenho obtido e o consumo de energia gasto em cada cenário de execução são discutidos. O impacto dos resultados anteriores no *energy-delay product* e na escalabilidade das aplicações são discutidos ao final do Capítulo.

O Capítulo 6 apresenta as conclusões obtidas com este trabalho e reforça a contribuição do mesmo. Também, trabalhos futuros são discutidos.

2 PROGRAMAÇÃO PARALELA E ARQUITETURAS MULTICORE

De um modo geral, a programação paralela consiste na divisão de tarefas de uma aplicação com a finalidade de reduzir seu tempo total de execução (RAUBER, 2010). 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. Ademais, com a popularização das arquiteturas *multicore*, as aplicações de uso geral (filtros de imagens e som, editores gráficos, servidores de internet, etc) também têm tirado proveito da programação paralela.

Tradicionalmente, os programas são desenvolvidos para serem computados sequencialmente, onde uma instrução é executada após a outra. Entretanto, a execução paralela de instruções sempre pode ser explorada. Arquiteturas superescalares podem executar simultaneamente instruções independentes dentro de um programa, desde que haja unidades funcionais suficientes para tal. Arquiteturas superescalares, então, conseguem extrair o paralelismo em uma granularidade mais fina no nível de instruções (ILP - *Instruction-Level Parallelism* – Paralelismo no Nível de Instrução).

Esta mesma lógica pode ser aplicada para grupos de instruções (e.g.: procedimentos), 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 – *Thread Level Parallelism* – Paralelismo no Nível de *Threads*). Para viabilizar isso, são utilizadas múltiplas unidades de processamento (núcleos), presentes nas arquiteturas *multicore*. Enquanto que a exploração de TLP em sistemas *multicore* de propósito geral vem sendo feita há alguns anos, apenas recentemente ela tem sido explorada em sistemas embarcados. Assim, a Seção 2.1 contextualiza sobre as arquiteturas *multicore*, evidenciando as principais diferenças entre processadores *multicore* para propósito geral e para sistemas embarcados.

Para que a exploração de TLP ocorra, em determinados momentos da execução é necessário prover troca de informações. Esta comunicação pode ser realizada através de dois modelos: variáveis compartilhadas e troca de mensagens. Ambas serão discutidas com mais detalhes na Seção 2.2. Para facilitar o processo de exploração de TLP e o tornar menos propenso a erros, diferentes Interfaces de Programação Paralela têm sido desenvolvidas, as quais serão apresentadas na Seção 2.3.

2.1 Arquiteturas *Multicore*

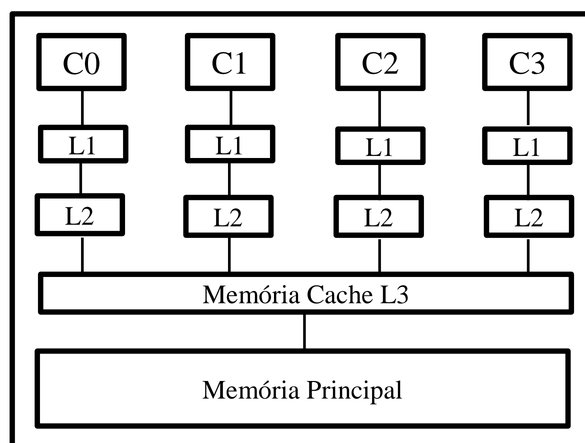
Buscando aumentar o desempenho e tirar vantagem do número crescente de transistores disponíveis devido à melhora no processo de manufatura de circuitos integrados, a indústria passou a investir na tecnologia *multicore* como uma alternativa às limitações na exploração do ILP que processadores do tipo superescalar demostravam; além de também se apresentar como uma solução para a alta potência consumida por estes.

Arquiteturas *multicore* caracterizam-se por possuírem múltiplas unidades de processamento, compartilhando acesso a um mesmo espaço de endereçamento (HENNESSY, 2012). Tais 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.

Entre os desafios enfrentados na concepção das arquiteturas *multicore*, um dos mais relevantes está relacionado ao acesso aos dados. Tradicionalmente, processadores com um único núcleo já incluíam níveis de memória *cache intra-chip* para diminuir a latência média de acesso à memória. Entretanto, quando múltiplos núcleos de processamento estão presentes, o problema se agrava, já que é necessária a comunicação entre estes núcleos. Assim, é necessário adaptar a organização da arquitetura incluindo níveis de memória compartilhada.

Uma representação esquemática de uma arquitetura *multicore* é apresentada na Figura 2.1. Neste exemplo, a arquitetura possui quatro núcleos de processamento (representados por C0, C1, C2 e C3). Cada núcleo possui memórias *cache* L1 e L2 privadas, ou seja, estes dois níveis de memória só poderão ser acessados pelo núcleo em questão. Quando há a

Figura 2.1- Exemplo de uma Arquitetura *Multicore*



necessidade de comunicação entre os processadores, níveis superiores da hierarquia de memória precisam ser acessados, os quais possuem maior custo (tempo e potência) por acesso (JI, 2008). Neste exemplo, tais níveis são representados pela memória *cache* L3 e a principal.

Como pode ser observado na Figura 2.1, o subsistema de memória *cache* ocupa uma área significativa numa arquitetura *multicore*, já que ela tem papel fundamental na determinação de seu desempenho: sua capacidade e velocidade determinam a vazão de dados disponibilizados aos *cores*. Portanto, reduzir a quantidade de comunicação necessária e melhorar a utilização das memórias *cache* possibilita maiores ganhos de desempenho e economia no consumo de energia.

Nota-se, então, que a memória compartilhada representa o gargalo da comunicação entre os processadores. Este gargalo aumenta quando existe dependência de dados entre as *threads*. Por exemplo, muitas vezes, a *thread* que foi executada em um processador depende do resultado de outra *thread*, que está sendo executada em outro processador. Quando isto acontece, a primeira *thread* fica bloqueada, isto é, esperando o dado ficar pronto para continuar a desempenhar suas funções. Portanto, quanto maior a necessidade de comunicação entre as *threads*, maior é este problema e, por consequência, maior é o tempo e energia consumida apenas para a comunicação.

Nos últimos anos, devido a popularização dos sistemas embarcados *multicore*, as arquiteturas *multicore* podem ser divididas em duas grandes classes: processadores de propósito geral e para sistemas embarcados. As principais diferenças entre estes sistemas estão relacionadas à restrição do consumo de energia por parte dos processadores e ao sistema de memória.

2.1.1 Processadores de Propósito Geral

Os processadores de propósito geral são encontrados em computadores como *desktops* ou estações de trabalho. Estes computadores são frequentemente utilizados no dia-a-dia e são desenvolvidos para manter compatibilidade de *software*, ou seja, executar qualquer tipo de *software*, sem demonstrar preocupação quanto ao consumo de energia. Estes processadores são representados, principalmente, pela ISA (*Instruction Set Architecture* – Conjunto de Instruções da Arquitetura) Intel x86.

Os primeiros processadores *multicore* de propósito geral foram desenvolvidos a partir de 2004. Inicialmente, a empresa AMD demonstrou o primeiro processador contendo dois

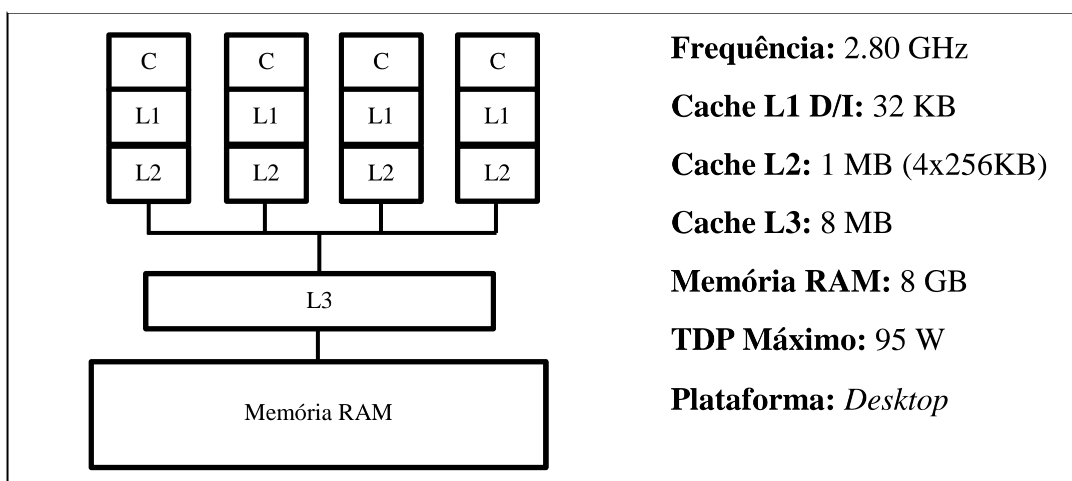
núcleos. A Intel introduziu seu primeiro processador *multicore* em meados de 2005, chamando-o de Pentium-D, que continha dois núcleos. A partir de então, o foco da indústria foi desenvolver novos processadores com mais núcleos em um chip.

Tal evolução chega aos dias de hoje com os processadores que podem ter até oito núcleos (e.g.: Intel *Xeon*). Estes processadores possuem frequências de operação que alcançam 3.7 GHz, que mostra o potencial de desempenho que eles oferecem. Por exemplo, a Figura 2.2 apresenta a organização da hierarquia da memória e as principais características do processador Intel Core i7.

Este processador (Core i7) tem sido amplamente utilizado em sistemas *desktop* e possui quatro núcleos de processamento (representados pela letra C) com frequência de 2.80 GHz cada. Seu suporte a *Hyper-Threading* (nome dado pela Intel para a técnica *Simultaneous Multi-Threading* (EGGERS et al., 1997)) possibilita a execução de até 8 *threads* simultâneas (duas *threads* em cada processador). O sistema de memória é composto por três níveis de *cache* (L1 separada em dados e instruções, e L2 privadas à cada *core*; e L3 compartilhada entre todos os *cores*); e uma memória principal (RAM) compartilhada entre todos os *cores*.

Conforme Molka (2009), para um processador *Intel Core i7* similar ao apresentado na Figura 2.2, o tempo de um acesso à memória *cache* L1 e L2 corresponde a 1.3 *ns* e 2.4 *ns* respectivamente. Já para as memórias compartilhadas, *cache* L3 e principal, o tempo de acesso sobe para 13 *ns* e 65.1 *ns* respectivamente. Isto implica que o acesso à memória principal é aproximadamente 50 vezes mais lento que o acesso a *cache* L1. Quando é considerada a energia consumida em cada acesso a memória, esta diferença é ainda maior. Por exemplo, conforme Cacti (2013) e Ji (2008), enquanto que um acesso de leitura a *cache* L1 consome aproximadamente 0,014 *nJ*, o acesso de leitura a memória principal consome

Figura 2.2 - Organização e Características do Processador Intel Core *i7*



aproximadamente 800 vezes mais (~15 *nJ*).

Outra característica a ser destacada nestes processadores é o TDP (*Thermal Design Power*) máximo, que representa a quantidade máxima de energia que o processador poderá dissipar. Recentemente, algumas versões dos processadores Intel Core i5 têm sido desenvolvida buscando diminuir o TDP máximo para aproximadamente 35W. No entanto, devido a serem voltados para o desempenho, a maioria dos processadores de propósito geral possuem TDP que varia de 95 W (Intel Core i7) até 150 W nos processadores utilizados para alto desempenho, como o Intel *Xeon*.

2.1.2 Processadores de Baixo Consumo Energético

Os processadores de baixo de consumo de energia têm sido amplamente utilizados em sistemas embarcados (e.g.: *tablets* e *smartphones*), visto que são alimentados por baterias. Atualmente, o principal desenvolvedor de projetos de *chip* para sistemas embarcados é a empresa ARM.

ARM é uma sigla para *Acorn RISC Machine*. Este nome é proveniente da predecessora da empresa que deu origem a ARM: a *Acorn Computers Ltd.* O projeto inicial deste processador surgiu em 1983. O primeiro chip produzido – o ARM1 – foi lançado em 1985. Entretanto, o primeiro processador da empresa a ter real penetração no mercado foi o ARM2, de 1986. Desde então, a ARM tornou-se líder de processadores de sistemas embarcados. Tem mais de 900 empresas parceiras, com mais de 25 bilhões de processadores vendidos.

A constante evolução dos seus processadores ao longo do tempo resultou no desenvolvimento de sistemas *multicore* através da família de processadores Cortex A. Ela representa o topo de linha dos processadores ARM, e são utilizados por todos aqueles sistemas embarcados de última geração que necessitam o máximo de desempenho (como as várias versões do *Apple iPhone* ou *Samsung Galaxy*). Atualmente, a família Cortex é composta de seis versões: A5, A7, A9, A12, A15 e A17. Todas as versões do Cortex A são geralmente vendidas no formato *multicore*. O A5 é o mais simples de todos, com implementações de frequência entre 400 e 800 MHz. Já o A17 é mais eficiente em termos de desempenho, utilizado em uma gama de dispositivos que varia de sistemas móveis até *Smart TVs*.

A Figura 2.3 apresenta a organização da hierarquia de memória do processador ARM Cortex-A9 e suas principais características. Este processador, com frequência de operação de

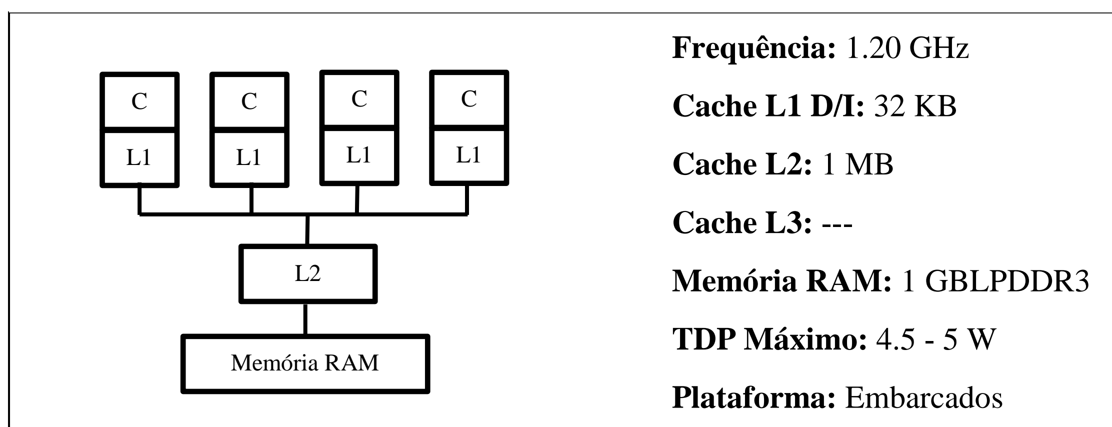
1.20 GHz, possui dois níveis de cache (L1 privada para cada *core*, separada em instruções e dados; e L2 unificada e compartilhada entre todos os *cores*), e uma memória principal (RAM) de 1GB desenvolvida para baixo consumo de energia. Pode-se notar que as principais diferenças com relação aos processadores de propósito geral (apresentadas na Figura 2.2), estão relacionadas à redução da configuração de componentes responsáveis por consumir mais energia. Embora a microarquitetura do ARM seja um dos fatores que contribua para o baixo consumo de energia, ele também está relacionado a redução da frequência de operação e alteração da hierarquia de memória.

Nos últimos anos, a Intel tem se voltado para o desenvolvimento de processadores de baixo consumo de energia, através do Intel Atom. Ele corresponde à linha de microprocessadores x86 da Intel projetados para utilização em dispositivos móveis. Dentre suas principais características está a compatibilidade com o conjunto de instruções X86 (presente nos processadores Intel Xeon, Core, etc.), que permite que as aplicações compiladas para os processadores *desktop* também executem no Atom.

As primeiras versões *multicore* do Atom começaram a ser comercializadas no ano de 2008 através da Série Atom 3xx. Atualmente, as séries N2000 e D2000 representam o Atom nos sistemas embarcados *multicore* (INTEL 2011). Tais processadores possuem dois *cores* que executam duas *threads* simultaneamente através de *Hyper-Threading*. A frequência de operação destes processadores é em torno de 1.66 GHz e 1.80 GHz, dependendo do modelo.

Os processadores para sistemas embarcados possuem TDP máximo que varia em torno de 0.8 e 10 W, o que representa menos de 10% do consumo máximo dos processadores de propósito geral (Intel Core i7 com TDP de 95 W (INTEL 2010)). Outra diferença significante está no sistema de memória. Dispositivos móveis normalmente possuem memórias desenvolvidas com tecnologia *Low Power* (e.g.: LPDDR, LPDDR2, LPDDR3). Conforme dados obtidos através da simulação de tais memórias no Cacti Tool (CACTI, 2013), a

Figura 2.3- Organização e Característica do Processador ARM Cortex-A9



diferença de consumo entre uma memória RAM DDR3 para uma LPDDR3 chega a ser de 7 vezes (15,6 *nJ* para 2,4 *nJ*).

2.2 Modelos de Comunicação

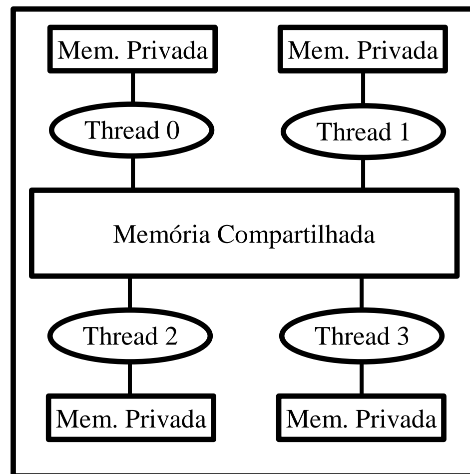
A computação paralela explora o uso de múltiplos processadores para executar partes diferentes de um mesmo programa simultaneamente. Assim, há a cooperação entre os processadores que executam de forma concorrente. Todavia, para que esta cooperação ocorra, os processadores deverão trocar informações em tempo de execução. Conforme Rauber et al. (2010), esta troca de informações pode ser realizada através de variáveis compartilhadas e/ou operações de comunicação.

2.2.1 Variáveis Compartilhadas

Este modelo é baseado na existência de uma memória global, que pode ser acessada por todos os processadores. Ele é amplamente utilizado quando o paralelismo é explorado ao nível das *threads*, pois elas compartilham o mesmo espaço de endereçamento da memória. Assim, as *threads* poderão ter variáveis privadas (a *thread* possui acesso exclusivo) e compartilhadas (todas as *threads* possuem acesso). Conforme podemos acompanhar na Figura 2.4, as variáveis privadas estão situadas em regiões privadas da memória (Mem. Privada). Quando é necessário trocar informações entre as *threads*, utiliza-se variáveis compartilhadas, que estão localizadas em regiões da memória que são acessadas por todas as *threads* (Memória Compartilhada).

Para controlar e coordenar o acesso às variáveis compartilhadas por múltiplas *threads*, operações são utilizadas para garantir que os acessos concorrentes a mesma variável ocorram de forma sincronizada. Por exemplo, a Figura 2.5 apresenta dois exemplos de comunicação entre duas *threads*. Na primeira situação, (Figura 2.5a), cada *thread* computará a função `soma()` e armazenará o resultado em uma variável privada, chamada de `SPriv`. Após, cada

Figura 2.4 - Comunicação através de Variáveis Compartilhadas



thread irá atualizar o valor da variável compartilhada `SComp` e seguir seu fluxo de execução. No entanto, a atualização simultânea da variável `SComp` pode gerar inconsistência no resultado final, pois as duas *threads* podem tentar acessar e alterar o valor de `SComp` ao mesmo tempo, caracterizando uma seção crítica.

Para que o resultado final da execução permaneça correto, somente uma *thread* pode acessar o endereço desta variável a cada momento. Este comportamento é chamado de exclusão mútua e possui dois estados: bloqueado, que significa que já existe uma *thread* acessando a região crítica; e desbloqueado, que indica que uma nova *thread* pode acessar a região crítica. Um exemplo da utilização de exclusão mútua é apresentado na Figura 2.5b, com duas *threads*. Nela, foram inseridas duas operações de sincronismo: `inicioSecaoCritica` e `fimSecaoCritica`, que indicam respectivamente o início e fim de uma seção crítica. Assim, quando uma *thread* estiver acessando e atualizando a variável `SComp`, a outra *thread* ficará aguardando o desbloqueio da seção crítica, para então acessá-la.

Figura 2.5 - Exemplo de Troca de Dados em Memória Compartilhada

```
01. #define NUM_THREADS 2
02.
03. main(){
04.     ...
05.     SPriv = soma(idThread);
06.     SComp = SComp + SPriv;
07.     ...
08. }
```

a) Sem ponto de sincronização

```
01. #define NUM_THREADS 2
02.
03. main(){
04.     ...
05.     SPriv = soma(idThread);
06.     inicioSecaoCritica
07.        SComp = SComp + SPriv;
08.     fimSecaoCritica
09.     ...
10. }
```

b) Com ponto de sincronização

2.2.2 Trocas de Mensagens

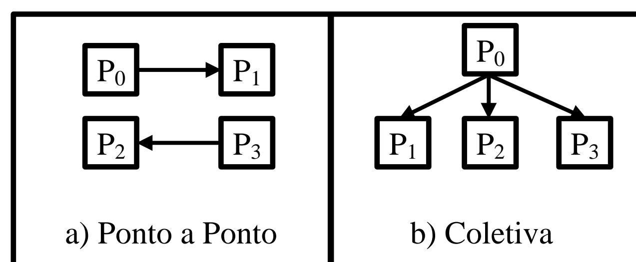
Nos ambientes onde o espaço de memória é distribuído e/ou onde os processos não compartilham o mesmo endereçamento de memória, a cooperação ocorre através do uso de operações de envio e recebimento de mensagens. Este modelo de comunicação é amplamente utilizado quando o paralelismo é explorado no nível de processos. Isto porque processos possuem endereçamento próprio de memória, em que um não possui acesso à região de memória de outros processos.

As comunicações ocorrem através de operações do tipo *send/receive* e podem ser do tipo ponto a ponto ou coletivas, conforme ilustradas na Figura 2.6. Nas operações ponto a ponto, a comunicação acontece entre pares de processos. No exemplo da Figura 2.6a, o processo P_0 envia uma mensagem para P_1. Assim, P_0 executa uma operação do tipo *send* enquanto que P_1 uma operação do tipo *receive*. O mesmo cenário ocorre quando P_3 envia uma mensagem para P_2.

As operações coletivas são utilizadas quando três ou mais processos precisam trocar informações. A Figura 2.6b apresenta um exemplo de comunicação coletiva chamada *broadcast*. Nesta operação, um único processo (P_0) envia mensagem para os demais processos (P_1, P_2 e P_3). Conforme Rauber et al. (2010), existem outros tipos de comunicação coletiva que possibilitam refinar a troca de mensagens entre os processos: *single-accumulation*, *scatter*, *gather*, *multi-broadcast*, *multi-accumulation* e *total exchange*. No entanto, elas não serão abordadas no escopo deste trabalho.

A troca de mensagens pode ser realizada através de operações síncronas ou assíncronas. Nas operações síncronas, os processos envolvidos necessariamente devem estar sincronizados entre si antes da comunicação iniciar e permanecem sincronizados durante a transmissão da mensagem. Quando o envio é feito, o processo remetente é bloqueado, ou seja, fica em estado de espera até que o receptor confirme o recebimento da mensagem. Em muitos casos, estas operações são também utilizadas como pontos de sincronização entre processos

Figura 2.6 Exemplos de Operações de Comunicação



durante a execução da aplicação. Já nas operações assíncronas, os processos não precisam estar sincronizados entre si para a comunicação ocorrer e elas podem ser divididas entre envio e recebimento assíncrono. No envio assíncrono, os dados são transferidos para um *buffer* e serão entregues no momento em que o processo receptor solicitar. Isto faz com que enquanto os dados estão sendo transmitidos, o processo receptor pode computar algo útil ao invés de ficar bloqueado aguardando a confirmação da conclusão da comunicação. O mesmo ocorre no recebimento assíncrono, porém o processo receptor faz uma chamada de recebimento assíncrono e pode computar enquanto essa mensagem não é recebida. Quando ela é recebida, ele confirma o recebimento para o processo remetente.

2.3 Interfaces de Programação Paralela

O desenvolvimento de aplicações (ou programas) capazes de explorar todo o paralelismo potencial das arquiteturas multiprocessadas depende de uma série de aspectos específicos da organização destas arquiteturas, entre eles, o tamanho e a estrutura hierárquica da memória. Os Sistemas Operacionais fornecem certa transparência com relação à existência de múltiplos núcleos e responsabilizam-se pela distribuição das tarefas solicitadas pelos usuários entre os núcleos disponíveis. Entretanto, contar apenas com o Sistema Operacional, em muitos casos, pode levar a um desperdício do poder computacional disponível. Por exemplo, estar executando apenas um processo de uma aplicação que poderia ser dividida em vários processos em um sistema *multicore* de 8 núcleos.

O processo de comunicação entre processos/*threads* apresentados na Seção 2.2 e a extração do paralelismo não são transparentes ao programador. Isto é, é necessário que o mesmo indique, explicitamente, regiões do programa que podem ser executadas em paralelo, e como será feita a troca de dados: distribuição dos dados de entrada entre as partes paralelas; comunicações durante a computação paralela; e composição da solução final a partir dos resultados parciais computados nas partes paralelas. Para facilitar esta tarefa, com o intuito de diminuir o tempo de desenvolvimento e o deixar menos propenso a erros, IPPs têm sido utilizadas: OpenMP (CHAPMAN et. al. 2008), Pthreads (BUTENHOF, 1997), Intel Cilk++ (LEISERSON, 2010), Intel *Threading Building Blocks* (ROBISON, 2011) para comunicação em memória compartilhada; MPI (GROPP, 1998) e PVM (MICHIELSE, 1994) para memória distribuída. No entanto, no contexto deste trabalho, serão avaliadas as IPPs que estão sendo amplamente utilizadas pela comunidade acadêmica e que não possuem restrição quanto à

arquitetura utilizada (por exemplo, Intel Cilk++ e Intel TBB não possuem suporte para processadores ARM). Assim, as subseções seguintes descrevem com maiores detalhes as interfaces OpenMP, Pthreads e MPI.

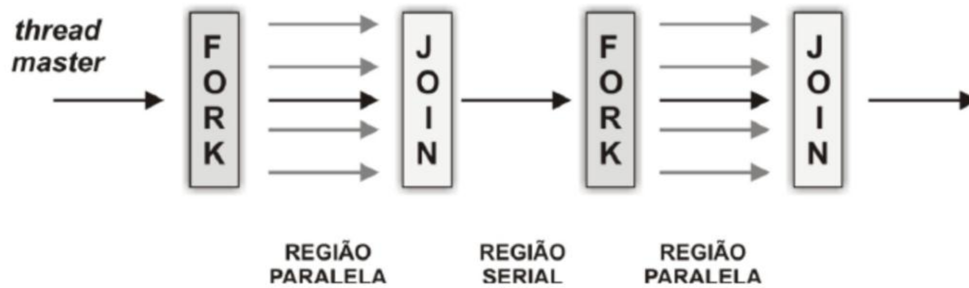
2.3.1 OpenMP

O OpenMP é uma API multi-plataforma para processamento paralelo baseado em memória compartilhada para as linguagens C/C++ e Fortran. Ele consiste em um conjunto de diretivas para o compilador, funções de biblioteca e variáveis de ambiente. Essa API foi especificada por um grupo dos grandes fabricantes de *hardware*/*software* com o intuito de ser portável e escalável, com uma interface de utilização simples e que pudesse ser utilizada tanto para aplicações de grande porte, quanto para aplicações *desktop* (CHAPMAN et. al. 2008).

O paralelismo é explorado através do uso de diretivas de compilação, que possuem significado especial para o compilador. Estas diretivas informam ao compilador as regiões do código que deverão ser executadas em paralelo. Explicadas a seguir, elas são divididas em construtor paralelo, construtores de compartilhamento de trabalho e diretivas de sincronização.

O **construtor paralelo** é a diretiva mais importante do OpenMP, pois informa ao compilador a região do código que será executada em paralelo. Este construtor segue o modelo *Fork*/*Join*, apresentado na Figura 2.7. Nele, inicialmente existe um fluxo de execução principal, chamado de *thread master*. Quando esta *thread* encontrar o construtor paralelo, ela cria um novo grupo de *threads* (*Fork*). No final da região paralela existe 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 master* continua a execução do código (TERUEL et al. 2009).

Figura 2.7 - Exemplo do Modelo *Fork-Join* do OpenMP



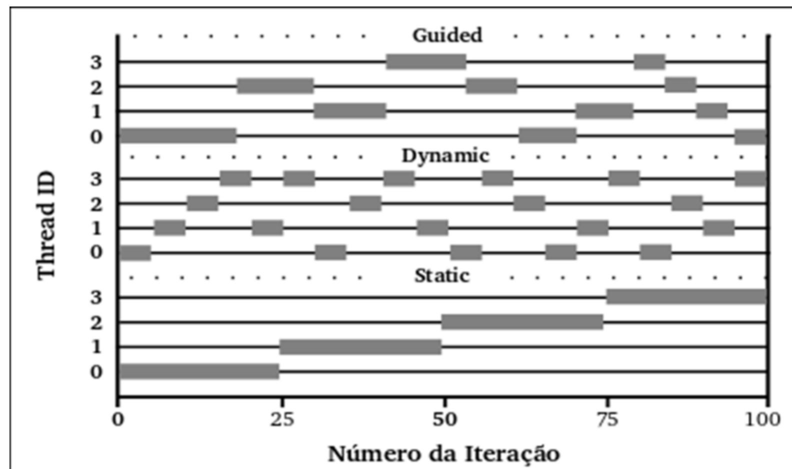
Os **construtores de compartilhamento de trabalho** são os responsáveis por informar ao compilador como será distribuída a carga de trabalho entre as *threads*. Necessariamente, eles devem ser utilizados internamente, em uma região paralela, e são divididos em: laços paralelos, seções paralelas e, construtor único de trabalho.

- *Laços paralelos*: Neste construtor, as iterações da estrutura de repetição (*for*) são distribuídas entre as *threads*. A granularidade da distribuição da carga de trabalho é determinada pelo *chunk*: um subconjunto contíguo de iterações (e.g.: uma iteração de um laço *for* ou múltiplas iterações). Os *chunks* podem ser escalonados entre as *threads* através de diferentes escalonadores: ***static***, onde os *chunks* são atribuídos para as threads estaticamente através da política *round-robin*, ordenados pelo número da *thread*; ***dynamic***, onde as iterações são atribuídas para as *threads* conforme solicitação; ***guided***, que é similar ao escalonamento *dynamic*, no entanto, o tamanho do *chunk* decrementa a cada iteração; e por fim, ***runtime***, em que 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.

A Figura 2.8 ilustra o comportamento destes escalonadores. Nela, 100 iterações de um laço (eixo X) são distribuídas entre quatro *threads* (eixo Y). No escalonador *static* podemos notar que cada *thread* irá computar o *chunk* definido de 25 iterações, distribuídas uniformemente e em ordem pelo identificador da *thread*. Para o escalonador *dynamic,* onde o *chunk* é de 5 iterações, cada *thread* receberá 5 iterações para computar a cada solicitação. Por fim, o escalonador *guided* também possui *chunk* igual a 5, no entanto, durante as primeiras solicitações, cada thread receberá porções maiores de iterações.

- *Seções paralelas*: Este construtor permite a atribuição de partes diferentes de trabalho entre as *threads*. Ele permite especificar diferentes regiões de código para serem executadas em paralelo por diferentes *threads*, assim, cada *thread* é responsável pela execução de cada bloco.

Figura 2.8 - Exemplo de Atribuição de Iterações ás *Threads*



- *Construtor único:* Neste construtor, um bloco de execução de código é atribuído para ser executado por uma única *thread*. Enquanto a *thread* escolhida está executando este bloco de trabalho, as demais aguardam em uma barreira implícita, no final desta região.

Por fim, as **diretivas de sincronização** são utilizadas principalmente para evitar condições de corrida, que acontecem quando duas ou mais *threads* tentam atualizar ao mesmo tempo a mesma variável. As três principais diretivas explícitas são ***critical***, que restringe a execução de uma determinada tarefa a apenas uma *thread* por vez; ***atomic***, onde a atualização de uma região de memória é realizada atomicamente por uma única *thread*; e ***barrier***, que é utilizada para sincronizar todas as *threads* em um dado momento da execução.

A sincronização entre as *threads* também pode ser implícita, como ocorre no início e fim de uma região paralela. Por exemplo, sempre que ocorre a criação de novas *threads* (*fork*), elas são sincronizadas entre si antes de iniciar a computação. O mesmo ocorre na finalização destas *threads* (*join*), que são todas sincronizadas antes de ser finalizadas. Sempre que houver ponto de sincronização (explícito e implícito) entre as *threads*, no OpenMP elas entram em estado de *busy-waiting*, ou seja, elas executam instruções de acessos a memória para checar a variável de controle repetidamente até o final da sincronização.

2.3.2 POSIX Threads

POSIX Threads ou simplesmente Pthreads, é uma API para programação em memória compartilhada para linguagens C/C++. Diferentemente do OpenMP, onde o paralelismo é expresso em alto nível de abstração com a inserção de diretivas no código sequencial, o paralelismo em Pthreads 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). As sub-rotinas que compreendem Pthreads 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*. Similar ao OpenMP, existe um fluxo de execução inicial que executa a parte sequencial do código e quando é necessário, novas *threads* são criadas através da função `pthread_create()`. A finalização de uma *thread* pode ocorrer explicitamente pela chamada à função `pthread_exit()`. A sincronização entre as *threads* é realizada através da função `pthread_join()`, que faz com que as *threads* fiquem bloqueadas até a finalização das demais.

Operações de ***mutex*** são utilizadas para implementar seções criticas e operações atômicas (GRAMA et al. 2003). Elas são definidas através de funções do tipo `pthread_mutex()`. A função `pthread_mutex_lock()` é utilizada para definir o início de uma seção crítica. Quando uma *thread* acessa a região crítica esta função bloqueia o acesso das demais. O acesso só será liberado para as demais *threads* quando a *thread* que está acessando a região crítica executar a função `pthread_mutex_unlock()` que define o fim da seção crítica.

As funções do tipo *mutex* são tipicamente utilizadas para habilitar acesso exclusivo de uma *thread* a uma região compartilhada da memória. Entretanto, algumas situações requerem dinamismo para gerenciar acesso exclusivo a estes recursos. Assim, **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 de funções do tipo `pthread_cond()`.

No entanto, assim que uma *thread* termina a execução de uma região crítica, ela segue seu fluxo de execução. Dependendo da aplicação, a etapa $x+1$ só pode começar após a finalização da etapa x. Portanto, fazem-se necessárias barreiras entre estas etapas, onde as *threads* só continuarão a execução quando todas as demais estiverem no mesmo ponto de execução. Este comportamento é fornecido através de funções de **sincronização** do tipo `pthread_barrier()`.

Diferente do OpenMP, em *Pthreads* a sincronização ocorre pelo bloqueio das *threads* através de *mutex* (TANENBAUN, 2009). Um *mutex* é uma variável que pode ter dois estados: livre ou ocupado. Quando uma *thread* precisa acessar uma região crítica, ela chama um

mutex_lock. Se o *mutex* está livre (significa que a região crítica está disponível), a chamada é bem sucedida e a *thread* que fez a chamada pode entrar na região crítica. Por outro lado, se o *mutex* está ocupado, a *thread* que fez a chamada é bloqueada até que a *thread* que se encontra na região crítica tenha terminado e chame *mutex_unlock*. Se vários processos estiverem bloqueados no *mutex*, um deles será escolhido aleatoriamente e poderá entrar na região crítica. Neste tipo de sincronização, as *threads* que estão aguardando "perdem" o processador e ficam em estado de espera e ganham novamente ele quando forem re-escalonadas. Embora esta técnica não impacte no número de acessos a memória, a troca de contexto (perder e ganhar o processador) pode levar a perda de desempenho se a sincronização for constantemente exigida.

2.3.3 *Message-Passing Interface*

O MPI é uma biblioteca padrão para comunicação em memória distribuída que foi 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. Um programa MPI é definido como uma coleção de processos e a comunicação entre estes processos se dá através de operações de comunicações. 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, 1998). No escopo deste trabalho, serão abordadas as normas MPI-1 e MPI-2.

Nos sistemas *multicore*, 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 FIFO. Para entender o comportamento, vejamos, por 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. Já 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 (PELETTI, 2007). Da mesma forma que a comunicação através de variáveis compartilhadas, estas operações utilizando filas de mensagens são realizadas em regiões compartilhadas da memória.

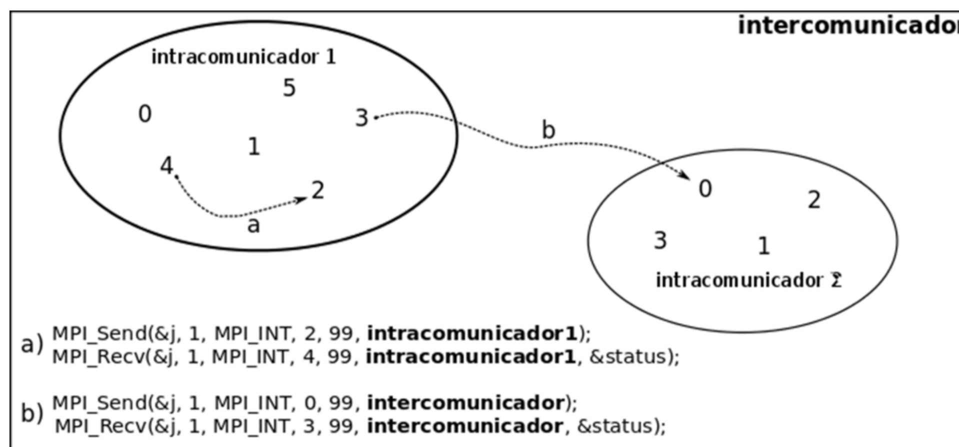
2.3.3.1 MPI-1

A primeira norma proposta, denominada MPI-1, especifica operações de comunicações ponto a ponto e coletivas, dentre outras características. Um programa desenvolvido utilizando MPI-1 cria estaticamente todos os processos no início da execução, portanto a quantidade de processos não sofre alterações durante a execução do programa. Ao iniciar o programa, cada processo executa a função de inicialização do ambiente de execução MPI, o `MPI_Init()`. Já a finalização de um processo MPI ocorre através da chamada à função `MPI_Finalize()`.

Após inicializar o ambiente de execução, cada processo MPI recebe o seu identificador (*rank*) 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. Por padrão, existe um comunicador pré-definido, chamado `MPI_COMM_WORLD`. O número total de processos de um determinado comunicador é obtido através da função `MPI_Comm_size()`.

Os comunicadores podem ser divididos em intracomunicadores, quando a comunicação é interna a um grupo de processos de um mesmo comunicador; e intercomunicadores, quando a comunicação é realizada entre processos envolvendo intracomunicadores diferentes. A Figura 2.9 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 2.9, 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.

Figura 2.9 - 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, 2010)

O cenário (**a**) corresponde a 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 cenário (**b**), corresponde à comunicação entre dois intracomunicadores diferentes (`intracomunicador1` e `intracomunicador2`) através do intercomunicador. 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()`.

Estas funções (utilizadas na Figura 2.9) realizam operações síncronas e, portanto, são utilizadas como pontos de sincronismo entre os processos envolvidos, o que significa que a função irá ficar bloqueada até que todos os dados tenham sido enviados ou copiados para o *buffer* de memória. Após a operação estar completa, cada processo segue o seu fluxo de execução. Entretanto, operações bloqueantes podem se tornar o gargalo da aplicação, já que mantêm os processos bloqueados até a comunicação estar concluída. Esta limitação pode ser contornada através do uso de operações assíncronas.

A operação de envio assíncrona (`MPI_Isend()`) retorna após a mensagem ser copiada para o *buffer* de envio. Desta forma, enquanto a mensagem está sendo transmitida, o processo que a enviou pode seguir com a sua execução. Similarmente, a operação de recebimento não bloqueante (`MPI_Irecv()`) inicia a operação, mas não a completa. Neste caso, a função completará quando a mensagem estiver armazenada no *buffer* de recebimento. Enquanto isto, o processo pode computar sobre outros dados. Entretanto, em algum momento da execução, deve ser realizada uma verificação, a fim de saber se a operação está completa

ou não. Assim, as funções `MPI_Wait()` e `MPI_Waitany()` podem ser utilizadas neste contexto, indicando que o processo só seguirá seu fluxo de execução quando concluir as comunicações iniciadas. As operações bloqueantes e não-bloqueantes podem ser utilizadas em conjunto (por exemplo: `MPI_Send()` e `MPI_Irecv()`), e a eficiência da aplicação em termos de comunicação é decorrente do uso adequado destas funções.

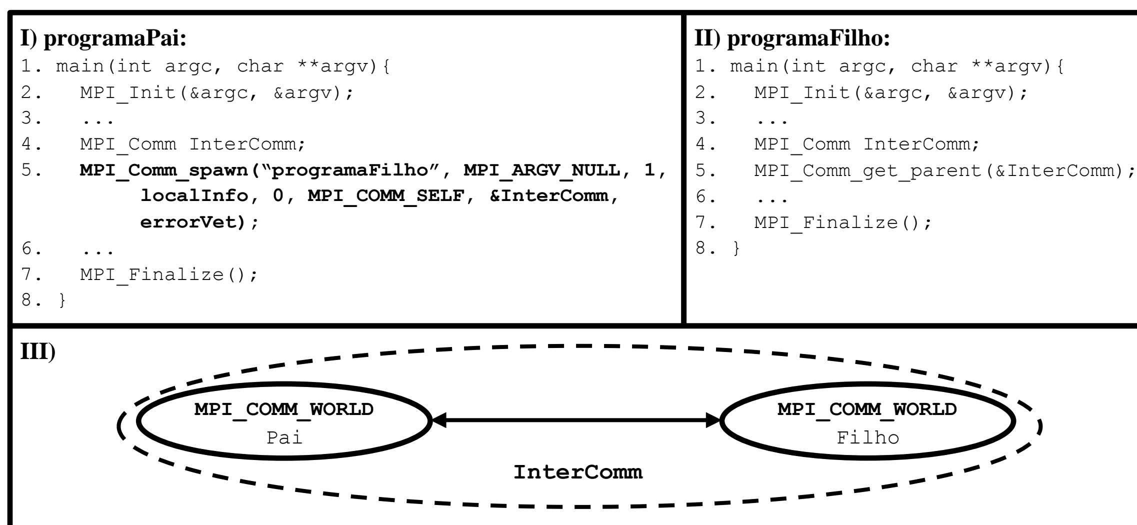
As operações coletivas em MPI são divididas em operações de sincronização, movimento de dados e computação coletiva. As operações de sincronização são aquelas onde os processos aguardam até que todos os demais processos do grupo atinjam o ponto de sincronização, para então continuar o fluxo de execução. Por outro lado, as operações de movimento de dados são aquelas que permitem 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, são aquelas que realizam operações sobre os dados provenientes de diversos processos. As operações mais comuns são: soma, multiplicação, máximo e mínimo valor, entre outras.

2.3.3.2 MPI-2

A norma MPI-2 define um conjunto de novos tópicos ao padrão MPI. Segundo Gropp (2003), MPI-2 se diferencia do MPI-1 nas seguintes características: criação dinâmica de processos, operações de entrada e saída paralela, comunicações assimétricas e coletivas estendidas.

Uma aplicação implementada com MPI-2 geralmente inicia a execução com um único processo. A criação dinâmica de processos é implementada através da primitiva `MPI_Comm_spawn()`. Ela deve ser invocada por um dos processos da aplicação MPI, o qual será chamado de pai. A invocação da primitiva leva a criação de um novo processo, chamado filho, o qual não precisa ser idêntico ao pai. Quando um processo filho é criado, ele irá pertencer a um intracomunicador diferente do pai e a comunicação entre eles ocorrerá através de um intercomunicador. No processo filho, o intercomunicador que o liga com o pai é retornado após a execução da função `MPI_Comm_get_parent()`. Já no processo pai, o intercomunicador que o liga ao filho é retornado na execução da função `MPI_Comm_spawn()`.

Figura 2.10 - 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, 2012)

A Figura 2.10 exemplifica a criação de um processo. No retângulo I, é descrito o pseudocódigo denominado `programaPai` que será executado pelo processo pai. Nele, na linha 2 e 7 ocorre respectivamente a inicialização e finalização do ambiente MPI. Já na linha 5, ocorre a criação do processo filho, que irá executar o programa chamado de `programaFilho`, que é definido no primeiro parâmetro (neste caso, 1) e o parâmetro 7 indica o intercomunicador que será utilizado para comunicações entre o pai e o(s) filho(s) (neste caso, `InterComm`). Note que a inicialização das variáveis, bem como outras definições foram omitidas.

Ainda na Figura 2.10, porém no retângulo II, é apresentado o pseudocódigo do processo filho. Após a inicialização do ambiente MPI, ele executará a função `MPI_Comm_get_parent()` localizada na linha 5. Esta função retornará o intercomunicador `InterComm` que ligará o seu intracomunicador com o do pai. Por fim, a ilustração do relacionamento hierárquico resultante após a criação do processo filho é encontrada no retângulo III da mesma figura. Nela, cada processo está interno ao seu intracomunicador (elipse) e interligado através do intercomunicador, representado pela elipse vazada.

Além das possíveis comunicações ponto a ponto (utilizando inter e intracomunicadores) e coletivas (utilizando intracomunicadores) descritas na Seção MPI-1, o MPI-2 permite ainda comunicações coletivas através de intercomunicadores. 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.

2.4 Conclusão do Capítulo

Este capítulo contextualizou a programação paralela através de três diferentes Interfaces de Programação Paralela com comunicação por memória compartilhada (OpenMP e Pthreads) e troca de mensagens (MPI-1 e MPI-2) em processadores *multicore*.

As arquiteturas *multicore* permitem a exploração do paralelismo no nível de *threads* através de múltiplas unidades de execução. Entretanto um dos principais gargalos da exploração do paralelismo está relacionado com a comunicação entre os processadores. Isto porque a troca de informações ocorre em níveis da memória que estão situadas hierarquicamente mais distantes do processador e, portanto possuem maior tempo de acesso e consumo de energia por acesso. Ademais, mostrou-se que o impacto da comunicação pode ser diferente em processadores de propósito geral e de baixo consumo de energia. Visto que o sistema de memória destes processadores é diferente (hierarquia, tamanho, custo por acesso em tempo e energia, etc.).

Também foram apresentados os modelos de comunicação entre as *threads*/processos para arquiteturas *multicore*: memória compartilhada, onde as *threads* compartilham regiões de memória e a comunicação é realizada através de variáveis compartilhadas; e troca de mensagens, onde os processos não compartilham regiões de memória e a comunicação ocorre com o uso de primitivas de comunicação do tipo *send/receive*.

As Interfaces de Programação Paralelas que fazem parte do escopo deste trabalho foram apresentadas: OpenMP, Pthreads e MPI. OpenMP cria *threads* seguindo o modelo *fork/Join,* o paralelismo é extraído através de diretivas de compilação e a sincronização ocorre através de *busy-waiting*. Pthreads apresenta exploração do paralelismo de forma explícita, e a sincronização através de *mutex*. MPI também explora o paralelismo de forma explícita, porém 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.

3 TRABALHOS CORRELATOS

Com relação aos trabalhos relacionados a esta dissertação, a Seção 3.1 discute os trabalhos desenvolvidos sobre as Interfaces de Programação Paralela. Nela são apresentados os principais eixos de comparação entre ambas: Facilidade de Programação, Controle e Flexibilidade, Sobrecarga imposto para o gerenciamento (criação/finalização, sincronização) das *threads*/processos, Desempenho e Consumo de Energia. Após, a Seção 3.2 apresenta as principais contribuições deste trabalho e a sua importância perante os trabalhos correlatos discutidos previamente.

3.1 Comparação entre Interfaces de Programação Paralela

3.1.1 Facilidade de Programação

Facilidade de programação está relacionada ao esforço de programação necessário para gerar códigos paralelos.

Kuhn (2000) compara OpenMP e Pthreads através da paralelização do programa *Genehunter*. Tal programa consiste em analisar testes de DNA nos membros de uma árvore genealógica, para verificar em quais membros da família uma doença está presente. O trabalho apresenta conceitos relacionados às seguintes diferenças entre OpenMP e Pthreads: estilo de codificação; especificações de escopo dos dados (privado ou compartilhado) para permitir a comunicação entre as *threads*; ferramentas para auxiliar no desenvolvimento de código OpenMP; além de resultados de desempenho relacionados a alocação de memória, pontos de sincronizações e escalonamento das tarefas. Dentre seus resultados, o principal aponta para a facilidade da exploração do paralelismo com OpenMP através da inserção de poucas diretivas no código sequencial.

Krawezik et al (2003) compara o desempenho de três estilos de programação paralela do OpenMP com o MPI em diferentes multiprocessadores de memória compartilhada. As aplicações alvo contemplam um subconjunto do NAS *Parallel benchmark* (CG, MG, FT e LU) e com dois tamanhos de entrada de dados (A e B). Os resultados mostram que OpenMP provê desempenho competitivo comparado ao MPI. Entretanto, o esforço computacional

necessário para escrever códigos paralelos com MPI é extremamente alto quando comparado com OpenMP.

Um estudo analítico, seguido de análise qualitativa entre OpenMP e UPC (*Unified Parallel C*) (PADUA, 2011), foi realizado em Marowka (2004). O estudo de caso foi a paralelização da aplicação Cálculo do Pi utilizando integração numérica. Conforme os autores, em termos de estilos de programação, ambas são relativamente simples (OpenMP através de diretivas de compilação e UPC através de construtores paralelos), com alta abstração, ou seja, o programador não precisa saber questões de baixo nível da arquitetura. Por outro lado, o programador tem responsabilidade em utilizar corretamente as diretivas e construtores paralelos, checar dependência de dados, conflitos, *deadlocks*, condições de corridas, entre outras questões que podem resultar na execução incorreta do programa.

Hochstein (2005) avalia o esforço de codificação na exploração do paralelismo com programadores iniciantes. Para tal, os programadores iniciantes codificaram diferentes tipos de aplicações com OpenMP e MPI. Ademais, foram utilizadas diferentes abordagens e práticas que podem auxiliar e reduzir o esforço de codificação. Conforme esperado pela sua facilidade, OpenMP possibilitou melhor exploração do paralelismo, com ganho de desempenho próximo do ideal. Por outro lado, MPI exigiu maior esforço de programação (maior número de linhas inseridas na versão sequencial) e a maioria dos códigos gerados foram ineficientes, ou seja, paralelizados de forma incorreta.

Patel (2008) compara a produtividade na programação com MPI e UPC, e o impacto de diferentes formas de codificação no desempenho das aplicações. Os resultados mostram que MPI e UPC obtiveram desempenhos similares na média, porém, UPC obteve alta variação de desempenho entre suas versões. A análise dos autores revela que diferentes escolhas no desenvolvimento da paralelização com UPC implicou em diferentes resultados de desempenho. Ademais, os autores afirmam que técnicas de inspeção de código podem automatizar e tornar menos propenso a erros o processo de codificação de aplicações paralelas.

Kasim (2008) acredita que a escolha do modelo de programação paralela a ser empregado na paralelização não deve levar em conta somente desempenho, mas também aspectos qualitativos de quão abstrata a programação é para os desenvolvedores. Assim, os autores estudam e avaliam qualitativamente seis IPPs para sistemas de memória compartilhada e distribuída: Pthreads, OpenMP, CUDA, MPI, UPC e Fortress. O trabalho consiste de um estudo em fase inicial sem a apresentação de resultados, que objetiva auxiliar

novos desenvolvedores a fazer a melhor escolha do modelo de programação considerando produtividade do programador paralelo.

Tristam (2010) investiga o esforço na codificação de três modelos de programação paralela para a linguagem C++ em sistemas *multicore*. A análise é realizada entre OpenMP, Intel TBB e Pthreads. Para tal, os autores programaram múltiplas versões paralelas do Conjunto de *Mandelbrot*. Os resultados mostram que OpenMP exige menor esforço de programação, e com apenas a inserção de uma linha no código sequencial (diretivas de compilação), obteve desempenho próximo a Intel TBB e Pthreads. Embora Intel TBB e Pthreads tenham obtido melhor desempenho, mostrou-se que, para gerar um código paralelo eficiente, foi preciso inserir, no código sequencial, 34 e 57 linhas em suas versões paralelas, respectivamente.

Shekhar (2011) avalia o uso do OpenMP, Pthreads e GCD (*Grand Central Dispatch*)[1] na paralelização de uma aplicação regular, ou seja, possui carga de trabalho fixa durante toda a execução da aplicação (detecção de faces) e irregular, onde a carga de trabalho varia durante a execução da aplicação (reconhecimento automático da fala). Os resultados mostram que os ganhos em escalabilidade com ambas IPPs são similares. Entretanto, o esforço de programação envolvido na paralelização é diferente. Enquanto na paralelização com Pthreads foi necessário reestruturar boa parte do código sequencial, em OpenMP e GCD apenas foi necessário identificar as regiões paralelas e expressar o paralelismo através de diretivas de compilação. Todavia, o controle do paralelismo é muito maior em Pthreads, onde o programador pode controlar fatores de baixo nível de programação (afinidade de *thread*, ajuste fino da carga de trabalho, entre outros), os quais não estão presentes em OpenMP e GCD. Assim, em Pthreads o programador ganha maior possibilidade de ajustar o paralelismo para obter melhor utilização da memória *cache*, resultando em menor redução no consumo de energia e melhor desempenho.

Ajkunic (2012) apresenta uma comparação do esforço de programação de cinco IPPs na paralelização da multiplicação de matriz: Intel Cilk Plus, OpenMP, Intel TBB, Pthreads e MPI. Os resultados mostram que tanto OpenMP quanto Intel Cilk Plus possuíram menor esforço de programação, com a adição de apenas uma única linha ao código sequencial. Já para Intel TBB, foi necessário adicionar 12 linhas ao código sequencial. O gerenciamento (criação/finalização) e distribuição da carga de trabalho entre as *threads* fizeram com que

---

[1] https://libdispatch.macosforge.org/

fosse necessário alterar 24 linhas do código sequencial utilizando Pthreads. No entanto, a IPP que exigiu maior esforço de programação foi MPI, com a alteração de 36 linhas do código sequencial. A diferença no esforço entre MPI e Pthreads está relacionada à necessidade do programador prover comunicação entre os processos MPI, já que ela ocorre através de troca de mensagens.

3.1.2 Controle e Flexibilidade

No contexto de programação paralela, as principais características incluem o gerenciamento (criação/finalização) das *threads/*processos, escalonamento das tarefas, distribuição da carga de trabalho, sincronização entre as *threads* e a estruturação de uma região paralela.

A criação dinâmica de processos em MPI-2 fornece maior dinamicidade e flexibilidade para as aplicações paralelas, possibilitando melhor aproveitamento dos recursos computacionais. Neste contexto, estudos têm sido realizados com o propósito de fornecer melhor aproveitamento dos recursos computacionais. Leopoldo e Sub (2006) utilizaram MPI-2 em conjunto com o OpenMP com o propósito de fornecer dinamicidade e adaptabilidade à aplicação *WaterGAP* modelada sobre o padrão Mestre/Trabalhador. Ela computa a disponibilidade atual e futura de água no mundo. Para isto, divide o continente em pequenas células (grades) de tamanho igual e utiliza dados de entrada (clima, vegetação) para simular o controle de água (precipitação) e a velocidade do fluxo da água em rios. Os cálculos realizados sobre cada célula são divididos em três níveis: grande, médio e fraco. Portanto, células podem ter tempos de computação diferentes. Com o gerenciamento de processos em tempo de execução, o processo Mestre, através de cálculos realizados por um gerenciador de carga de trabalho, consegue adaptar a aplicação à arquitetura disponível conforme há aumento da carga de trabalho. Os resultados mostram que o uso de MPI-2 melhorou o desempenho da aplicação, principalmente por permitir melhor gerenciamento dos recursos disponíveis.

Estudos têm sido realizados com o propósito de fornecer melhor aproveitamento dos recursos computacionais, porém através da melhoria de políticas de distribuição de novos processos nos recursos disponíveis da arquitetura alvo. Cera et al. (2006) propõe a implementação de um módulo extra para o escalonador de processos. Este módulo tem a finalidade de definir em tempo de execução em qual recurso computacional o novo processo será alocado. Ele interage com o processo que realizará a chamada para a primitiva

`MPI_Comm_spawn` e, através de um grafo de tarefas presente no gerenciador de recursos do escalonador, indica qual recurso é mais adequado para alocar fisicamente um novo processo. De posse do local onde será criado o novo processo, ocorre então a criação do mesmo. Quando o processo criado for finalizado, ele informa ao gerenciador de recursos, o qual irá atualizar o grafo de tarefas. Este módulo foi implementado em dois modelos de escalonador: *Round-Robin* e *List Scheduling*. Os resultados mostram que, para ambos os escalonadores, conseguiu-se obter equilíbrio na distribuição dos processos, melhorando assim, a distribuição dos processos criados em tempo de execução entre os recursos computacionais disponíveis.

Na busca por melhorar a utilização dos recursos computacionais através da criação dinâmica de processos, Cera, et al. (2010) discute o uso de aplicações dinâmicas MPI para aumentar a taxa de utilização dos recursos de *clusters* de computadores. Neste trabalho, faz-se a utilização de trabalhos (*jobs*) maleáveis (ou seja, aqueles que podem se adaptar aos recursos com disponibilidade dinâmica) através da dinamicidade oferecida pelo MPI-2 para avaliar o aproveitamento dos recursos. Os resultados mostram que a utilização do *cluster* pode ser aumentada em cerca de 25% com o uso de trabalhos maleáveis, quando comparado a uma abordagem não maleável.

Outra possibilidade do MPI-2 é oferecer meios de implementar tolerância a falhas a uma aplicação através da gerência de processos em tempo de execução. Por exemplo, o *midleware EasyGrid* MPI apresentado em Silva e Rebello (2011), implementa funções de criação dinâmica de processos para que aplicações MPI possam executar de maneira eficiente em ambientes onde as falhas de sistema, *hardware* e aplicação são recorrentes. Portanto, ao ocorrer uma falha na aplicação, o *midleware* desenvolvido contorna esta falha com o gerenciamento de processos em tempo de execução, sem causar maiores danos à aplicação (i.e., finalização total da aplicação, causando o desperdício do trabalho computado até o momento em que a falha ocorreu).

Stamatakis et al. (2008) compara o desempenho, flexibilidade e compatibilidade do OpenMP, Pthreads e MPI através da paralelização da função *Phylogenetic Likelihood*, amplamente utilizada na bioinformática. No trabalho são abordadas diferentes estratégias de paralelização executando em diferentes arquiteturas. Conforme os autores, Pthreads possibilitou melhor flexibilidade e controle da aplicação paralela. Isto, pois, permite explicitar o local de alocação da memória, ou seja, distribuir as estruturas de dados e assim facilitar o processo de manutenção e desenvolvimento, comparado as versões MPI e OpenMP.

3.1.3 Custo Extra Imposto pela Paralelização

Enquanto a paralelização leva ao ganho de desempenho, o custo adicional para a criação, finalização e gerenciamento das *threads/*processos influencia na eficiência da aplicação.

A programação paralela com OpenMP é simplificada pelo uso de diretivas de compilação. No entanto, o uso ineficiente destas diretivas pode trazer impacto negativo no desempenho da aplicação. Assim, Fredrickson (2003) avalia o impacto das diretivas de criação/finalização de *threads*, distribuição da carga de trabalho (*schedules*) e de sincronização entre as *threads* no desempenho das aplicações. Observou-se que o maior sobrecarga ocorre na criação/finalização das *threads* através da diretiva *#pragma omp parallel*, e que este *overhead* aumenta com o aumento do número de *threads*. Considerando as diretivas de sincronização explícita (*critical*, *atomic*, *lock/unlock*, *barrier*, *ordered* e *single*), a que obteve pior desempenho foi a *single* (onde somente uma *thread* executa um trecho de código e as demais aguardam em uma barreira implícita), adicionando um *overhead* de 60 *microssegundos* em uma única chamada na execução com 70 *threads*, enquanto que as demais diretivas não passaram de 20 microssegundos de *overhead* para o mesmo número de *threads*. Por fim, o *schedule dynamic* possui maior *overhead* na distribuição da carga de trabalho para *chunk* igual a 1 (granularidade fina). Tal *overhead* chega a ser 22 vezes maior que o mecanismo de agendamento *static* na execução com 6 *threads* e de até 10 mil vezes para 64 *threads*. Embora este *overhead* diminua conforme aumenta o valor do *chunk*, ele ainda continua maior que o *static*. Cabe ressaltar, que o maior *overhead* por parte do *schedule dynamic* é por causa da alocação das iterações em tempo de execução, enquanto que no static esta distribuição é feita em tempo de compilação.

Nanjegowda (2009) apresenta o impacto de diferentes implementações de barreiras de sincronização entre *threads* no OpenMP, incluindo o algoritmo de bloqueio centralizado e diferentes variações do algoritmo *busy waiting*. Tais algoritmos foram executados em operações de barreira (*#pragma omp barrier*), redução (*#pragma omp reduction*) e sincronização implícita na criação/finalização de *threads* (*#pragma omp parallel*). Os resultados mostram que embora o algoritmo *busy-waiting* tenha impacto direto na quantidade de acessos a memória, ele é o que obteve melhor desempenho na sincronização de até 256 *threads*. Ademais, os autores reforçam que o desempenho das barreiras de sincronização é dependente de vários fatores, como: número de *threads*, arquitetura (organização da memória), aplicação e a carga de trabalho do sistema.

Conforme apresentado na Seção 2.3.3, o MPI-2 possibilita a criação dinâmica de processos em tempo de execução. Embora esta característica possibilite flexibilidade as aplicações, a tarefa de criar processos em tempo de execução possui *overhead* no desempenho da aplicação. Assim, Lorenzon et al. (2012) analisa o impacto da criação dinâmica de processos e do consequente relacionamento hierárquico em MPI-2. Para isso, partiu-se de uma versão Mestre/Trabalhadores em MPI-1 do Jogo da Vida e procurou-se adicionar a criação dinâmica de processos mantendo as mesmas características da aplicação. Na primeira versão, todos os processos são criados em uma única chamada ao `MPI_Comm_spawn`. Isto faz com que todos os processos filhos pertençam ao mesmo comunicador e permite que troquem informações entre si. Já na segunda versão, um único processo é criado a cada `MPI_Comm_spawn`, o que faz com que cada processo pertença a um comunicador diferente e a única forma de comunicação entre os processos filhos é através do processo pai. Assim, enquanto que na primeira versão os processos comunicam entre si, na segunda versão, todas as comunicações são mediadas pelo processo pai.

Nas comparações envolvendo a primeira versão, os resultados mostram um acréscimo constante em torno de 2 segundos para a versão MPI-2 quando comparada com a versão MPI-1. Já os resultados da segunda versão mostram que é possível atenuar o custo adicional proveniente da criação dinâmica de processos através da implementação de múltiplos `MPI_Comm_spawn`. Assim, tão logo um Trabalhador seja criado, ele já recebe sua parcela de dados a computar. Desta maneira, alguns Trabalhadores iniciam sua computação enquanto os demais ainda estão sendo criados. Na comparação entre as duas versões MPI-2, criar um processo a cada `MPI_Comm_spawn` mostrou-se mais eficiente, mesmo que este modelo de criação gere a necessidade de envolver o Mestre em todas as comunicações.

Lorenzon et al. (2013) realiza um estudo sobre o impacto da criação dinâmica de processos em MPI e como ele pode ser contornado através de técnicas de programação eficientes e características avançadas fornecidas pelo próprio MPI. Para isto, foram implementadas aplicações em MPI-1 e MPI-2, sob dois problemas com características diferentes: Jogo da Vida, representando os problemas com carga regular de trabalho e cujo tempo de computação é similar entre os processos; e o *Skyline Matrix Solver*, com carga irregular de trabalho e tempo de computação diferente entre os processos. Os resultados mostram que para aplicações com carga de trabalho regular, no melhor dos casos, a criação de processos em tempo de execução impactou em apenas 0,2% no tempo total da aplicação. Já para aplicações com carga de trabalho irregular, mostrou-se que é possível obter ganho com a

criação dinâmica de processos, em que no melhor caso, a implementação com MPI-2 foi 6% mais eficiente que a implementação MPI-1.

3.1.4 Desempenho

Mállon et al (2009) avalia o desempenho do MPI, UPC e OpenMP em arquiteturas multicore através de um subconjunto de aplicações do NAS *Parallel Benchmark.* As execuções foram realizadas em ambiente com memória híbrida (memória distribuída e compartilhada) e somente memória compartilhada. Os resultados mostram que no ambiente de memória híbrida, MPI obteve melhor desempenho geral comparado ao UPC, devido a melhor utilização da localidade dos dados na memória *cache*. Entretanto, em algumas aplicações, UPC foi melhor que MPI. A partir da execução com 128 processos, MPI e UPC não obtiveram o desempenho esperado, mostrando que a comunicação através da rede pode ser um fator de contenção para a escalabilidade de muitas aplicações. No ambiente de memória compartilhada, embora a comunicação entre as *threads* OpenMP ocorra através de variáveis compartilhadas, em alguns casos MPI e UPC obtiveram melhor desempenho, novamente, devido a melhor exploração da localidade de dados na *cache*.

Tristam et al. (2010) investiga o desempenho de OpenMP, Intel TBB e Pthreads em sistemas *multicore*. Para tal, os autores utilizaram múltiplas implementações paralelas do algoritmo Mandelbrot Set, usando cada uma das IPPs acima. Os resultados mostram que, quanto maior for o controle da paralelização pelo programador, maior é o desempenho obtido. Isto porque o programador pode realizar ajustes finos na paralelização, como ajustar a carga de trabalho da forma de se obter melhor balanceamento. Assim, as implementações utilizando Pthreads e Intel TBB obtiveram os melhores resultados de desempenho.

Wahlén (2010) mostra que o desempenho de uma aplicação paralela é altamente influenciado pelas características da aplicação e da IPP utilizada. Assim, o trabalho discute o comportamento de Pthreads, OpenMP e Cilk++ no desempenho de quatro aplicações com paradigmas recursivo e iterativo executando em um processador *dual Quad-Core AMD Opteron*. As aplicações escolhidas para a classe de paradigma iterativo correspondem ao cálculo de números primos, simulação de correios e Jogo da Vida. Para o cálculo de números primos, que é trivialmente paralelizado sem a existência de dependência de dados, ambas IPPs obtiveram bom desempenho. Pthreads obteve melhor *speedup*, de 6.05 com 8 *threads*,

enquanto que OpenMP e Cilk++ tiveram desempenhos similares (5.60 e 5.79 respectivamente).

Para a aplicação simulação de correios, que é similar a aplicações do tipo produtor/consumidor, Pthreads novamente obteve melhor *speedup* de 1.73 para 8 *threads*. Já Cilk++ e OpenMP atingiram *speedup* de 1.55 e 1.49, respectivamente, para a mesma quantidade de *threads*. O desempenho das versões paralelas foi baixo, pois a aplicação possui grande quantidade de comunicação entre as *threads*. Assim, há a necessidade de prover mecanismos de exclusão mútua para garantir esta comunicação e isto contribui para o baixo desempenho.

Para o Jogo da Vida, Cilk++ obteve o melhor desempenho com *speedup* de 4.42 enquanto que Pthreads e OpenMP atingiram 3.97 e 3.90 respectivamente. Conforme o autor, Cilk++ obteve o melhor desempenho devido a utilização de um modelo de barreira de sincronização melhor que as demais IPPs. Já para a aplicação recursiva, chamada quadratura adaptativa, Cilk++ obteve melhor desempenho, com *speedup* de 7.71, comparado com 2.09 e 3.31 para Pthreads e OpenMP, respectivamente. A grande diferença de desempenho ocorre porque em aplicações recursivas paralelas o número de *threads* altera dinamicamente.

Lorenzon et al. (2011) apresenta um estudo comparativo entre três IPPs (OpenMP, Pthreads e Intel Cilk++) através da paralelização de um método de determinação de similaridades entre histogramas, com foco no rastreamento de veículos. Os resultados mostram que através do ajuste fino na distribuição da carga de trabalho entre as *threads,* Pthreads obteve os melhores resultados de desempenho, com *speedup* de 3.94 para execução com 4 *threads*. Já Cilk++ e OpenMP, através da inserção de uma única linha de código, conseguiram obter ganhos expressivos, chegando ao *speedup* de 3.41 e 3.68 respectivamente.

Recentemente, com a popularização dos sistemas embarcados *multicores*, alguns trabalhos têm avaliado o uso de programação paralela para estes sistemas. Lee et al. (2011) apresenta um estudo do desempenho obtido com OpenMP em sistemas embarcados *multicore*. Para tal, os autores paralelizam duas aplicações: *Dijkstra* e *Stringsearch*. Os resultados mostram que é possível obter *speedup* de até 111% com relação à execução sequencial. Isto mostra o potencial do uso de programação paralela neste tipo de sistema.

Com o advento dos sistemas *Many-core*, Cao (2013) analisa o comportamento do OpenMP, Intel TBB e Cilk++ quanto a escalabilidade destas interfaces neste tipo de sistema. Dentre 5 aplicações (*Alignment, SparseLU, Strassen, Sort* e *Health*) executadas com OpenMP, somente *Alignment* obteve speedup próximo do linear quando executado com 32 threads. Já *SparseLU* e *Sort* atingiram *speedup* de 17 e 12 respectivamente. Por fim, *Strassen*

e *Health* tiveram os piores resultados, com apenas 3 e 2 de *speedup* com 32 *threads*. Dentre as razões para o baixo desempenho, os autores citam que a estratégia de escalonamento provida pelo sistema de execução pode interferir negativamente no desempenho, fazendo com que sejam desperdiçados recursos computacionais através da má utilização e alocação de recursos.

Hua et al. (2013) apresenta comparação e análise de desempenho entre OpenMP e MPI através do Algoritmo de Iteração de Newton, um algoritmo não-linear e de simples paralelização. Os autores utilizaram as ferramentas *Intel VTune Performance Analyzer*[1], *Intel Thread Checker*[1] e *Intel Thread Profiler*[2] para analisar o comportamento de tais IPPs. Os resultados mostram que a utilização de OpenMP em sistemas de memória compartilhada podem levar a bons resultados de desempenho, enquanto que o MPI possui desempenho favorável para execuções em sistemas de memória distribuída.

3.1.5 Consumo de Energia

A análise do consumo de energia tem se popularizado com a possível chegada da computação *exascale* e uso de sistemas embarcados. Entretanto, poucos trabalhos tem analisado o consumo de energia das Interfaces de Programação Paralela.

Lively et al. (2011) explora o consumo de energia e o desempenho de diferentes implementações paralelas de aplicações científicas. Em particular, o experimento foca na comparação de implementações escritas puramente em MPI e híbridas (MPI/OpenMP), através de dois benchmarks paralelos do NAS: *Block* Tridiagona (BT) e *Gyrokinetic Toroidal Code* (GTC). Para coletar dados do consumo de energia, os autores utilizaram o *framework PowerPack* (GE et al. 2010). Os resultados mostram que para a execução de até 16 *cores*, as aplicações puramente MPI obtiveram melhores ganhos de energia e desempenho. Já para a execução em 32 *cores*, a implementação híbrida (MPI/OpenMP) obteve melhores ganhos de energia e desempenho. No entanto, isto é altamente dependente das características da aplicação. Os autores avaliaram também o uso de *CPU Frequency Scalling*. Neste caso, o cenário que mostrou maior economia e energia não foi o que mesmo que demonstrou melhor desempenho. Por exemplo, a aplicação GTC híbrida que executou com a frequência da CPU em 1.6 GHz (padrão é 1.8 GHz) consumiu menos energia, no entanto o tempo de execução aumentou em 8.62% com relação à versão sem *CPU Frequency Scalling*.

---

[2] Disponíveis em https://software.intel.com/en-us/intel-vtune-amplifier-xe

Balladini et al. (2011) avaliam a influência de diferentes modelos de programação paralela no consumo de energia de sistemas de alto poder de processamento computacional (HPC) e o comportamento do mesmo com diferentes frequências de *clock* da CPU. Para tal, os autores exploraram o OpenMP, representando o paradigma de memória compartilhada; e o MPI com o paradigma de troca de mensagens. Foram utilizadas quatro aplicações paralelizadas em OpenMP e MPI que fazem parte do NAS *Parallel Benchmark*: CG, EP, IS e MG. Os resultados mostram que o modelo de programação possui maior impacto no consumo de energia dos sistemas computacionais. Verificou-se, também, que o impacto da redução da frequência do *clock* da CPU no tempo de execução da aplicação, eficiência energética e potência máxima, não depende somente da característica da aplicação, mas também do modelo de programação paralela utilizado.

Zecena et al. (2012) avaliam o desempenho e o consumo de energia de implementações OpenMP de três tradicionais algoritmos de ordenação: *Odd-Even Sort*, *ShellSort* e *QuickSort*. Os algoritmos *Odd-Even Sort* e *ShellSort* foram implementados de forma iterativa, através do uso de laços paralelos. Já o algoritmo *QuickSort*, que possui como característica principal a recursividade, foi implementado com o uso de tarefas paralelas (*tasks*). Os experimentos foram realizados em um computador com dois processadores *Quad-Core AMD Opteron*™ executando a 2.5 GHz. Na comparação dos três algoritmos, os resultados mostram que a implementação paralela do algoritmo *QuickSort* obteve os melhores resultados de desempenho e economia de energia. Adicionalmente, os autores observaram que a economia de energia pode ser melhor, quando a granularidade de trabalho a ser atribuída para cada *thread* é corretamente selecionada.

Dong et al. (2012) discutem métodos para reduzir o consumo de energia de aplicações OpenMP paralelizadas com laços paralelos através do uso da técnica de redução de consumo de energia DVFS (*Dynamic Voltage and Frequency Scaling*), com certa perda de desempenho. Para tal, os autores propuseram duas otimizações no consumo de energia para os escalonadores *static* e *dynamic* com pouco impacto no desempenho. Foram utilizadas cinco aplicações do NAS *Parallel Benchamrk*: IS, EP, FT, CG e MG com tamanho de entrada correspondente a classe C. Resultados mostram que os algoritmos podem obter redução no consumo de energia sobre diferentes condições do escalonador. No entanto, os autores desconsideraram um fator de extrema importância e que pode alterar todos os resultados: o *overhead* e a localidade de dados imposta pela característica de cada escalonador.

Porterfield et al. (2013) analisam fatores que podem influenciar no desempenho e no consumo de energia de aplicações OpenMP compiladas com ICC e GCC. Através de

contadores de desempenhos de *hardware* presentes na microarquitetura Intel *Sandybridge*, os autores mediram o consumo de energia para uma variedade de programas OpenMP: micro *benchmarks* simples, suíte de *benchmark* com paralelismo de tarefas e uma mini aplicação de hidrodinâmica. A avaliação revela substanciais variações no consumo de energia dependendo do algoritmo, compilador e seu nível de otimização, número de *threads* e a temperatura do chip. Na maioria das aplicações executadas o aumento do número de *threads* permitiu melhor desempenho, porém com aumento substancial no consumo de energia.

3.2 Contexto desta Dissertação

Conforme descrito nas seções anteriores, ao longo da última década diversos trabalhos têm explorado as características das IPPs em termos de facilidade de exploração do paralelismo; controle e flexibilidade do gerenciamento de regiões paralelas; *overhead* na criação e distribuição da carga de trabalho entre *threads*/processos; e desempenho das aplicações. No entanto, podemos notar que a análise da eficiência energética de diferentes IPPs considerando arquiteturas de propósito geral e sistemas embarcados ainda está em sua fase inicial.

O trabalho de Adve (1998) propôs um modelo determinístico para avaliar o desempenho de IPPs. Este modelo provê análise detalhada de sincronização, escalonamento de tarefas, custo de comunicação entre as *threads*/processos, entre outros fatores. Considerando sistemas embarcados, os autores Hanawa (2009) e Lee (2011) mostraram o potencial do uso de OpenMP para melhorar o desempenho de aplicações embarcadas. Em todos os casos, o consumo de energia não foi levado em conta.

Existem poucos trabalhos que investigam o impacto do uso programação paralela no consumo de energia. Técnicas de escalonamento de laços (usando laços paralelos providos pelo OpenMP) em conjunto com *Dynamic Voltage Scaling* têm sido analisadas em Chen (2008), enquanto que os autores em Balladini (2011) compararam OpenMP (memória compartilhada) com MPI (troca de mensagens) executando sobre uma única máquina *multicore*.

A análise da escalabilidade das aplicações paralelas tem sido estudada pelos autores Suleman (2008), Chandha (2012), Joao (2012) e Yavits (2013). Ambos discutem os principais gargalos que prejudicam a eficiência das aplicações paralelas (seções críticas, barreiras,

poucos estágios do *pipeline*, sincronização de dados, entre outros fatores). No entanto, tais análises são realizadas utilizando uma única IPP.

Stanisic (2013) apresenta o estudo inicial do projeto *European Mont-Blanc*. Tal projeto consiste na utilização de processadores com baixo consumo de energia em computação de alto desempenho. O trabalho analisa o desempenho de aplicações e *benchmarks* HPC em plataformas embarcadas de baixo consumo de energia (ARM Cortex-A9). Os resultados são comparados em termos de desempenho, consumo de energia e escalabilidade com sistemas x86 clássicos (Intel Xeon). Embora o estudo/projeto esteja em seu estágio inicial, ele mostra que o uso de arquiteturas embarcadas poderá uma solução para a computação *exascale*.

Os autores em Blem et al. (2013) compararam o impacto de diferentes microarquiteturas e ISAs no desempenho, potência, energia e EDP em sistemas embarcados e de propósito geral. Um grande número de variáveis foi avaliado, como o número de instruções executadas, ciclos de processador, tamanho médio das instruções, número de acessos à memória (*cache misses* e *hits*), tempo de execução entre outros. Entretanto, somente aplicações sequenciais executando em um único processador foram consideradas.

Esta dissertação estende os trabalhos relacionados pela comparação de processadores para sistemas embarcados com processadores para propósito geral e pela investigação do impacto de diferentes IPPs em ambos processadores. Para isso, serão considerados o número de acesso a memória, instruções executadas, desempenho (número de ciclos do processador), consumo de energia, *Energy-Delay Product*, escalabilidade e eficiência dos recursos computacionais; e como cada uma destas variáveis influencia nas demais.

4 METODOLOGIA

Esta seção apresenta a metodologia utilizada neste trabalho. Inicialmente, na Seção 4.1 são apresentadas as principais características do conjunto de *benchmarks* desenvolvido e uma breve descrição das aplicações que compõem tal conjunto. A Seção 4.2 discute a estratégia de paralelização utilizada em cada um dos *benchmarks*. O ambiente de execução escolhido para a análise, bem como as métricas analisadas são apresentadas na Seção 4.3.

4.1 Conjunto de *Benchmarks*

Foram implementadas 12 aplicações (brevemente descritas nas Seções 4.1.1, 4.1.2 e 4.1.3) classificadas em três grupos de acordo com o número de acessos a memória (endereços compartilhados e privados), dependência de dados e sincronização entre *threads*/processos: ***CPU-bound*** **(CPU-B)**, com aplicações que possuem uso intensivo do processador; ***Weakly Memory-bound*** (**WMEM-B**), com aplicações que também utilizam bastante o processador, no entanto, com acessos de leitura a endereços de memória compartilhados entre as *threads*, e que possuem pouca dependência de dados; e ***Memory-bound*** **(MEM-B)**, com aplicações que possuem dependência de dados, sincronização e comunicação entre *threads*/processos, o que impacta em maior número de acessos a memória compartilhada. Optou-se por implementar um novo conjunto de *benchmarks* com as IPPs alvo, pois os diferentes conjuntos de *benchmarks* existentes (NAS *Parallel Benchmark*, *ParMibench*, etc.) não possuem implementações paralelas de todas as aplicações considerando as IPPs alvo deste trabalho.

A Tabela 4.1 apresenta as principais características de cada *benchmark*, os quais foram obtidos através de dados de acesso a memória coletados do Pin Tool (LUK et al. 2005) e análise prévia das aplicações paralelizadas. Nela podem ser observados os números de acesso às memórias compartilhada e privada, considerando leitura, escrita e o total para cada *benchmark*. Dependência de dados significa que existe ao menos uma *thread* que somente irá iniciar sua computação quando o resultado da computação de uma ou mais *threads* estiver pronto, o que implica na existência de comunicação entre *threads*/processos. Pontos de sincronização determinam que em certos pontos de execução da aplicação todas as *threads*/processos devem ser sincronizadas antes de iniciar uma nova tarefa. Por fim, a métrica TLP é usada 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

Tabela 4.1 - Classificação dos Benchmarks

Benchmarks		Escrita (%)		Leitura (%)		Total (%)		TLP			Depend. de Dados	Pontos de Sincr.
		Priv	Comp	Priv	Comp	Priv	Comp	2	3	4		
CPU-B	PI	99.99	0.01	96.30	3.70	98.51	1.49	2.00	3.00	3.92	Não	Não
	CM	99.99	0.01	95.81	4.19	97.21	2.79	1.70	2.77	3.74	Não	Não
	SE	99.99	0.01	93.28	6.71	94.91	5.09	2.00	2.98	3.92	Não	Não
WMEM-B	LU	99.42	0.58	61.69	38.31	63.05	36.95	1.98	2.93	3.66	Sim	Sim
	DJ	99.99	0.01	71.94	28.06	73.86	26.14	2.00	2.96	3.86	Não	Não
	JA	92.99	7.01	54.64	45.36	58.16	41.84	2.00	2.99	3.88	Sim	Sim
	MM	50.40	49.60	58.88	41.12	57.98	42.02	2.00	2.99	3.87	Não	Não
	SH	99.99	0.01	87.17	12.83	89.90	10.10	1.96	2.90	3.70	Não	Não
MEM-B	JV	66.63	33.37	68.34	31.66	68.25	31.75	1.91	2.75	3.33	Sim	Sim
	GS	70.49	29.51	53.98	46.02	56.01	43.99	1.99	2.98	3.84	Sim	Sim
	OE	58.56	41.44	50.39	49.61	52.02	47.98	1.99	2.86	3.39	Não	Sim
	TR	59.93	40.07	82.24	17.76	69.73	30.27	2.00	2.97	3.73	Sim	Não

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 3 para 3 *threads* significa que durante a execução da aplicação, as três *threads* estiveram executando concorrentemente durante 100% do tempo. A equação é definida abaixo:

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

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 *idle* do sistema (nenhuma thread está executando a aplicação). Pode-se notar ainda que as aplicações escolhidas possuem alto grau de exploração de paralelismo no nível de *threads*, o que possibilita melhor obtenção de desempenho através da paralelização. Portanto, neste trabalho, consideraremos aplicações que são paralelizáveis e que possuem ganho de desempenho com relação à versão sequencial.

4.1.1 *CPU-Bound* (CPU-B)

Cálculo do Pi (PI): Este programa calcula o valor de Pi, que é definido como a relação entre o diâmetro e a circunferência de um círculo. Diversos métodos são disponíveis para calcular este valor. Neste trabalho, nós utilizamos o método de Leibniz (ANDREWS,

1999), que dado um número total de iterações, consiste na aproximação do valor de Pi através de sucessivos cálculos internos à uma estrutura de repetição (controlada pelo número de iterações).

Conjunto de *Mandelbrot* (CM): O algoritmo consiste em visitar *pixel* por *pixel* de um fractal (matriz) verificando se o *pixel* (ponto x,y) atual pertence ou não ao Conjunto de *Mandelbrot* após sucessivos cálculos recursivos da distância do pixel para a origem (ponto 0,0). A entrada do algoritmo consiste do tamanho da matriz (altura *x* largura) e da distância máxima para o *pixel* estar presente ou não no conjunto. A saída consiste de uma matriz contendo os pontos que compõem o conjunto de *Mandelbrot*.

Série Harmônica (SE): Consiste de uma série finita que calcula a soma de precisão arbitrária depois do ponto decimal (GOLDSTON, 2000). Tal valor é obtido através da fórmula abaixo:

$$S_n = \sum_{i=1}^{n} \frac{1}{i}$$

Onde n representa o valor da série harmônica a ser calculado. Contém ainda um vetor que armazena a soma da precisão em cada i do algoritmo, e seu principal processamento consiste de sucessivas operações de divisão e cálculo do módulo de cada valor de i.

4.1.2 *Weakly Memory-bound* (WMEM-B)

Decomposição Lower-Upper (LU): Dada uma matriz quadrada A, este algoritmo fatora a matriz A como o produto de uma matriz triangular inferior e uma triangular superior. Nós escolhemos o Método de *Doolittle* para computar a Decomposição LU (PRESS, 2007). O método consiste de, dado uma matriz de entrada A de tamanho *N x N*, através de sucessivas computações em *N* fases sobre os elementos da matriz A, produzir uma matriz triangular inferior (L) e uma superior (U) de saída. Entre cada fase, faz-se necessária prever a troca de dados entre as partes envolvidas.

Dijkstra (DJ): Este algoritmo computa o caminho mais curto entre vértices de um grafo, com custos não negativos para as arestas. Para um dado vértice fonte no grafo, o algoritmo encontra o caminho com menor custo (i.e., o menor caminho) entre este vértice e qualquer outro vértice (DIJKSTRA, 1959). Ele consiste em encontrar a distância entre cada vértice de uma matriz de adjacência de tamanho *N x N*. A saída consiste de um vetor contendo a distância mínima entre cada um dos vértices.

Método de Jacobi (JA): Consiste de um algoritmo iterativo para determinar a solução de sistemas lineares que envolvem uma grande porcentagem de coeficientes nulos. 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, 2007).

Multiplicação de Matriz (MM): Multiplica duas matrizes quadradas (A e B) e armazena o resultado na matriz C. 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 da matriz de saída C. Importante notar na Tabela 4.1 que embora esta aplicação possua grande quantidade de escrita em endereços compartilhados, ela é realizada no final da computação e não representa comunicação entre as *threads*/processos.

Similaridade entre Histogramas (SH): Consiste em encontrar similaridades 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, 2010). A aplicação percorre uma imagem (matriz) *pixel* a *pixel* comparando o histograma dos *pixels* adjacentes com o vetor de histogramas de outra imagem. A saída consiste de uma nova imagem (matriz) indicando quais *pixels* da imagem são similares ao vetor de histogramas de entrada.

4.1.3 *Memory-bound* (MEM-B)

Jogo da Vida (JV): 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 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 da sociedade evoluída após i gerações.

Gram-Schmidt (GS): Consiste de um método para ortonormalizar um conjunto de vetores em um espaço de produto interno, chamado de espaço Euclidiano R_n (CHENEY, 2009). O algoritmo possui uma matriz de entrada (onde cada linha representa um vetor), e através de sucessivas computações em N etapas (as quais devem ser computadas uma após a outra e a computação de N+1 depende do resultado de N), produz uma nova matriz de saída.

Ordenação Par-Impar (OE): Consiste de um algoritmo de ordenação por comparação baseado no *bubble-sort*, que compara pares de elementos e é dividido em duas etapas. Na primeira etapa, os pares indexados são analisados (impar-par). Se o valor do primeiro elemento (impar) é maior que o segundo (par), eles são trocados. A segunda etapa é similar à primeira, porém com o padrão invertido (par-impar). Estas duas etapas são repetidas alternadamente em *N/2* iterações, onde *N* é o número de elementos do vetor unidimensional.

Turing-Ring (TR): Descreve um sistema espacial em que predadores e presas interagem em um mesmo local. O sistema consiste da simulação da iteração e evolução entre presas e predadores através da utilização de equações diferenciais, e a evolução ocorre de acordo com as células vizinhas (PAUDEL, 2011). 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.

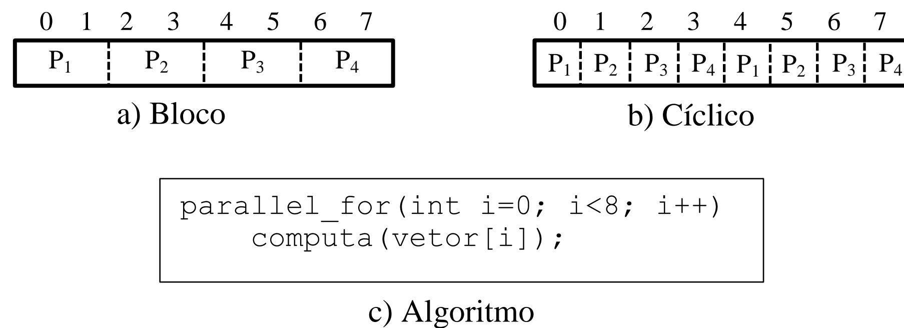
4.2 Paralelização do Conjunto de *Benchmark*

Esta Seção discute a estratégia utilizada na paralelização do conjunto de *benchmark* apresentado na seção anterior com as IPPs Pthreads, OpenMP, MPI-1 e MPI-2. No entanto, os principais detalhes da paralelização de cada aplicação com cada uma das IPPs podem ser encontrados no Apêndice A.

De acordo com Rauber (2010), a paralelização de uma aplicação sequencial pode ser dividida em três etapas: **Decomposição da Computação**, que consiste em decompor um programa sequencial em diferentes instruções que possam ser executadas concorrentemente. **Atribuição das tarefas para processos/*threads***, que determina qual processo/*thread* irá computar cada tarefa (carga de trabalho) definida na etapa de decomposição; e **Mapeamento dos processos/*threads* em unidades físicas de processamento**, realizada pelo algoritmo de escalonamento do sistema operacional.

Sabe-se que a maneira como a aplicação é paralelizada pode influenciar no seu comportamento durante a execução em termos de número de instruções, acesso a memória, desempenho, entre outros fatores, os quais influenciam na eficiência energética. Assim, para manter a coerência entre as implementações com as diferentes IPPs, foram seguidas as direções indicadas por Foster (1995), Wilson (1996), Butenhof (1997), Gropp (1998), Gropp (1999) e Rauber (2010).

Figura 4.1 - Decomposição de Domínio 1D



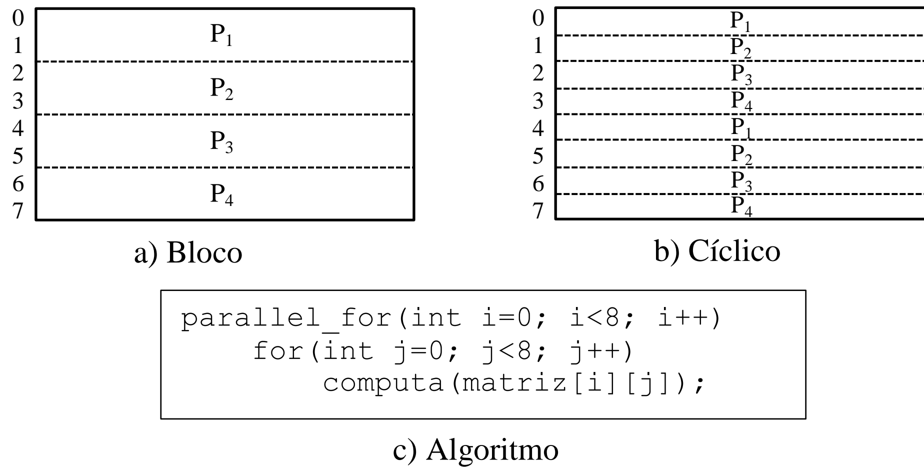
Em específico, na paralelização com Pthreads, MPI-1 e MPI-2, as etapas de decomposição da computação e atribuição das tarefas para processos/*threads* ocorreram de forma explícita, visando o melhor balanceamento da carga de trabalho. Adicionalmente à Pthreads, em MPI-1 e MPI-2 foram incluídas funções de troca de mensagens entre os processos, além da criação dinâmica de processos em MPI-2. Por outro lado, a paralelização com OpenMP foi realizada com o uso de laços paralelos com granularidade fina e grossa. Conforme Foster (1995), Chapman (2008) e Rauber (2010), esta técnica é mais apropriada para paralelizar aplicações que realizam cálculos iterativos e percorrem estruturas de dados contíguos (ex. matriz, vetor, etc.).

As aplicações Ordenação Par-Impar e Série Harmônica realizam computação sobre estruturas de dados com uma única dimensão (vetores). Portanto, para a sua paralelização foi utilizada a decomposição de domínio em blocos de uma dimensão (1D), conforme apresentado na Figura 4.1. Neste modelo, cada *thread*/processo computa sobre diferentes elementos do vetor, que são distribuídos em forma de blocos de iterações (Figura 4.1a); ou cíclica (Figura 4.1b). A diferença entre os dois modos está na forma como as iterações são atribuídas para as *threads*. Enquanto na divisão por blocos as iterações são atribuídas de uma única vez, na cíclica elas são atribuídas conforme solicitação.

Um possível algoritmo para este modelo de decomposição é apresentado na Figura 4.1c. 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, MPI-1 e MPI-2, foi utilizada a distribuição em blocos (Figura 4.1a) e realizado ajustes finos na divisão. Já na paralelização com OpenMP foi utilizada a distribuição em blocos para granularidade grossa e cíclica para granularidade fina (Figura 4.1b).

As aplicações Conjunto de *Mandelbrot*, *Dijkstra*, Método de Jacobi, Multiplicação de Matriz, Similaridade entre Histogramas, *Turing Ring* e Jogo da Vida computam sobre

Figura 4.2 - Decomposição de Domínio 2D



estruturas de dados bidimensionais (matrizes). Assim, a estratégia de paralelização utilizada foi a decomposição de domínio em blocos de duas dimensões (2D), apresentada na Figura 4.2. O algoritmo deste modelo é apresentado na Figura 4.2c. Ele mostra que o laço *for* a ser paralelizado é o mais externo (*parallel_for*), ou seja, o que controla a distribuição das linhas entre as *threads/*processos.

Novamente, na paralelização com Pthreads, MPI-1 e MPI-2, foi usada a decomposição em bloco (Figura 4.2a) e realizado ajustes finos na paralelização. Já na paralelização com OpenMP foi utilizada a distribuição em blocos para granularidade grossa e cíclica para granularidade fina (Figura 4.2b).

Embora as aplicações Decomposição-LU e Gram-Schmidt também computem sobre matrizes, elas foram paralelizadas utilizando abordagens definidas por Wilson (1996) e Rauber (2010), devido a suas limitações de dependência de dados e sincronização entre os processos/*threads* e são explicadas com maiores detalhes no Apêndice A. Por fim, a paralelização envolvendo o Cálculo do Pi ocorreu através da divisão do número de iterações pelo número de *threads*/processos.

4.3 Ambiente de Execução

4.3.1 Arquiteturas Alvo

Os experimentos foram realizados em quatro diferentes processadores: Intel Core i7 e Core2Quad, representando os processadores de propósito geral; e Intel Atom e ARM Cortex-

Tabela 4.2- Principais características dos processadores

	Core i7 860	**Core2Quad Q8400**	**Atom N2600**	**Cortex-A9**
Frequência	3.4 GHz	2.66 GHz	1.6 GHz	1.2 GHz
Tecnologia	45 *nm*	45 *nm*	32 *nm*	40 *nm*
# de Núcleos	4	4	2	4
# de Threads	8	4	4	4
Cache L1 Dados/Instrução	32/32 KB	32/32 KB	24/32 KB	32/32 KB
Cache L2	4 x 256 KB	2 x 2 MB	2 x 512KB	1 MB
Cache L3	8 MB	----	----	----
Memória RAM	8 GB	4 GB	2 GB	1 GB
Plataforma	Propósito Geral	Propósito Geral	Embarcada	Embarcada

A9, representando os sistemas embarcados. As principais características de cada processador e a hierarquia do sistema de memória são apresentadas na Tabela 4.2 e na Figura 4.3, respectivamente.

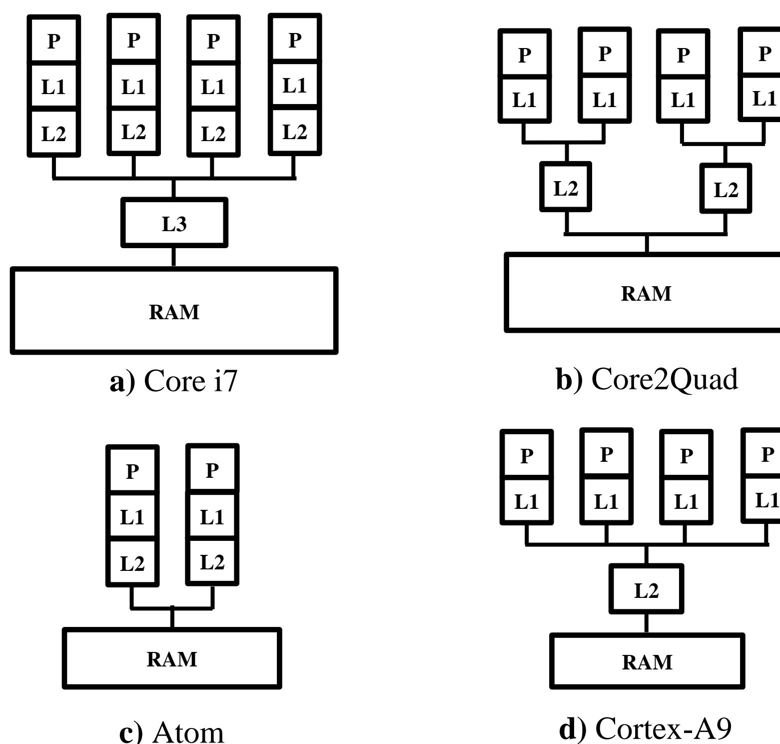
O processador Intel Core i7 860 é equipado com quatro *cores* operando na frequência de 2.8 GHz com suporte a *Hyperthreading*. Conforme pode ser visto na Figura 4.3a, seu sistema de memória é formado por três níveis de *cache*: uma *cache* L1 de dados e outra de instruções com tamanho de 32 KB cada, privada para cada *core*; uma *cache* L2 unificada (dados e instruções) com tamanho de 256 KB, também privada para cada *core*; e uma *cache* L3 unificada e compartilhada entre todos os *cores* com tamanho de 8 MB. A memória principal (RAM) tem tamanho de 8 GB e tecnologia DDR3.

O processador Intel Core2Quad também possui quatro *cores*, porém, operando na frequência de 2.66 GHz e sem suporte a *HyperThreading*. Seu sistema de memória, ilustrado na Figura 4.3b, consiste em 2 níveis de memória *cache*: *cache* L1 privada e separada em dados e instruções com tamanho de 32 KB cada; e duas *caches* L2, cada uma com tamanho de 2 MB e compartilhada entre cada par de *core*. Possui memória principal de 4 GB e tecnologia DDR3.

Já o processador da Intel Atom N2600 possui somente dois *cores* operando na frequência de 1.6 GHz e com suporte a *Hyperthreading*. O sistema de memória, apresentado na Figura 4.3c, contém os dois níveis de *cache* privados: L1 de dados com tamanho de 24 KB e de instruções com tamanho de 32 KB; e *cache* L2 unificada de tamanho 512 KB. A memória principal tem 2 GB e tecnologia LPDDR3, que é projetada especificamente para o baixo consumo de energia, visto que são utilizadas em dispositivos móveis, tais como *smartphones*, *tablets* e *ultrabooks*.

O processador ARM Cortex-A9 possui quatro *cores* operando na frequência de 1.2 GHz, sem suporte à execução simultânea de duas *threads* por *core* (SMT). A Figura 4.3d ilustra o seu sistema de memória, que é composto de dois níveis de memória *cache*: L1

Figura 4.3 - Organização Hierárquica dos Processadores Alvo



a) Core i7 **b)** Core2Quad **c)** Atom **d)** Cortex-A9

separada de dados e instruções com tamanho de 32 KB, privadas; e uma *cache* L2 unificada e compartilhada entre todos os *cores,* com tamanho de 1 MB. A memória principal tem 1 GB e tecnologia LPDDR3.

4.3.2 Extração de Dados e Métricas Analisadas

A análise das Interfaces de Programação Paralela e das arquiteturas alvo foi realizada com base nos dados obtidos de número de acessos à memória (*cache* e principal), número de instruções executadas e o total de ciclos que cada aplicação levou para executar. Para a obtenção destes dados, foi utilizada a ferramenta PAPI (BROWNE, 2000). Ela consiste de um conjunto de componentes que permite a extração de dados de contadores de *hardware* do processador com a finalidade de avaliar o comportamento da aplicação. PAPI tem sido amplamente utilizado pela comunidade acadêmica nos últimos anos, como em Johnson (2012), John (2013), Weaver (2013), McCraw (2013).

Os dados extraídos através do PAPI foram utilizados para calcular e analisar os seguintes eixos:

Desempenho: O tempo total de execução de cada aplicação foi obtido através da divisão do número total de ciclos que a aplicação levou para executar pela frequência do processador. Como o PAPI coleta informações diretamente de contadores de *hardware*, os dados obtidos não possuem a influência do sistema operacional (como por exemplo, impacto de trocas de contexto). Assim, todos os dados são somente da aplicação sem interferência externa.

Eficiência dos Recursos Computacionais: Eficiência é aqui definida como o melhor uso possível dos processadores disponíveis, considerando a execução sequencial da aplicação em um único *core* como base. Por exemplo, um programa que é dividido em 4 *threads* pode ser mais rápido e menos eficiente que um programa que é dividido em somente 2 *threads*. Neste caso, cada um dos quatro processadores foi usado menos (e.g.: eles gastaram mais tempo aguardando em barreiras de sincronização) durante a execução do programa que cada um dos dois processadores. Para este cálculo, a seguinte fórmula foi usada:

$$R_{ef} = \frac{R_{Seq}}{R_{Par}\ x\ NT}$$

Onde R_{Seq} corresponde ao uso do processador executando a versão sequencial da aplicação, R_{Par} é o mesmo, porém para a versão paralela, e NT é o número de *threads*/processos da aplicação.

Consumo de energia: Para estimar o consumo total de energia (E_{Total}), nós estamos levando em conta a energia consumida das instruções executadas (E_{Ins}), acesso a memória cache e principal (E_{Mem}) e a energia estática (E_{est}), conforme a seguinte equação:

$$E_{Total} = E_{Ins} + E_{Mem} + E_{Est}$$

Para obter a energia consumida pelas instruções executadas, a seguinte equação foi utilizada

$$E_{Ins} = I_{Exe}\ x\ P_{Ins}$$

onde E_{Ins} corresponde ao número de instruções executadas (I_{Exe}) multiplicado pelo consumo médio (P_{Ins}) para executar cada instrução. O total de energia consumida no sistema de memória foi obtido através da equação abaixo:

$$E_{Mem} = (CacheDados_{Total} x\ P_{CD}) + (CacheInst_{Total}\ x\ P_{CI}) + (Miss_{Total}\ x\ P_{RAM})$$

onde ($CacheDados_{Total}\ x\ P_{CD}$) e ($CacheInst_{Total}\ x\ P_{CI}$) correspondem ao total de energia consumida em acessos à memória cache de dados e de instruções respectivamente. Nela, P_{CD} e P_{CI} representam a energia gasta por acesso a cache de dados e de instrução respectivamente.

Tabela 4.3- Dados de energia de cada processador

	Core i7 860	Core2Quad Q8400	Atom N2600	Cortex-A9
Potência Média Processador	21.95 W	21.95 W	2.42 W	1.25 W
Potência Estática Processador	4.39 W	4.39 W	0.484 W	0.250 W
Potência Estática RAM	0.720 W	0.360 W	0.149 W	0.120 W
Energia média por instrução executada	1.26 nJ	1.26 nJ	0.413 nJ	0.290 nJ
Energia acesso L1-D	0.017 nJ	0.017 nJ	0.014 nJ	0.017 nJ
Energia acesso L1-I	0.017 nJ	0.017 nJ	0.017 nJ	0.017 nJ
Energia acesso RAM	24.6 nJ	15.6 nJ	3.94 nJ	2.77 nJ

Já a energia gasta para acessos a memória principal é dada por ($Miss_{Total}$ *x* P_{RAM}), onde $Miss_{Total}$ representa a quantidade de cache miss e consequente acesso a memória principal e P_{RAM} a energia gasta por acesso a memória RAM.

Por fim, a equação abaixo foi usada para estimar o consumo de energia estática de cada componente. Como a energia estática é consumida enquanto o componente estiver ligado, este valor deve ser considerado durante o tempo total de execução da aplicação, que corresponde ao total de ciclos executados (*#Ciclos*) da aplicação dividido pela frequência de operação do processador (*freq.*). Para este cálculo, nós consideramos o consumo estático do processador (P_{CPU}), memória cache de dados (P_{CD}) e instruções (P_{CI}) e memória RAM (P_{RAM}).

$$E_{Est} = \frac{\#Ciclos}{freq} \, x \, (P_{CPU} + P_{CD} + P_{CI} + P_{RAM})$$

Para calcular o consumo de energia, utilizaram-se dados providos por Blem (2013), Cacti Tool (CACTI, 2013) e manuais dos processadores disponíveis em ARM (2010), Intel (2009), Intel (2010) e Intel (2011). Tais dados são apresentados na Tabela 4.3.

Energy-Delay Product: Esta métrica é usada para estudar o custo-benefício entre o total de energia consumida e o tempo de execução de uma aplicação. Sua fórmula, que consiste da multiplicação da energia consumida (*Energia*) pelo tempo de execução (*Tempo*), é mostrada abaixo.

$$EDP = Energia \; x \; Tempo$$

Ela vem sendo bastante utilizada pois possibilita analisar, em um único valor, a relação entre o consumo de energia e o desempenho. Por exemplo, considerando dois cenários: (*i*) uma aplicação é executada em 100 segundos e consumiu 10 joules de energia; (*ii*) uma aplicação é executada em 50 segundos e consumiu 40 joules de energia; o cenário *i* teria EDP de 1000. enquanto que o cenário *i* teria EDP de 2000. Isso mostra que embora o cenário *i*

tenha sido 2 vezes mais lento, ele possui a melhor relação de consumo de energia e desempenho.

Escalabilidade: Consiste em analisar o comportamento das IPPs e das arquiteturas alvo em termos de desempenho, consumo de energia e EDP quando o grau de paralelismo é aumentado (i.e.: quando mais *threads*/processos executam concorrentemente a mesma aplicação).

5 RESULTADOS

Este capítulo apresenta a comparação entre os processadores Intel Core i7, Core2Quad, Atom e ARM Cortex-A9 considerando as diferentes Interfaces de Programação Paralela alvo, em termos de: número de acessos à memória de dados e instruções executadas (Seção 5.1); desempenho (Seção 5.2); consumo de energia (Seção 5.3), *Energy-Delay Product* (Seção 5.4) e escalabilidade da eficiência dos recursos computacionais (Seção 5.5).

Os resultados apresentados neste capítulo correspondem à execução de cada *benchmark* com o tamanho de entrada descrito na Tabela 5.1. Os programas foram divididos em 2, 3 e 4 *threads*/processos para manter a coerência entre os processadores (todos podem executar, em *hardware*, 4 *threads* simultaneamente). O compilador usado em ambos os processadores foi o GCC 4.7.3 sem *flags* de otimização. A distribuição MPI usada foi a OpenMPI 1.6, enquanto que a versão do OpenMP foi a 3.0 e Pthreads a versão POSIX.1-2008. Para aumentar o universo de comparações, as versões OpenMP foram executadas com diferentes tamanhos de *chunk* (Fina – 1 única iteração é atribuída para cada *thread* em cada solicitação ao escalonador; e Grossa – onde o número de iterações é dividido pelo número de *threads* e atribuído para as mesmas de uma única vez pelo escalonador). Para todas as execuções, o sistema operacional *Linux* (*Debian*) esteve sempre sob as mesmas condições, ou seja, após cada execução ele foi reiniciado para evitar que as aplicações aproveitassem dados restantes na memória *cache* das execuções anteriores. Os resultados apresentados são a média de 10 execuções. Para todos os cenários, o desvio padrão foi sempre inferior a 1% do resultado obtido, o que justifica o baixo número de repetições.

Tabela 5.1- Tamanho de Entrada de cada *Benchmark*

Benchmarks		Tamanho de Entrada
CPU-B	**Calculo do Pi**	4 bilhões de iterações
	Conjunto de *Mandelbrot*	Matriz 1024 x 768 e 14000 iterações
	Série Harmônica	Vetor com 100000 elementos
WMEM-B	**Decomposição-LU**	Matriz 2048 x 2048
	Dijkstra	Matriz 2048 x 2048
	Método de Jacobi	Matriz 2048 x 2048
	Multiplicação Matriz	Matriz 2048 x 2048
	Similaridade Histogramas	Imagem 1920 x 1080
MEM-B	**Jogo da Vida**	Matriz 4096 x 4096
	Gram-Schmidt	Matriz 2048 X 2048
	Ordenação Par-Impar	Vetor com 150000 elementos
	Turing Ring	Matriz 2048 x 2048

5.1 Número de Instruções Executadas e Acessos a Memória de Dados

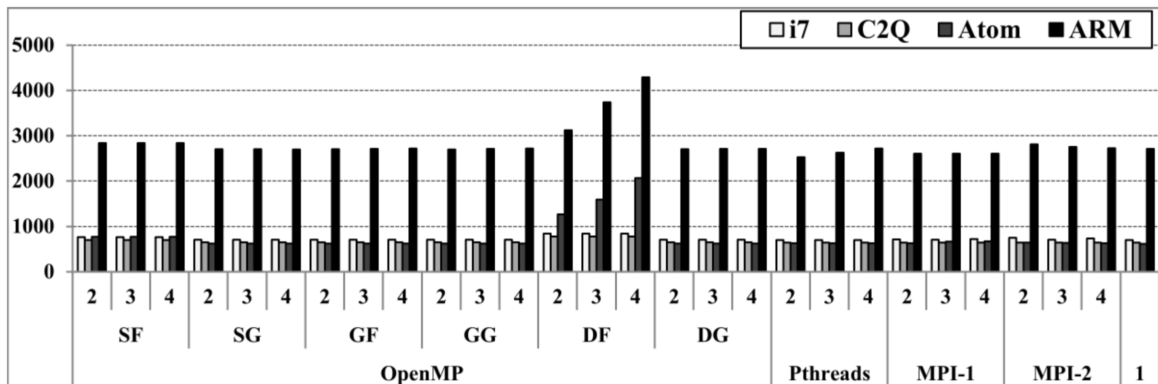
O número de instruções executadas mostra o impacto que cada Interface de Programação Paralela impôs com relação ao número total de instruções executadas na versão sequencial. Basicamente, o impacto é resultado das tarefas de criação de *threads*/processos; da divisão da carga de trabalho; da comunicação e sincronização entre *threads*/processos. Da mesma forma, o número de acessos à memória de dados compartilhada tem grande importância, pois reflete na taxa de comunicação entre as *threads*/processos.

As Figura 5.1, 5.2 e 5.3 apresentam os resultados da soma do número de instruções executadas e da quantidade de acessos à memória de dados para as aplicações CPU-B, WMEM-B e MEM-B. Por exemplo, os gráficos para as aplicações CPU-B contêm a soma dos resultados das aplicações Cálculo do Número Pi, Conjunto de *Mandelbrot* e Série Harmônica, e o mesmo ocorre para as aplicações WMEM-B e MEM-B. Os resultados de cada aplicação podem ser encontrados no Apêndice B. O eixo *x* de cada gráfico corresponde ao número de *threads*/processos e a Interface de Programação Paralela. As políticas de escalonamento do OpenMP estão abreviadas para: S (*Static*), G (*Guided*) e D (*Dynamic*); enquanto que a granularidade grossa está abreviada para “G”; e fina para “F”. “1” significa a execução sequencial. O processador Intel Core i7 está abreviado para i7, Intel Core2Quad para C2Q, Intel Atom para Atom e ARM Cortex-A9 para ARM.

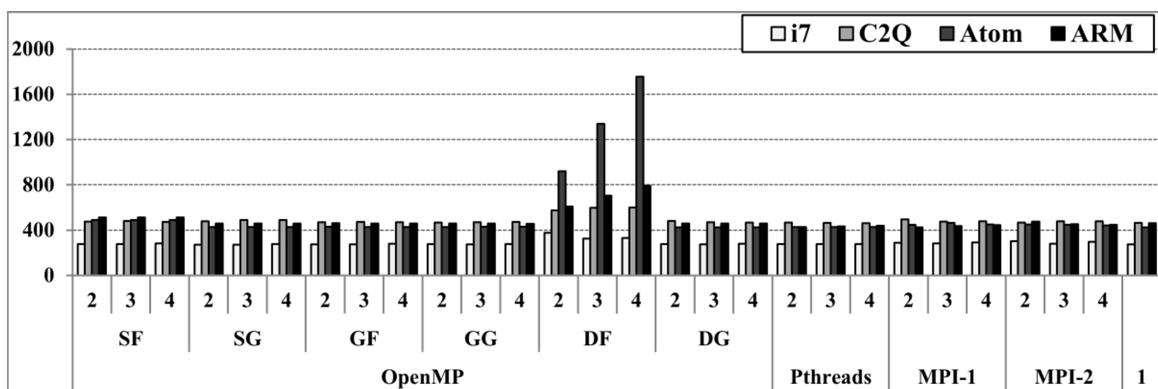
5.1.1 Análise das Aplicações CPU-B

Como pode ser observado nos resultados das aplicações *CPU-Bound* (CPU-B), mostrados na Figura 5.1, na maioria dos casos, a paralelização não impôs custo adicional representativo no número de instruções executadas (Figura 5.1a) e acessos à memória de dados (Figura 5.1b) com relação à versão sequencial. Ademais, o resultado entre as diferentes IPPs é muito similar devido à pequena quantidade de comunicação entre as *threads*/processos. A exceção é para os *benchmarks* implementados usando OpenMP com a política de escalonamento *dynamic* e granularidade fina (DF) em todos os processadores. Cabe ressaltar que, dentre os *benchmarks*, o que mais apresentou aumento na quantidade de instruções e acessos à memória foi o Cálculo do Número Pi, conforme pode ser conferido no Apêndice B (Seção B.1). Nestes casos, quando a carga de trabalho atribuída para cada *thread* é muito pequena, o *overhead* causado pelo gerenciador do escalonador do OpenMP para controlar tal

Figura 5.1 - Resultados CPU-B



a) Número de Instruções Executadas (em Bilhões)



b) Número de Acessos a Memória de Dados (em Bilhões)

distribuição em tempo de execução é muito alto (CHAPMAN, 2008). Este impacto é maior nos processadores Intel Atom e ARM Cortex-A9, pois conforme verificado em testes adicionais usando o próprio PAPI, eles possuíram maior número de falhas na TLB (*Translation Lockaside-Buffer*) que nos demais, e isto impactou em maior número de instruções de *busy-waiting* pelas *threads* que estavam aguardando nos pontos de sincronização.

Ainda na Figura 5.1a, pode-se notar que o processador ARM Cortex-A9 executou maior número de instruções que os demais processadores (média de 3,5 vezes). Isto se dá devido ao diferente comportamento dos tipos de dados (*long int* e *long long int*) e operações específicas (*mod, sqrt, sqrtf* e recursividade) presentes nas aplicações desta classe. Conforme mostrado na Tabela 5.2, o ARM precisa executar mais instruções para realizar tais operações tomando como *baseline* o processador Intel Core i7. Enquanto nos processadores Intel, as operações *sqrt* e *sqrtf* são executadas em apenas uma única instrução específica para tal, no ARM Cortex-A9 utilizado neste trabalho, a biblioteca matemática (cabeçalho *math.h*) insere um algoritmo para cálculo da raiz quadrada que executa maior número de instruções.

Tabela 5.2 - Diferença no Número de Instruções Executadas (*baseline* Core i7)

Tipo	Operação	Core i7	Core2Quad	Atom	ARM Cortex-A9
Long Long Int	**Div**	1.00	1.00	0.90	3.22
Long Int	**Mod**	1.00	1.00	0.90	3.97
SQRT		1.00	0.33	0.51	1.70
SQRTF		1.00	0.39	0.70	8.84
Recursividade		1.00	1.00	3.15	2.18

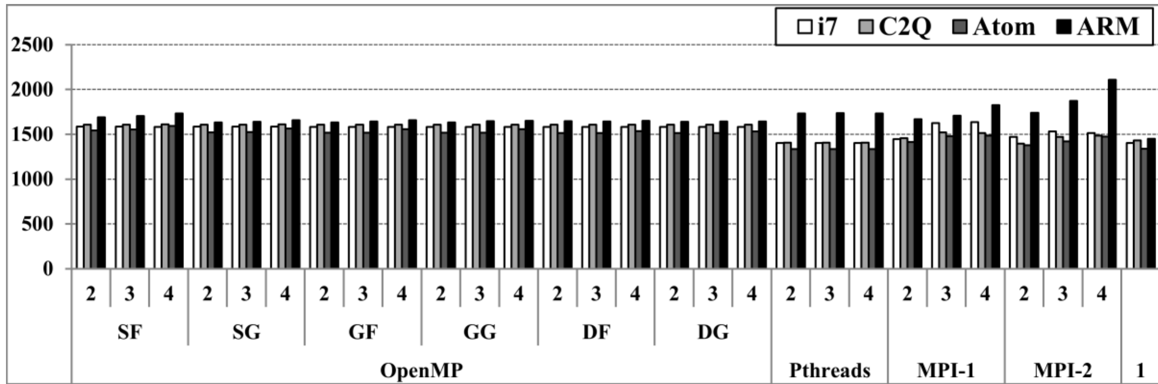
5.1.2 Análise das Aplicações WMEM-B

No resultado da classe *Weakly Memory-Bound* (WMEM-B) mostrado na Figura 5.2, pode-se notar que as IPPs possuem comportamento diferente com relação ao número de instruções executadas (Figura 5.2a). Pthreads foi melhor nos processadores Intel, ou seja, executou menor número de instruções, o qual é similar ao resultado da versão sequencial; enquanto que OpenMP foi melhor no ARM. Esta diferença no comportamento das IPPs OpenMP e Pthreads entre os processadores está relacionada com o modo que as variáveis locais (declaração interna a *main*) e globais (declaração externa a *main*) são tratadas pelos compiladores em cada arquitetura. Enquanto que a declaração das variáveis locais no OpenMP possuiu pior comportamento nos processadores Intel, ou seja, executou maior número de instruções, o contrário ocorreu no ARM. Já em Pthreads, a utilização de variáveis globais possuiu melhor comportamento nos processadores Intel que no ARM, onde executou em média 19.6% mais instruções que a versão sequencial. Ademais, com relação à quantidade de acessos a memória de dados (Figura 5.2b), Pthreads foi melhor em todos os processadores.

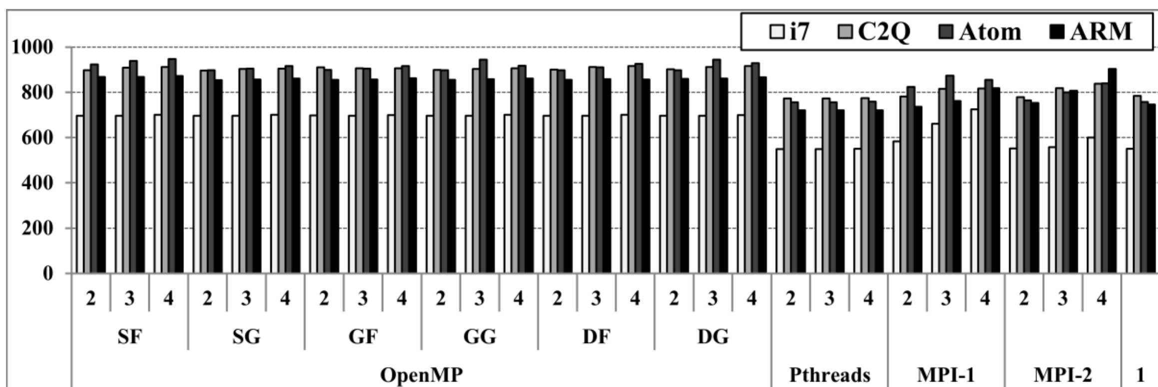
Nas comparações envolvendo o número de instruções executadas no MPI (Figura 5.2a), podem-se observar dois comportamentos diferentes: MPI-2 foi melhor que MPI-1 nos processadores Intel; e pior que MPI-1 no ARM. A criação dinâmica de processos resultou em menor número de instruções executadas em todos os processadores, pois os processos somente são criados e inicializados quando há computação útil para ser realizada. No entanto, verificou-se que, no processador ARM, o impacto da comunicação através de intercomunicadores (pai e filho, por exemplo) foi maior que o ganho obtido com a criação dinâmica de processos.

Com relação ao número de acessos a memória no MPI (Figura 5.2b), MPI-1 realizou, em média, 6% mais acessos a memória de dados que MPI-2 nos processadores Intel, enquanto que no processador ARM, MPI-1 realizou 6% menos acessos que MPI-2. Pode-se notar,

Figura 5.2 - Resultados WMEM-B



a) Número de Instruções Executadas (em Bilhões)



b) Número de Acessos a Memória de Dados (em Bilhões)

ainda, que conforme o grau de exploração do paralelismo aumenta em MPI (por exemplo, 2 para 4 processos), também aumenta o número de instruções executadas e de acessos a memória, devido ao aumento do número de operações de comunicação entre os processos (*MPI_Send* e *MPI_Recv*, por exemplo).

Nos resultados obtidos com OpenMP, ainda para as aplicações WMEM-B, o impacto do número de instruções executadas com relação a versão sequencial foi em média 13,5% em todos os processadores (Figura 5.2a). Já para o número de acessos a memória de dados, o impacto médio aumentou para 17,4% (Figura 5.2b). Embora o comportamento das variáveis globais possa ter contribuído para esta diferença com relação a versão sequencial, ela é principalmente explicada pela técnica de sincronização entre as *threads* OpenMP. Conforme discutido na Seção 2.3.1, quando as *threads* OpenMP sincronizam de forma explícita ou implícita, elas entram em estado de *busy-waiting* (executando instruções de acesso a memória constantemente). O tempo que cada *thread* permanece neste estado depende de vários fatores, tais como:

Desbalanceamento de Carga: Quando a carga de trabalho de um programa paralelo não é corretamente distribuída entre os processadores, alguns processadores irão aguardar

enquanto outros finalizam suas tarefas. No caso do OpenMP, *threads* com baixa carga de computação irão aguardar em estado de *busy-waiting* até que as outras *threads* completem suas tarefas.

Pontos de Sincronização: Maior número de pontos de sincronização implica em maior tempo das *threads* em estado de *busy-waiting*; e

Número de *Threads*: Um alto número de *threads* executando implica em mais *threads* para sincronizar implicitamente (*fork*/*join*) e explicitamente (*mutexes*).

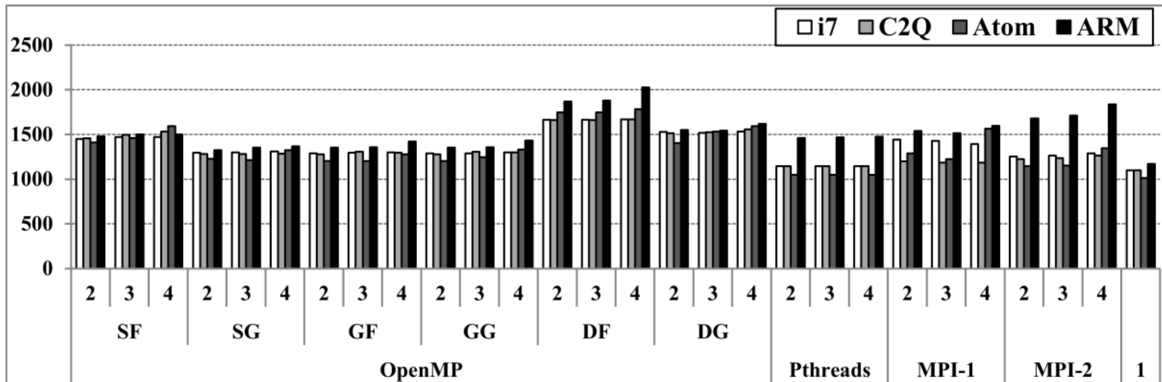
5.1.3 Análise das Aplicações MEM-B

A classe *Memory-Bound* (MEM-B) possuiu maior impacto com relação à versão sequencial dentre as três classes, devido à característica de comunicação intensa. Conforme mostrado na Figura 5.3a, nos processadores Intel, Pthreads novamente executou menos instruções que as demais IPPs, com média de aproximadamente 4% de impacto sobre a versão sequencial. Já no processador ARM, Pthreads executou aproximadamente 25,2% mais instruções que a versão sequencial. Com relação ao número de acessos a memória (Figura 5.3b), o impacto de Pthreads foi de aproximadamente 4,6% com relação à versão sequencial nos quatro processadores. Esta diferença do comportamento de Pthreads nos processadores Intel e ARM mostra que os processadores Intel executam menos instruções para gerenciar as variáveis globais (utilizadas somente em Pthreads) que o ARM.

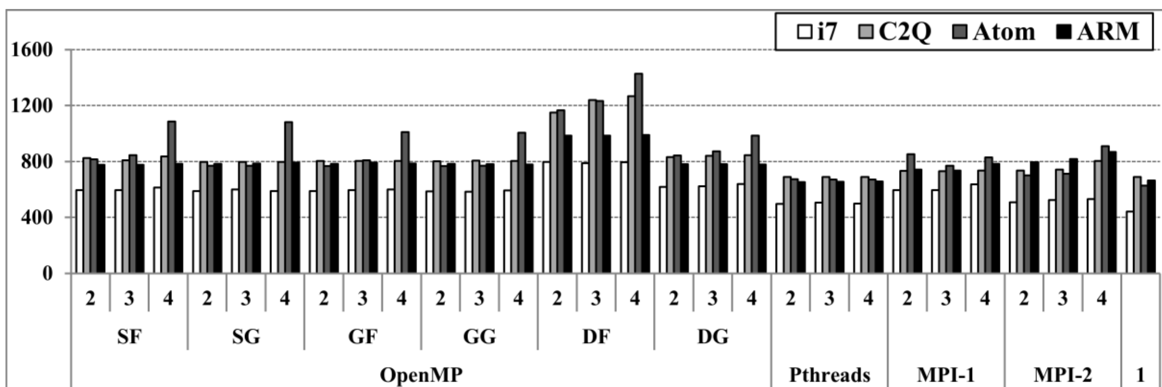
Na comparação com MPI, novamente o MPI-2 executou menos instruções e acessou menos a memória de dados que o MPI-1 nos processadores Intel (Figura 5.3a e Figura 5.3b). Por outro lado, no processador ARM, as comunicações através de intercomunicadores novamente impactaram de forma negativa no MPI-2, com a execução de aproximadamente 12% instruções e 10% acessos à memória de dados a mais que MPI-1.

Analisando-se ainda a Figura 5.3a, nas comparações do número de instruções executadas envolvendo OpenMP, as políticas de escalonamento SG, GF e GG obtiveram os melhores resultados no processador ARM com impacto de aproximadamente 18% com relação a versão sequencial. Pode-se notar que a simples alteração da política de escalonamento pode causar impacto diferente no número de instruções executadas e acessos a memória de dados, levando a desperdício dos recursos computacionais. Por exemplo, A utilização da política de escalonamento DF (*dynamic* com granularidade fina) possui impacto de pouco mais de três vezes maior que a política SG (60% e 18% respectivamente) no número

Figura 5.3 - Resultados MEM-B



a) Número de Instruções Executadas (em Bilhões)



b) Número de Acessos a Memória de Dados (em Bilhões)

de instruções executadas e aproximadamente três vezes na quantidade de acessos a memória (76% para 26% respectivamente).

A diferença entre as políticas de escalonamento do OpenMP está relacionada ao determinismo da distribuição das iterações. Por exemplo, na política *static*, a distribuição é determinística, ou seja, sempre que uma *thread* computar sobre a mesma estrutura de dados em pontos diferentes do código, ela computará sob as mesmas posições (iterações). Por outro lado, na política *dynamic* a distribuição é não determinística, visto que a atribuição das iterações é definida em tempo de execução de acordo com a solicitação de cada *thread*. A distribuição não determinística pode levar a uma maior quantidade de *cache misses* (resultados apresentados no Apêndice B para cada aplicação), pois as *threads* poderão computar sobre dados que antes estavam sendo computados por outras *threads* e assim por diante.

5.1.4 Análise Crítica

Os resultados apresentados nesta seção mostraram que as Interfaces de Programação Paralela possuem comportamento diferente no número de instruções executadas e quantidade de acessos a memória, e que esse comportamento pode variar dependendo do tipo de aplicação e do processador utilizado. Observou-se que para aplicações com uso intensivo da CPU e pouca comunicação (CPU-B), as IPPs possuem comportamento similar. Para o número de instruções executadas das aplicações WMEM-B e MEM-B, Pthreads surge como a melhor opção para os processadores Intel com pequeno impacto com relação a versão sequencial para as aplicações WMEM-B e MEM-B; enquanto que o OpenMP foi o melhor para o ARM nas mesmas aplicações. Considerando o número de acessos a memória de dados nestas duas classes, Pthreads obteve os melhores resultados em todos os processadores. Ademais, mostrou-se que enquanto o MPI-1 surge como melhor opção para o processador ARM, o MPI-2 foi melhor para os processadores Intel em termos de instruções executadas e acessos à memória de dados. Também mostrou-se que o uso de diferentes políticas de escalonamento do OpenMP podem provocar impacto negativo na aplicação, se não aplicadas adequadamente.

5.2 Desempenho

Os resultados apresentados na Tabela 5.3 mostram o tempo total que cada processador levou para executar todas as aplicações de cada classe e ela será utilizada como base para as análises nas duas próximas seções. Os resultados individuais de cada aplicação estão no Apêndice B.

5.2.1 Análise das Arquiteturas

Como pode ser observado na Tabela 5.3, Intel Core i7 (i7) foi melhor na maioria dos casos, ou seja, executou o conjunto de *benchmark* em menos tempo. Isto já era esperado, pois ele é um processador mais robusto e possui maior frequência de operação (3.4 GHz comparado com 2.66 GHz do Core2Quad (C2Q); 1.6 GHz do Atom; e 1.2 GHz do ARM Cortex-A9).

Para as aplicações com uso intensivo da CPU (CPU-B), Intel Core i7 foi aproximadamente 1,3 vezes mais rápido que o Core2Quad; 7,4 vezes mais rápido que o

Tabela 5.3 – Tempo Total de Execução (em segundos)

			CPU-B				WMEM-B				MEM-B			
			i7	**C2Q**	**Atom**	**ARM**	**i7**	**C2Q**	**Atom**	**ARM**	**i7**	**C2Q**	**Atom**	**ARM**
Sequencial			298.6	370.4	1778.6	2209.7	458.3	750.9	3059.6	3257.8	219.3	240.0	900.5	1092.2
OMP	**SF**	**2**	151.3	185.8	119.4	1271.8	270.6	468.9	1907.6	1773.0	161.7	217.0	765.0	780.6
		3	99.7	132.2	793.7	931.0	173.1	332.0	1327.0	1222.6	115.6	162.9	614.7	553.9
		4	75.5	97.8	597.7	744.6	133.0	275.1	1005.8	1050.7	98.6	135.2	1108.1	462.0
	SG	**2**	150.7	186.9	1080.4	1234.9	2669	467.2	1885.6	1740.4	163.5	142.6	698.8	666.1
		3	115.7	144.4	895.6	1080.4	174.2	321.5	1312.0	1195.7	113.4	95.9	489.7	449.1
		4	84.4	105.0	656.4	835.8	128.8	277.7	991.5	1048.3	94.5	72.8	1057.2	344.7
	GF	**2**	149.2	186.2	1053.9	1235.8	258.5	468.9	1939.6	1742.7	141.7	154.3	659.4	670.8
		3	104.3	133.7	800.4	955.0	172.8	330.5	1306.0	1222.7	97.2	110.4	482.3	454.7
		4	84.8	97.6	601.4	759.6	132.5	269.5	1079.5	1060.6	86.2	85.2	845.1	357.6
	GG	**2**	149.2	186.3	1045.2	1235.3	264.7	463.5	1895.5	1748.1	170.2	157.8	655.9	670.9
		3	104.7	132.0	800.0	961.2	171.7	330.5	1298.3	1210.7	95.5	110.3	479.5	454.7
		4	84.7	102.2	600.1	762.4	131.4	275.7	1013.3	1065.6	82.1	85.0	812.5	357.0
	DF	**2**	241.7	519.6	2460.2	1508.6	263.2	468.4	1838.7	1726.0	436.0	459.8	1079.2	1217.3
		3	180.4	449.5	2376.4	1313.2	170.0	331.8	1280.5	1181.9	398.5	13491	1523.4	902.0
		4	177.1	414.3	2182.4	1190.8	130.3	275.2	991.3	1044.9	388.8	1412.7	1440.8	781.0
	DG	**2**	152.8	186.8	1052.1	1238.3	252.5	467.6	1840.4	1718.5	219.9	236.8	917.6	754.3
		3	99.3	143.4	863.6	927.4	168.0	334.7	1369.1	1186.9	147.1	170.6	700.4	538.8
		4	88.6	105.3	646.9	742.2	130.2	276.1	993.7	1023.2	126.1	130.8	743.1	441.6
Pthreads		**2**	148.2	191.3	1029.0	1118.3	242.1	428.3	1699.2	1752.4	129.4	121.5	484.8	767.7
		3	99.4	127.4	744.8	852.2	161.9	294.9	1263.1	1173.7	94.6	82.3	328.9	517.4
		4	74.9	97.0	576.6	719.5	122.4	238.6	879.6	1021.9	76.7	62.2	248.5	392.5
MPI-1		**2**	157.5	185.4	1076.5	1297.1	257.7	431.0	1754.9	1800.6	142.9	133.6	706.4	856.3
		3	106.9	128.3	779.1	917.2	166.0	323.7	1260.5	1359.0	101.4	89.6	554.1	569.9
		4	79.0	95.2	595.3	749.4	142.7	267.6	1241.9	1261.8	77.1	71.3	854.4	478.6
MPI-2		**2**	175.3	185.8	1071.6	1288.2	252.8	426.3	1727.5	1864.3	140.0	148.4	739.3	951.3
		3	114.8	128.4	775.6	906.3	164.4	316.2	1218.1	1457.3	89.5	94.1	593.4	694.4
		4	81.9	95.7	580.7	736.8	130.6	262.4	996.1	1397.0	78.3	76.4	711.6	651.5

Atom; e 8,2 vezes mais rápido que o ARM Cortex-A9. Embora a diferença entre o Core i7 e os processadores Atom e ARM esteja relacionada à diferença da frequência de operação, ela também se dá pela diferença de desempenho na execução das operações específicas da aplicação Conjunto de *Mandelbrot*: *sqrt, sqrtf* e recursividade. Conforme apresentado na Tabela 5.4, a execução delas é mais lenta no Atom e no ARM. Enquanto que no Intel Core i7 as funções *sqrt* e *sqrtf* são executadas em uma única instrução, no ARM e no Atom elas são substituídas por algoritmos que realizam mais instruções e gastam mais tempo para executar.

Nas aplicações WMEM-B, enquanto que a diferença para o Core2Quad aumentou para 1,8, no Atom e no ARM ela ficou em aproximadamente 7,3 e 7,4 vezes. Considerando as aplicações MEM-B, a diferença entre os processadores sofre atenuação para 1,2 no Core2Quad; 5,4 vezes para o Atom; e 4,8 vezes para o ARM. Esta atenuação na diferença de desempenho está relacionada diretamente ao sistema de memória que é particular a cada processador.

Tabela 5.4 - Diferença no Número de Ciclos Executados (*baseline* Core i7)

Operação	Core i7	Core2Quad	Atom	ARM Cortex-A9
SQRT	1.00	0.65	2.34	3.14
SQRTF	1.00	0.61	3.35	5.99
Recursividade	1.00	0.98	0.50	7.22

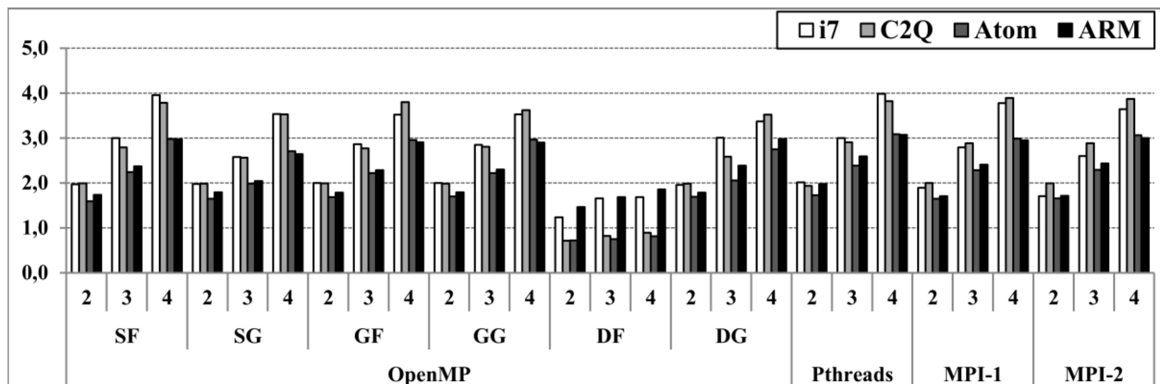
5.2.2 Análise das Interfaces de Programação Paralela

As Figura 5.4, 5.5 e 5.6 apresentam os resultados de *speedup* para a soma dos resultados de cada aplicação das classes CPU-B, WMEM-B e MEM-B, respectivamente, considerando todos os cenários simulados (IPP, processador e número de *threads*/processos). Na maioria dos casos, Pthreads obteve os melhores resultados de desempenho, visto que possibilita melhor exploração do paralelismo através de ajustes finos (melhor distribuição da carga de trabalho).

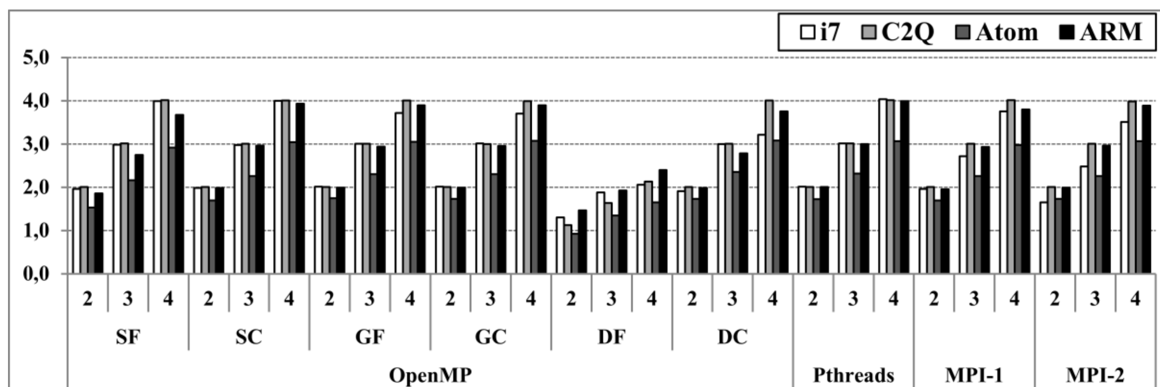
Nas aplicações CPU-B (Figura 5.4a), embora Pthreads tenha obtido os melhores resultados, o desempenho das IPPs foi similar, pois as aplicações ocupam intensivamente a CPU e possuem pouca troca de dados. Assim, as características de cada IPP com relação a comunicação e sincronização não tiveram influência no resultado. A exceção é com a política de escalonamento *dynamic* com granularidade fina do OpenMP. Nela, o *overhead* do escalonador para gerenciar a atribuição de pequenos *chunks* de iterações para as *threads* impactou de forma negativa no desempenho, principalmente para a aplicação Cálculo do Número Pi.

Como pode ser observado na Figura 5.4a, nos processadores Intel Core i7 e Core2Quad, o *speedup* esteve sempre próximo do ideal (*speedup* de 2, 3 e 4 para 2, 3 e 4 *threads*/processos respectivamente), o que não foi observado nos processadores Intel Atom e ARM. Neles, para 2 *threads*/processos, o *speedup* máximo obtido é próximo de 2 em Pthreads. No entanto, para 3 e 4 *threads*/processos, o desempenho foi abaixo do ideal. Conforme já descrito na seção anterior, o desempenho desta classe nesses processadores foi prejudicado pelo impacto da execução das operações de divisão sobre ponto flutuante de precisão simples, *sqrt, sqrtf* e recursividade (vide Tabela 5.4). Embora a versão sequencial também possua impacto na execução destas operações, nas versões paralelas o *overhead* para executar tais operações acaba impondo limites no ganho que o paralelismo proporciona.

Figura 5.4 - Resultado de *Speedup* nas Aplicações CPU-B



a) Com a aplicação Conjunto de *Mandelbrot*



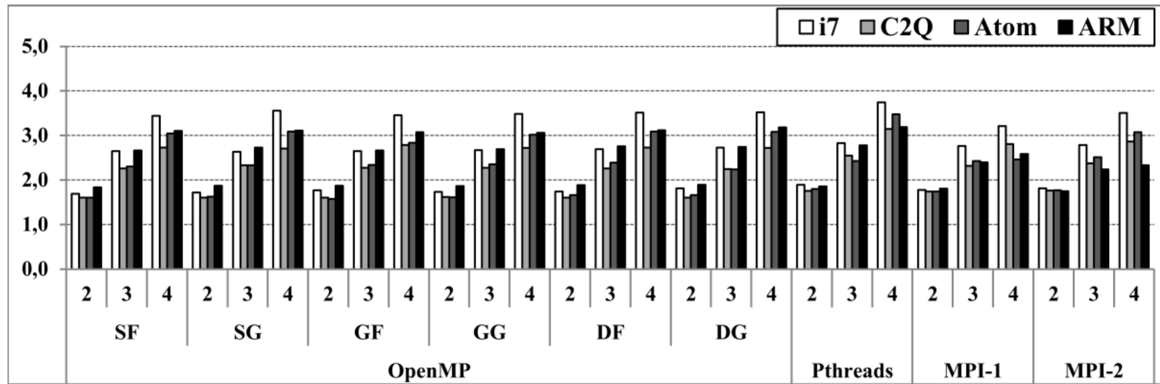
b) Sem a aplicação Conjunto de *Mandelbrot*

Portanto, para considerar mais justa a análise das IPPs com cenários similares em todos os processadores, a Figura 5.4b apresenta o *speedup* obtido sem o Conjunto de *Mandelbrot*. Nela, pode-se notar que o aumento na exploração do TLP seguiu similar nos processadores Intel Core i7, Core2Quad e ARM. Já o mesmo comportamento não ocorreu no processador Atom, o qual a exploração do TLP ficou sempre abaixo do ideal.

Nos resultados das aplicações WMEM-B mostrados na Figura 5.5, Pthreads continuou com os melhores resultados em ambos os processadores. Embora na exploração do paralelismo com 2 *threads*/processos as IPPs tenham tido desempenho similar, na exploração com 3 e 4 *threads*/processos a diferença entre elas aumentou. Isto porque enquanto OpenMP tem o *overhead* do *busy-waiting* e da divisão da carga de trabalho, MPI precisa trocar dados entre os processos através de operações *send/receive*. Assim, conforme aumenta o nível de exploração do paralelismo, o impacto de tais características passa a ter maior influência no desempenho.

O comportamento de OpenMP e MPI foi diferente nos processadores Intel e ARM. Nos processadores Intel, na exploração com 2 e 3 *threads*/processos, MPI obteve desempenho melhor. No entanto, para a exploração com 4 *threads*/processos, OpenMP foi um pouco

Figura 5.5 - Resultado de *Speedup* nas Aplicações WMEM-B



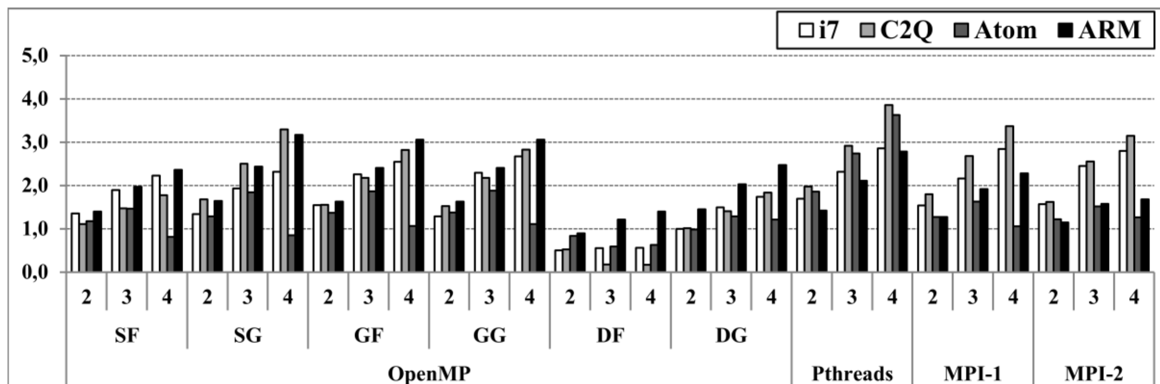
melhor. Isto significa que, quando o número de processos para comunicar é pequeno, a melhor distribuição da carga de trabalho em MPI compensa o *overhead* da comunicação, e é menor que o impacto do controle de distribuição da carga de trabalho (iterações) no OpenMP. No entanto, para 4 processos, o alto *overhead* das comunicações através de mensagens em MPI passou a ter maior representatividade que o gerenciamento da distribuição da carga de trabalho em OpenMP. Cabe destacar que o tamanho das mensagens é uniforme, independente do número de processos executando. Logo, o aumento do *overhead* está relacionado a quantidade de comunicações, e não a quantidade de dados transmitidos em cada comunicação. Já no processador ARM, OpenMP foi melhor que MPI em todos os casos, o que mostra que nestes processadores, o uso de diretivas *send/receive* possui maior impacto que a distribuição da carga de trabalho em OpenMP.

Comparando as implementações MPI, a criação dinâmica de processos possibilitou melhores ganhos com o MPI-2 nos processadores Intel: MPI-2 foi aproximadamente 2% mais eficiente que MPI-1 para todos os números de processos. Já no ARM, MPI-2 foi mais lento que MPI-1 para todas as quantidades de processos. Conforme discutido na Seção 5.1 quanto ao número de instruções executadas, o impacto das comunicações através de intercomunicadores no MPI-2 (pai-filho) foi maior no ARM que nos processadores Intel, o que acaba contribuindo para o baixo desempenho do MPI-2 nos processadores ARM.

Nos resultados das aplicações MEM-B mostrados na Figura 5.6, pode-se notar que o desempenho das IPPs passa a ser diferente nos processadores alvo. Isto porque as aplicações possuem comunicação intensa e cada processador possui organização hierárquica de memória diferente. Por exemplo, OpenMP com a política de escalonamento *static* com granularidade grossa obteve desempenho superior no processador ARM que no processador Intel Core i7.

Pthreads manteve o melhor desempenho nos processadores Intel, enquanto que OpenMP obteve os melhores resultados no processador ARM. Diante da característica de

Figura 5.6 - Resultado de *Speedup* nas Aplicações MEM-B



comunicação intensiva e da quantidade de pontos de sincronização existente nas aplicações desta classe, isto significa que o uso de *mutex* em Pthreads (para permitir a comunicação entre as *threads*) apresentou maior impacto no desempenho no processador ARM do que no Intel. Em Pthreads, quando uma *thread* chega a um ponto de sincronização (*mutex, barrier*, etc.), ela entra em estado de espera e acaba perdendo o processador até que receba um sinal informando que ela pode voltar a execução. Recebido este sinal, ela será escalonada para algum *core* e seguirá sua execução. No entanto, conforme Akkan et al. (2013) o desempenho deste mecanismo pode ser inferior ao utilizado por OpenMP e dependente do sistema operacional e arquitetura utilizada.

As políticas de escalonamento do OpenMP possuíram comportamento diferente. Conforme discutido na Seção 5.1, esta diferença é devido ao desbalanceamento de carga causado pela forma como as iterações são distribuídas entre as *threads*. Assim, o melhor resultado ocorreu com as políticas *static* com granularidade grossa e *guided* com granularidade fina e grossa em ambos processadores.

Na comparação envolvendo MPI-1 e MPI-2, no processador Core2Quad, MPI-2 obteve melhores desempenhos para 2 e 3 processos. No entanto, para os demais processadores, o impacto da criação dinâmica de processos e o consequente impacto do relacionamento hierárquico das comunicações entre pai e filho no MPI-2 adicionaram *overhead* nas implementações MPI-1.

Os resultados apresentados nesta seção mostram que as Interfaces de Programação Paralela possuem desempenho diferente quando considerados processadores de propósito geral e embarcados. Nos processadores de propósito geral (Intel Core i7 e Core2Quad), Pthreads obteve os melhores resultados nas três classes, com desempenho muito próximo do ideal. Já nos processadores para sistemas embarcados, Pthreads obteve melhor desempenho no Atom para as três classes. No entanto, no ARM, conforme o número de comunicações e

sincronização aumenta, nota-se que o impacto no desempenho do uso de operações de sincronização em Pthreads (*mutex, barrier*, etc.) foi maior que o impacto da distribuição da carga de trabalho e do *busy-waiting* em OpenMP. Assim, para as aplicações com uso intensivo da CPU e pouca comunicação Pthreads obteve melhor desempenho enquanto que OpenMP foi melhor nas aplicações com bastante comunicação e pontos de sincronização.

5.3 Consumo de Energia

Os resultados apresentados na Tabela 5.5 correspondem ao total de energia consumida para executar todas as aplicações de cada conjunto de *benchmarks* considerando as diferentes Interfaces de Programação Paralela e processadores.

5.3.1 Análise das Arquiteturas

No geral, o processador ARM consumiu menos energia para executar as três classes de *benchmarks,* economizando aproximadamente 14% de energia com relação ao Atom; 50% com relação ao Core i7; e 54% comparado ao Core2Quad. No entanto, é importante notar que, conforme discutido nas Seções 5.1 e 5.2, o comportamento no ARM das operações *sqrtf*, *sqrt*, *mod* e recursividade presentes nas aplicações da classe CPU-B possam ter contribuído para diminuir esta diferença. Já que desconsiderando as aplicações CPU-B, a diferença na economia de energia aumenta para 32% com relação ao processador Atom; 62% ao Core i7; e 66% ao Core2Quad.

Pode-se notar na Tabela 5.5 que, conforme o sistema de memória passa a ter maior influência nos resultados (aplicações WMEM-B e MEM-B), a economia de energia dos processadores para sistemas embarcados aumenta com relação aos processadores de propósito geral: ARM economizou 59% e 65%, e Atom economizou 40% e 48% nas aplicações WMEM-B e MEM-B com relação ao Core i7. O aumento desta diferença no consumo de energia entre as classes pode ser atribuído ao sistema de memória, que é mais estressado nas aplicações MEM-B e, conforme apresentado na Seção 4.3, a memória principal (RAM) do Atom e do ARM é menor e possui menor consumo de energia por acesso que o Core i7 e o Core2Quad.

É importante ressaltar que o acesso à memória RAM não impacta somente na energia consumida pela quantidade de acessos a memória. Cada acesso a este nível de memória

implica em maior tempo de espera por parte do processador para que o dado esteja disponível para computar, e este tempo é diferente nos quatro processadores. Por exemplo, um acesso à memória RAM no ARM corresponde ao tempo de acesso a *cache* L1 + L2 + memória RAM, enquanto que no Core i7, faz-se necessário acessar o terceiro nível de *cache*, impondo maior tempo de espera do processador.

Outro fator que também impacta no consumo de energia da aplicação é o desempenho da mesma, que influencia no consumo estático dos componentes (processador, memória, etc.). O consumo estático ocorre enquanto o componente estiver ativo/ligado, portanto, quanto maior o tempo de execução de uma aplicação, maior será a energia estática consumida.

É possível observar ainda, na Tabela 5.5 que para as aplicações CPU-B e WMEM-B, em sua maioria, as versões paralelas consumiram menos energia estática que a versão sequencial. Isto é possível pela redução da energia estática consumida pela memória. Por exemplo, considerando o consumo de energia estática da versão sequencial em 100 joules, distribuídos em 40 joules do processador e 60 joules da memória. Na versão paralela com 2 *threads*/processos, enquanto que o consumo estático do processador será semelhante ao da versão sequencial, pois consiste da soma do consumo estático de cada *core*, o consumo de

Tabela 5.5 - Consumo Total de Energia (em Joules)

			CPU-B				WMEM-B				MEM-B			
			i7	**C2Q**	**Atom**	**ARM**	**i7**	**C2Q**	**Atom**	**ARM**	**i7**	**C2Q**	**Atom**	**ARM**
Sequencial			1324.9	1212.4	803.4	1278.5	2456.9	2604.3	1556.3	1118.2	1478.9	1382.0	720.1	586.8
OMP	**SF**	**2**	1273.5	1201.0	828.2	1217.0	2578.8	2868.2	1555.2	1021.8	1946.6	1969.5	949.1	686.8
		3	1234.1	1201.6	790.0	1189.2	2481.3	2883.0	1485.7	968.5	1979.3	2096.5	1028.2	695.6
		4	1220.1	1190.0	767.6	1179.1	2454.4	2976.5	1468.1	989.3	2035.2	2201.8	1477.9	711.1
	SG	**2**	1223.5	1153.9	750.2	1163.5	2560.9	2859.5	1534.5	993.2	1721.4	1556.3	841.5	598.6
		3	1196.4	1142.8	727.4	1170.5	2476.7	2837.4	1466.2	936.5	1700.7	1541.5	808.0	581.8
		4	1175.3	1128.6	700.9	1122.8	2430.8	2982.2	1444.7	962.9	1722.7	1536.6	1239.9	574.8
	GF	**2**	1221.1	1153.1	741.5	1164.4	2538.4	2863.0	1549.7	994.4	1657.4	1596.3	814.7	607.9
		3	1188.6	1142.4	712.9	1144.5	2469.5	2870.5	1466.5	945.0	1641.3	1624.4	829.6	582.6
		4	1209.0	1129.5	694.4	1132.6	2443.6	2940.2	1487.4	965.9	1688.4	1619.5	1121.4	593.1
	GG	**2**	1221.0	1153.3	736.2	1162.0	2557.0	2848.8	1540.3	996.4	1740.2	1603.9	813.2	607.6
		3	1188.6	1140.0	712.6	1144.1	2468.7	2869.9	1462.8	943.7	1625.8	1624.7	827.1	582.6
		4	1208.0	1143.6	693.7	1134.4	2442.8	2967.4	1448.1	965.6	1665.6	1619.3	1132.4	596.6
	DF	**2**	1706.1	2185.2	1669.5	1387.0	2561.0	2867.9	1517.9	995.7	3487.4	3142.5	1257.3	968.6
		3	1725.9	2517.5	2173.7	1631.3	2475.2	2882.7	1451.8	936.0	3848.0	7076.8	1842.6	960.2
		4	1907.2	2832.3	2567.2	1859.7	2440.8	2973.4	1423.0	959.1	4255.1	8884.7	1986.6	1021.3
	DG	**2**	1228.5	1154.9	738.2	1166.1	2530.4	2865.1	1519.2	985.6	2103.4	2020.5	1025.0	690.3
		3	1185.0	1142.9	716.2	1144.9	2465.2	2892.6	1496.7	938.2	2044.0	2048.7	1084.2	667.6
		4	1226.4	1130.8	692.7	1134.2	2438.8	2978.7	1424.4	950.7	2096.0	2075.2	1197.8	687.5
Pthreads		**2**	1212.9	1161.1	730.7	1081.2	2329.4	2580.2	1383.9	1025.9	1494.8	1382.7	683.3	667.8
		3	1177.6	1131.5	701.6	1099.9	2267.3	2556.4	1328.2	955.1	1489.0	1370.9	657.0	644.5
		4	1162.7	1131.5	691.2	1130.2	2217.6	2612.2	1273.0	973.0	1492.0	1363.8	647.1	637.4
MPI-1		**2**	1238.9	1146.6	751.3	1138.6	2436.8	2651.0	1449.8	1020.4	1836.5	1476.7	880.6	715.6
		3	1205.2	1135.3	736.6	1099.6	2557.6	2804.0	1445.3	1010.8	1864.6	1474.7	866.2	675.9
		4	1199.0	1127.0	720.7	1107.5	2678.0	2899.6	1593.5	1088.2	1856.0	1502.6	1272.8	707.8
MPI-2		**2**	1327.5	1147.4	749.2	1210.4	2387.9	2553.8	1419.0	1067.2	1634.0	1526.0	835.1	795.5
		3	1236.0	1140.0	722.9	1155.5	2372.7	2670.9	1386.0	1099.7	1664.8	1534.2	865.7	798.7
		4	1224.5	1128.3	694.4	1140.3	2351.1	2753.6	1425.5	1327.5	1765.2	1595.0	1156.8	877.1

memória é relacionado ao tempo de execução da versão paralela, pois só existe uma memória. Portanto, neste caso, a versão paralela com 2 *threads*/processos consome 70 joules (40 do processador e 30 da memória).

Neste trabalho, conforme dados obtidos de simulações dos processadores no McPAT (LI, et al. 2013), considera-se que a energia estática representa 20% do total do processador. No entanto, diferentes trabalhos utilizam valores que variam entre 10% e 30% (HOROWITZ, 2007) (BORKAR et al., 2011) (ESMAEILZADEH, 2012). Não existe um consenso definido do quanto da energia consumida pelos processadores é estática, pois ela depende de vários fatores, como a tecnologia de fabricação utilizada (32 nn, 45 nn, etc.).

5.3.2 Análise das Interfaces de Programação Paralela

Como pode ser observado na Tabela 5.5, Pthreads foi melhor, ou seja, consumiu menos energia nos processadores Intel para a execução de todos os *benchmarks*. Já no processador ARM, Pthreads consumiu menos energia nas aplicações CPU-B, enquanto que OpenMP foi melhor nas aplicações WMEM-B e MEM-B.

Nas aplicações CPU-B, embora Pthreads tenha obtido menor consumo de energia, a energia consumida pelas IPPs na maioria dos casos, é similar em todos os processadores. Isto porque elas possuem resultados similares no desempenho, número de instruções executadas e acessos à memória de dados. Cabe ressaltar que a política de escalonamento *dynamic* obteve o pior (mais alto) consumo de energia devido ao comportamento dela, conforme discutido nas Seções 5.1 e 5.2. Pelo fato das aplicações possuírem uso intensivo da CPU e carga de trabalho balanceada e equivalente, a criação dinâmica de processos em MPI-2 teve acréscimo de energia de aproximadamente 2% sobre o MPI-1.

Para as aplicações WMEM-B, pode-se notar que houve diferença no consumo de energia entre as IPPs. Pthreads continuou sendo a IPP que menos consumiu energia nos processadores Intel, pois possuiu menor número de instruções executadas, acessos à memória de dados e melhor desempenho (vide Seção 5.1.2 e 5.2). Isto não se repetiu no ARM, onde o OpenMP foi melhor na maioria das políticas de escalonamento. Embora Pthreads tenha obtido desempenho pouco melhor que OpenMP nesta classe, ele executou mais instruções decorrente do gerenciamento das variáveis globais, o qual foi maior que o impacto do *busy-waiting* em OpenMP, conforme discutido na Seção 5.1.2.

Considerando os resultados das implementações MPI nas aplicações WMEM-B, MPI-2 consumiu, em média 7% menos energia que MPI-1 nos processadores Intel. Isto porque, criando processos em tempo de execução reduz o consumo de energia estática da aplicação. Por exemplo, considerando que a aplicação MPI-1 levou 100 segundos para executar com quatro processos, o consumo estático será a energia estática da memória e processador durante 100 segundos para os quatro processos, pois todos os processos iniciam e finalizam a execução ao mesmo tempo. Isto não ocorre com MPI-2, pois um único processo executará os 100 segundos, enquanto que os outros três executarão somente quando houver carga de trabalho útil para computar, que consequentemente será menor que 100 segundos, reduzindo assim o consumo estático do processador e memória. O comportamento anterior não se repete no processador ARM, em que as comunicações através de intercomunicadores no MPI-2 possuíram impacto maior no desempenho que o ganho obtido com a criação dinâmica. Sendo assim, MPI-2 consumiu 10% mais energia que MPI-1 neste processador.

Nas aplicações MEM-B a diferença no consumo de energia entre as IPPs aumentou, visto que existe maior número de comunicações e pontos de sincronização. Nos processadores Intel, Pthreads continuou sendo a IPP que consumiu menos energia e a diferença no consumo varia conforme o processador utilizado (Core i7, Core2Quad e Atom). Já no processador ARM, OpenMP, com as políticas de escalonamento SG, GF e GG, foi a IPP que consumiu menos energia, pois embora tiveram número de acessos a memória de dados similar a Pthreads, possuíram os melhores resultados de desempenho e número de instruções executadas.

Considerando as diferentes políticas de escalonamento do OpenMP, pode-se notar que a simples alteração da política e da granularidade pode resultar em grande impacto no consumo de energia. Por exemplo, enquanto que a política *static* com granularidade grossa (SG, melhor caso do OpenMP) consome aproximadamente 1721 joules na execução com 2 *threads* no processador Core i7, a mesma política, porém com granularidade fina (SF), consome 1946 joules no mesmo cenário. Isto representa uma economia de 13% em apenas utilizar a política de escalonamento adequada para tal nicho de aplicação.

Comparando os resultados do MPI para as aplicações MEM-B (Tabela 5.5), o fato de criar processos dinamicamente fez com que MPI-2 economizasse energia com relação ao MPI-1 na maioria dos casos, nos processadores Intel. Isto porque, conforme discutido anteriormente (análise das aplicações WMEM-B) criar processos somente quando eles são úteis faz com que diminua o consumo estático da aplicação. No processador ARM, MPI-1 consumiu menos energia que o MPI-2 e a diferença é maior que para as aplicações WMEM-

Tabela 5.6 - EDP Total (em Joules *x* segundos) – em milhares

			CPU-B				WMEM-B				MEM-B			
			i7	**C2Q**	**Atom**	**ARM**	**i7**	**C2Q**	**Atom**	**ARM**	**i7**	**C2Q**	**Atom**	**ARM**
Sequencial			148.9	177.6	508.6	1012.3	331.1	630.4	1335.8	1032.1	109.4	109.2	217.7	218.9
OMP	**SF**	**2**	71.1	87.0	327.4	545.6	189.3	414.9	812.9	490.7	110.1	166.3	270.5	200.0
		3	45.1	60.3	218.2	394.9	118.3	302.9	533.5	321.1	78.5	150.6	251.3	148.8
		4	33.8	44.2	158.8	317.8	90.2	274.9	391.2	296.9	65.5	134.0	888.1	131.2
	SG	**2**	69.1	85.1	306.2	522.8	185.8	413.7	780.4	471.4	92.6	72.4	187.0	133.1
		3	50.8	62.3	230.6	483.9	118.8	282.0	518.8	299.6	63.3	48.0	123.9	86.4
		4	36.5	45.2	162.1	357.1	85.4	275.0	378.1	285.4	51.9	36.1	639.1	64.5
	GF	**2**	68.0	84.8	290.7	524.0	181.9	414.1	813.1	468.1	80.3	83.0	173.5	135.6
		3	45.9	58.8	201.2	411.5	116.7	301.0	518.0	307.1	53.8	61.8	126.4	87.4
		4	39.4	42.7	147.2	322.6	88.8	260.0	412.7	288.1	47.0	47.6	402.9	70.7
	GG	**2**	67.9	84.9	285.9	523.2	186.6	410.7	795.5	464.4	105.2	86.1	172.8	135.8
		3	46.0	58.2	201.0	409.4	115.3	302.4	514.4	302.1	52.6	61.8	127.8	87.4
		4	39.3	44.6	146.8	324.7	87.9	275.4	390.4	291.1	44.5	47.3	418.0	71.0
	DF	**2**	152.7	543.7	2050.4	695.2	185.9	414.2	748.0	461.9	965.4	858.5	665.8	593.2
		3	119.3	617.1	3153.3	741.3	114.8	303.3	495.1	298.4	1044.0	7980.3	1662.4	451.5
		4	138.7	686.2	3681.5	826.4	88.3	275.9	373.0	294.0	1157.8	10927.3	1884.7	437.8
	DG	**2**	69.0	85.1	289.1	525.8	175.2	415.7	751.5	455.9	202.0	194.6	420.7	191.1
		3	43.7	62.0	216.0	387.0	112.5	307.4	522.1	300.1	125.4	142.2	306.9	134.4
		4	40.1	45.3	157.1	312.9	87.7	278.0	374.7	278.8	110.2	113.4	411.2	116.9
Pthreads		**2**	66.8	86.2	279.1	433.2	160.8	373.9	654.2	494.9	66.6	56.4	116.4	189.4
		3	43.1	56.1	186.3	341.7	104.5	248.6	450.8	304.5	50.0	37.7	75.2	120.8
		4	32.1	42.3	140.2	304.9	77.3	214.2	312.8	281.9	40.5	28.3	55.3	88.8
MPI-1		**2**	71.9	83.7	300.2	536.7	169.8	363,8	685.4	485.5	80.3	61.0	166.8	208.2
		3	46.7	56.6	206.8	368.0	110.1	275.2	478.8	344.5	55.2	39.9	124.4	127.4
		4	34.9	41.7	153.5	307.3	98.0	241.8	492.5	343.1	39.9	31.2	317.3	107.5
MPI-2		**2**	80.0	84.0	294.7	577.3	171.2	361.6	688.0	508.3	70.6	75.1	171.8	234.3
		3	49.5	56.6	197.0	389.7	108.7	270.1	454.7	387.0	44.9	43.1	135.2	161.6
		4	35.7	41.8	142.0	315.8	84.8	237.9	359.3	457.5	38.9	34.5	245.2	160.4

B. Isto pois, esta classe possui maior número de comunicações entre os processos; e conforme discutido previamente nas Seções 5.1.3 e 5.2, as comunicações através de intercomunicadores possuíram maior impacto no número de instruções executadas e desempenho que o ganho com a criação dinâmica.

5.4 *Energy-Delay Product*

Os resultados apresentados na Tabela 5.6 correspondem ao total de EDP para cada grupo de *benchmark*, enquanto que a Figura 5.7 explora a eficiência dos recursos considerando EDP.

5.4.1 Análise das Arquiteturas

Comparando os quatro processadores na Tabela 5.6, em todos os casos o EDP do Core i7 foi menor (melhor). Considerando as aplicações CPU-B, foi 1,6 vezes menor que Core2Quad; 6,2 vezes que Atom; e 7,9 vezes que ARM. Embora exista uma grande diferença

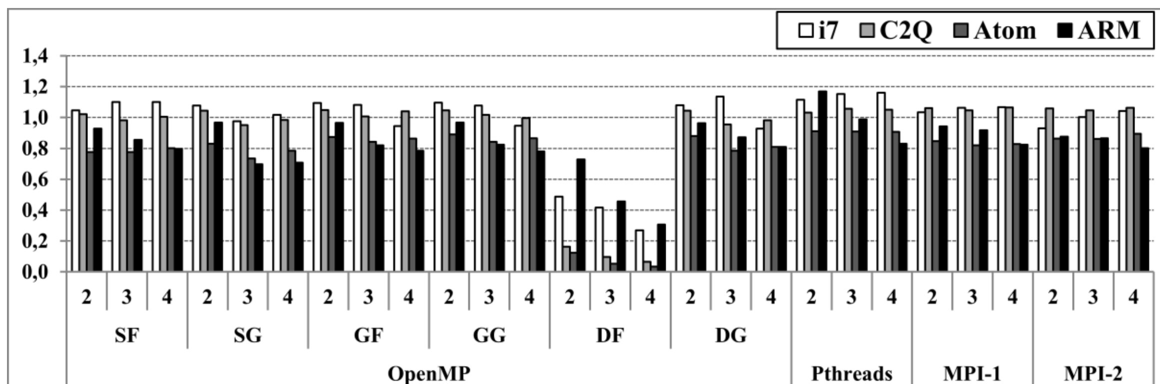
no desempenho entre Core i7 e os processadores Atom e ARM, essa diferença é atribuída principalmente a energia consumida e ao desempenho para executar as operações/instruções das aplicações desta classe (conforme discutido nas Seções 5.1, 5.2 e 5.3).

Pode-se notar, na Tabela 5.6, que conforme o sistema de memória passa a ser mais estressado pelas aplicações (WMEM-B e MEM-B), a diferença de EDP entre os processadores de propósito geral e para embarcados diminui. Isto porque conforme apresentado na Seção 5.2.1, a frequência do processador passa a ser menos importante que o sistema de memória, e a economia de energia nos sistemas embarcados aumenta.

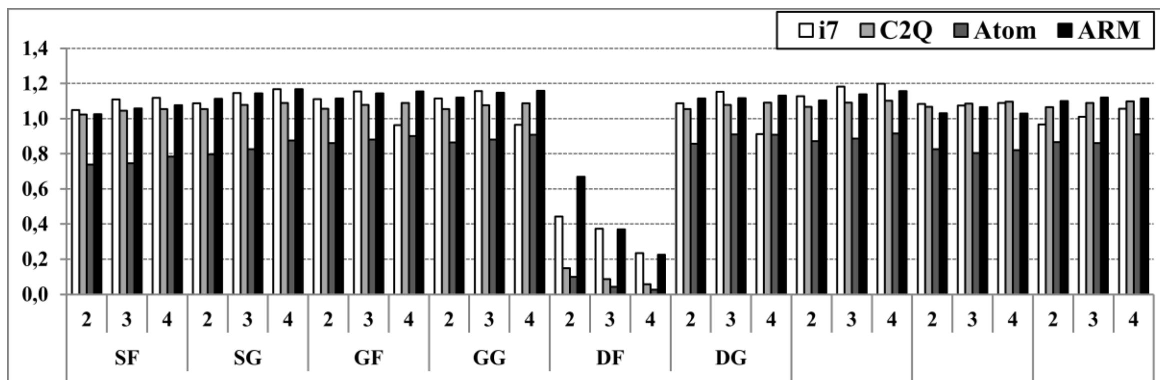
5.4.2 Análise das Interfaces de Programação Paralela

Quando se compara a eficiência do EDP de cada Interface de Programação Paralela nas Figura 5.7, 5.8 e 5.9, pode-se notar que os *benchmarks* que apresentam a melhor eficiência são os CPU-*Bound* (Figura 5.7a): eles têm melhor desempenho e baixo consumo de energia devido ao alto grau de paralelismo (sem dependência de dados, pouca comunicação entre as *threads*/processos e barreiras, impactando em menor número de acessos à memória

Figura 5.7 - Resultados de Eficiência do EDP – Aplicações CPU-B

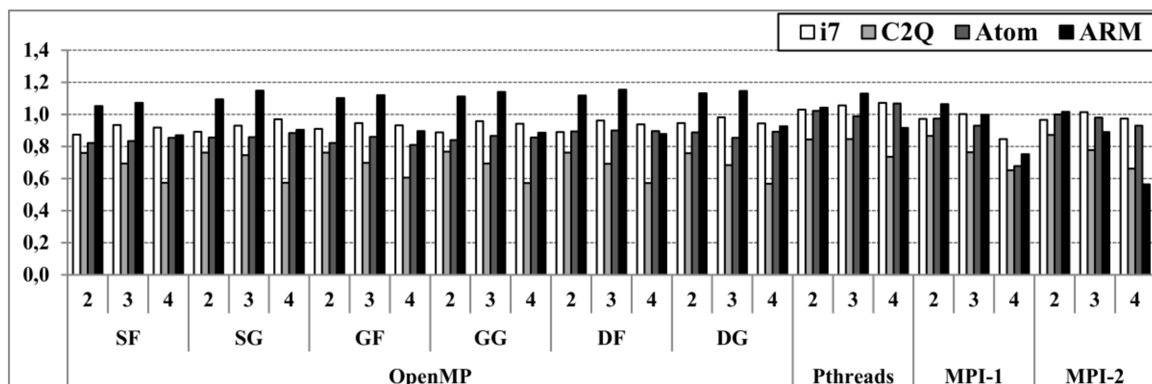


a) Com Conjunto de *Mandelbrot*



b) Sem Conjunto de *Mandelbrot*

Figura 5.8 - Resultados de Eficiência do EDP – Aplicações WMEM-B



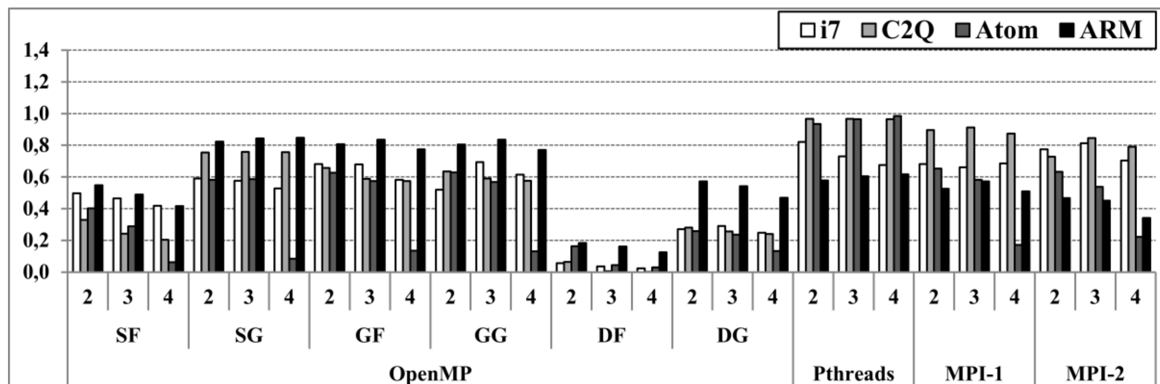
principal). Nesta classe, embora Pthreads tenha obtido os melhores resultados para ambos processadores, o comportamento das IPPs foi bastante similar. Isto é possível, pois além de terem obtido desempenho próximo do ideal, a paralelização nesta classe possibilitou economizar energia estática da memória com relação à versão sequencial (vide discussão na Seção 5.3.1). A única exceção é para o OpenMP com a política de distribuição de iterações *dynamic* com granularidade fina (DF). Conforme discutido previamente, este escalonador possui pior desempenho e maior consumo de energia, devido ao *overhead* para controlar a distribuição da carga de trabalho entre as *threads*.

Novamente, se for desconsiderado o impacto do uso das operações *sqrt*, *sqrtf* e recursividade na aplicação Conjunto de *Mandelbrot* (Figura 5.7b), a eficiência das IPPs no ARM se equipara a eficiência de EDP obtida nos processadores de propósito geral. O mesmo não acontece com o Atom, pois embora tenha consumo reduzido de energia, o ganho de desempenho ficou abaixo do ideal.

Nas aplicações WMEM-B (Figura 5.8), Pthreads continuou com os melhores resultados nos processadores Intel. No entanto, para o processador ARM, OpenMP obteve melhor eficiência do EDP. No MPI, conforme ocorre o aumento no número de processos, pode-se notar perda de eficiência do EDP em ambos os processadores devido ao baixo desempenho e aumento do consumo de energia resultante da comunicação. No entanto, MPI-2 obteve melhores resultados que MPI-1 nos processadores Intel devido a criação dinâmica de processos. O mesmo não ocorre para o ARM, em que o relacionamento hierárquico entre pai e filho (comunicações entre intercomunicadores) impactou de forma negativa no desempenho e no número de instruções executadas.

Por fim, nas aplicações MEM-B (Figura 5.9), pode-se notar que a eficiência do EDP das IPPs é muito menor que nas outras duas classes; e que as IPPs possuem maior diferença entre elas. Nos processadores Intel, Pthreads obteve o melhor nível de EDP para os

Figura 5.9 - Resultados de Eficiência do EDP – Aplicações MEM-B



processadores Core2Quad e Atom para todos os montantes de *threads*. Já para o processador Core i7, MPI-2 obteve o melhor EDP, o que mostra que nesse processador a criação dinâmica de processos e a comunicação através de primitivas *send*/*receive* foi mais eficiente que a comunicação através de variáveis compartilhadas e que o uso de *mutex* para controlar a comunicação/sincronização entre as *threads* Pthreads. OpenMP obteve novamente os melhores resultados no processador ARM. Isto reforça que, embora a técnica de *busy-waiting* empregada pelas *threads* OpenMP possua impacto na quantidade de acessos a memória, ele é inferior ao impacto no desempenho causado pelas operações de sincronização (*mutex*, *barrier*, etc.) em Pthreads.

5.5 Escalabilidade

Esta seção discute a escalabilidade das Interfaces de Programação Paralela em termos de desempenho, consumo de energia e *Energy-Delay Product*. Neste trabalho, a escalabilidade consiste em avaliar o comportamento das Interfaces de Programação Paralela e das arquiteturas alvo com relação à eficiência dos recursos usados pela aplicação conforme aumenta o grau de exploração do paralelismo. Para tal, está sendo considerado que a versão sequencial possui eficiência dos recursos de 100% em um único *core*. Sabe-se que a perda de eficiência dos recursos é normal quando ocorre o aumento do grau de exploração paralelismo, no entanto, quanto menor for esta perda, melhor é, porque significa que os recursos computacionais disponíveis estão sendo mais bem utilizados.

Para tornar a comparação justa entre as IPPs nos diferentes processadores, será desconsiderado o programa Conjunto de *Mandelbrot* da classe CPU-Bound. Conforme discutido nas Seções 5.1, 5.2 e 5.3, as instruções executadas por esta aplicação tiveram desempenho e consumo de energia pior no ARM devido a limitações do *hardware*. Ademais,

os resultados apresentados da IPP OpenMP correspondem ao melhor resultado obtido dentre todas as políticas utilizadas em cada classe de *benchmark* para cada processador. Por exemplo, o melhor desempenho do OpenMP na execução da classe CPU-Bound no processador Core2Quad foi com a política de escalonamento *static* com granularidade fina (SF). Portanto, estes resultados serão utilizados na análise de desempenho, e o mesmo ocorre para todos os outros cenários (processador, classe de *benchmark*, e eixo avaliado – energia e *Energy-Delay Product*).

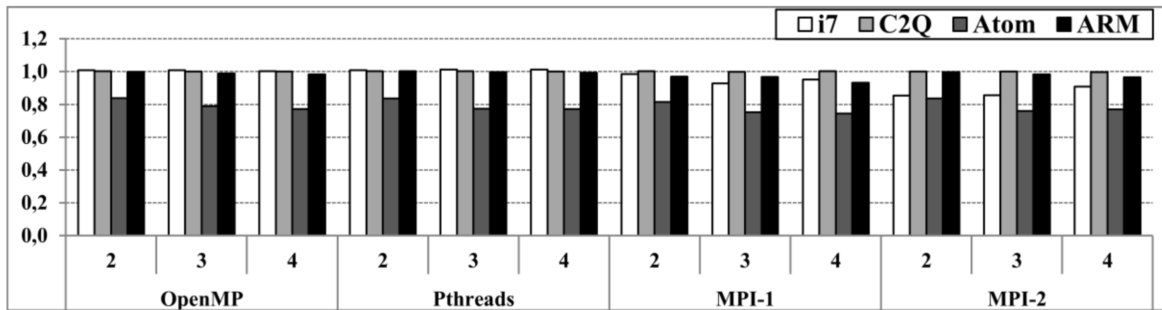
5.5.1 Desempenho

A Figura 5.10 apresenta os resultados de eficiência do desempenho considerando os melhores resultados obtidos com as IPPs em cada um dos processadores. Nos processadores Intel, a IPP que obteve os melhores resultados de escalabilidade e eficiência para todos os conjuntos de *benchmark* foi Pthreads. Já para o processador ARM, Pthreads foi melhor nas aplicações CPU-B, enquanto que nas aplicações WMEM-B e MEM-B, OpenMP foi melhor. Conforme pode ser acompanhado na Figura 5.10, a classe que possibilitou os melhores ganhos em termos de eficiência e escalabilidade foi a CPU-B (Figura 5.10a). Na maioria dos casos, o aumento no grau de exploração do paralelismo de 2 para 4 *threads/*processos não impactou em perda da eficiência.

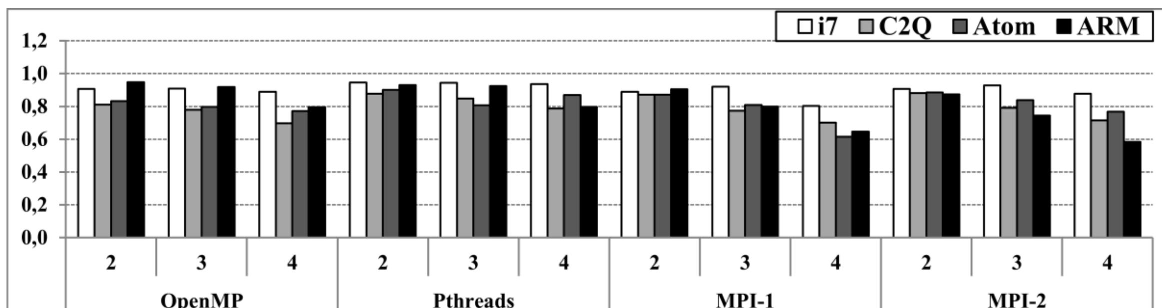
Nas aplicações WMEM-B (Figura 5.10b), o comportamento entre as IPPs é diferente e, conforme ocorre o aumento no grau do paralelismo, ocorre também perda na eficiência. Considerando o OpenMP, o melhor resultado desta IPP com 2 e 3 *threads* ocorreu no processador ARM. No entanto, aumentando o grau de exploração do TLP para 4 *threads*, o ARM perdeu 12% de eficiência comparado a apenas 2% no Core i7. Em Pthreads, a exploração do TLP com 2 e 3 *threads* foi similar nos processadores ARM e Core i7. Porém, quando o grau de exploração do TLP aumentou para 4, enquanto o Core i7 perdeu apenas 2% de eficiência, o ARM perdeu 6% de eficiência. Esta perda de desempenho maior no ARM está relacionada ao impacto da sincronização que, conforme discutido na Seção 5.2, foi maior neste processador.

Ainda nas aplicações WMEM-B (Figura 5.10b), na exploração com MPI, pode-se notar que, com o aumento no número de processos de 2 para 4, a eficiência decresce em todos os processadores. Isto porque, embora o aumento no número de processos permita melhorar o desempenho, ele é limitado até certo ponto pelo consequente aumento no número de

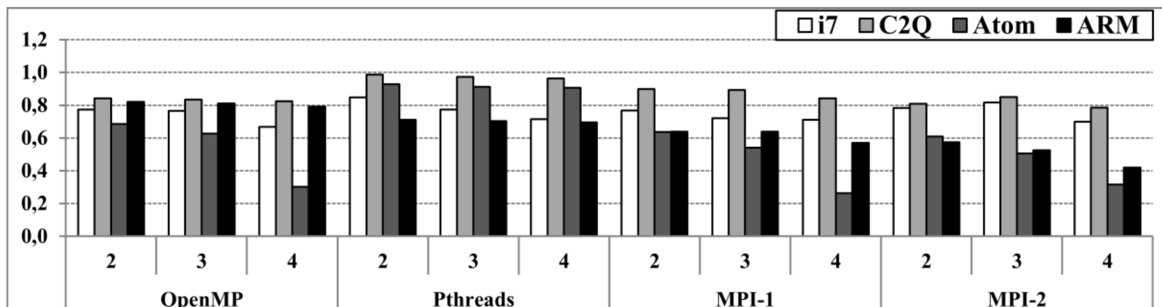
Figura 5.10 - Resultados de Eficiência/Escalabilidade do Desempenho



a) Aplicações CPU-B



b) Aplicações WMEM-B



c) Aplicações MEM-B

operações para troca de mensagens. Ademais, nos processadores Intel, o uso da criação dinâmica de processos no MPI-2 permitiu melhores ganhos de eficiência e escalabilidade comparando as implementações OpenMP e MPI-1.

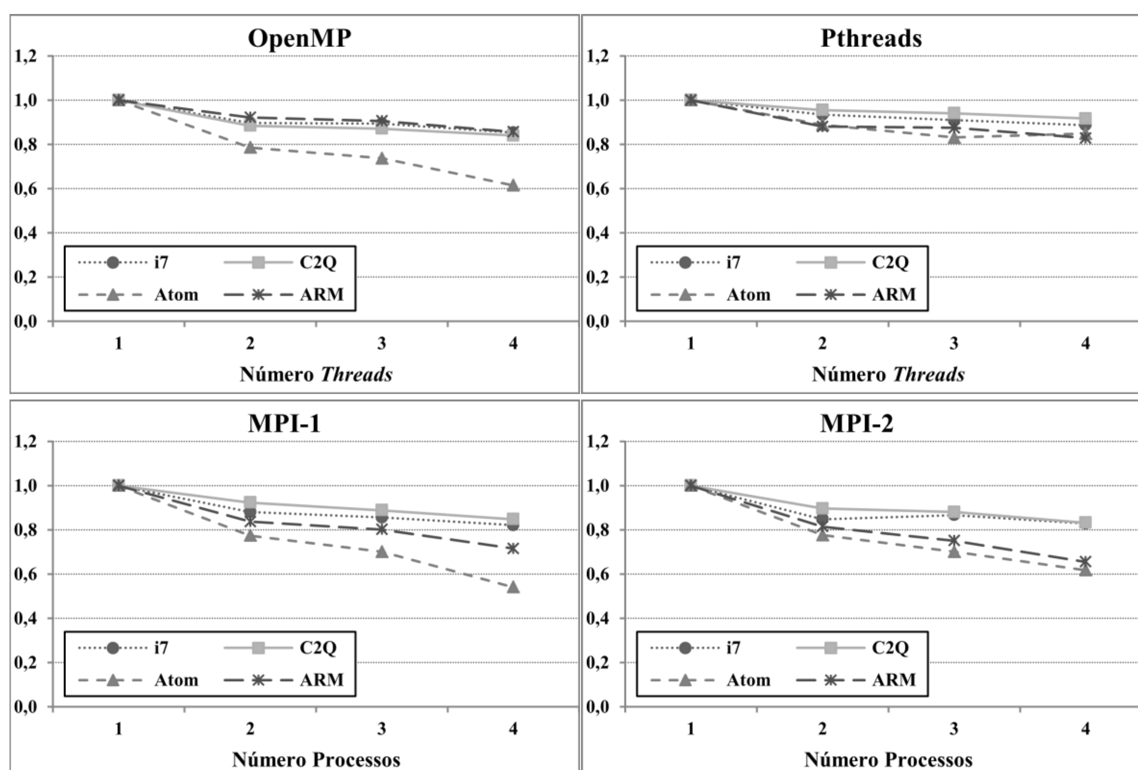
Considerando as aplicações MEM-B (Figura 5.10c), os melhores resultados ocorreram no processador Core2Quad com a IPP Pthreads. Enquanto que no ARM, o OpenMP perdeu apenas 3% de eficiência no aumento do grau de TLP de 2 para 4, no Core i7 ele perdeu 10%, e no Atom aproximadamente 39%. O mesmo ocorre para Pthreads no ARM, que embora não tenha obtido os melhores resultados, perdeu apenas 1% de eficiência de 2 para 4 *threads*, comparado ao Core i7 que perdeu 14%, chegando a resultados similares na exploração do paralelismo com 4 *threads*. Nos processadores Atom, Pthreads obteve melhores resultados, com perda de apenas 2% na eficiência de 2 para 4 *threads*, o que representa a perda de 20

vezes menos que o OpenMP; 19 vezes menos que MPI-1; e 15 vezes menos que MPI-2 no mesmo processador.

A Figura 5.11 apresenta os resultados médios de eficiência de cada IPP considerando a execução de todas as classes de *benchmark*. Comparando os resultados do OpenMP nos quatro processadores alvo, a exploração do paralelismo no ARM obteve menor perda de eficiência. Na execução com 2 *threads*, o ARM obteve eficiência de 2% maior que Core i7 e 4% maior que Core2Quad. Embora na exploração do paralelismo com 4 *threads*, a diferença tenha diminuído, ARM possibilitou melhores resultados de eficiência na exploração do paralelismo com OpenMP. Nos resultados com a IPP Pthreads, pode-se observar que a melhor eficiência foi obtida com o processador Core2Quad e que com o aumento do número de *threads* de 2 para 4, todos os processadores perderam aproximadamente 4% de eficiência. Isto mostra que embora o Atom e ARM não alcancem os melhores resultados de eficiência, eles escalam de forma similar aos processadores de propósito geral na execução com Pthreads.

Ainda na Figura 5.11, os melhores resultados de exploração do paralelismo com MPI-1 e MPI-2 foram obtidos no processador Core2Quad. No MPI-1, enquanto que a escalabilidade de 2 para 4 processos foi similar entre Core2Quad e Core i7, nos processadores ARM e Atom, o aumento de 2 para 4 processos impactou em perda na eficiência de 12% no ARM e 23% no Atom. No MPI-2, embora Core2Quad tenha obtido melhor eficiência, ele perdeu 7% de

Figura 5.11 - Resultado Médio de Eficiência/Escalabilidade do Desempenho por IPP



eficiência de 2 para 4 processos enquanto que Core i7 perdeu apenas 2%. Novamente, Atom e ARM perderam mais eficiência. Isto mostra que embora os processadores para sistemas embarcados possuíram boa eficiência do desempenho na exploração do paralelismo com MPI, eles não escalaram conforme os processadores de propósito geral.

Os resultados apresentados nesta seção mostram que a eficiência do desempenho varia conforme o processador, IPP utilizada e número de *threads*/processos. Portanto, a escolha adequada da IPP e processador pode levar ao melhor aproveitamento dos recursos computacionais. Por exemplo, paralelizar aplicações similares à classe MEM-B com OpenMP ao invés de Pthreads no ARM pode levar a obtenção de eficiência de desempenho 10% melhor para 2, 3 e 4 *threads*. Por outro lado, no processador Intel Core i7, a escolha por Pthreads pode levar a obter eficiência de até 11% melhor na execução com 4 *threads* comparado ao OpenMP.

5.5.2 Consumo de Energia

A Figura 5.12 apresenta a eficiência do consumo de energia dos melhores resultados obtidos para cada IPP em cada processador para as três classes de *benchmarks*. Conforme podemos acompanhar na mesma figura, a eficiência do consumo de energia decresce quando ocorre o aumento da exploração do TLP. Por exemplo, enquanto que na execução sequencial, obviamente a eficiência é de 100%, na exploração do paralelismo, a eficiência com duas *threads*/processos é aproximadamente a metade, com três *threads*/processos é aproximadamente 1/3; e assim por diante. Isto mostra que embora a paralelização aumente o desempenho da aplicação, o consumo total de energia da versão paralela é similar ao consumo da versão sequencial.

Os processadores de baixo consumo energético mostraram-se mais eficientes que os processadores de propósito geral: 2% nas aplicações CPU-B; 4% nas aplicações WMEM-B; e 6% nas aplicações MEM-B. Nas aplicações CPU-B (Figura 5.12a), embora OpenMP e Pthreads tiveram os melhores resultados em ambos os processadores, a diferença entre elas e MPI(-1 e -2) é insignificante (menos de 1%).

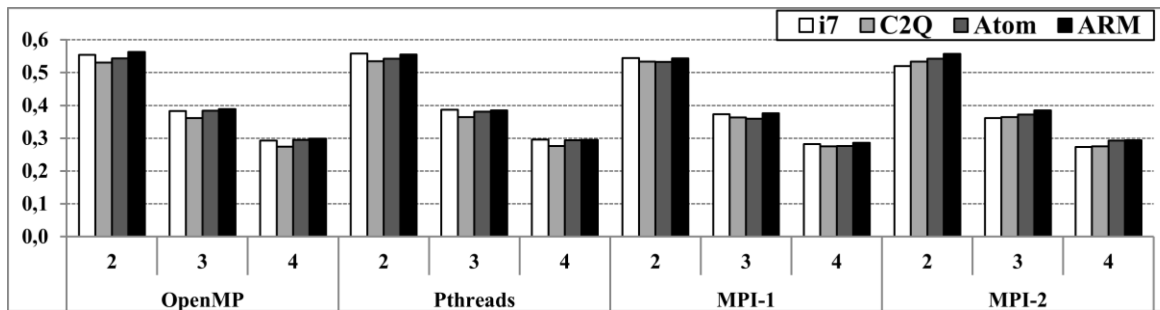
Para as aplicações WMEM-B, na Figura 5.12b, os melhores resultados de eficiência foram obtidos com os processadores para embarcados Atom e ARM. Pthreads e MPI-2 foram melhores no processador Atom, enquanto que OpenMP e MPI-1 foram melhores no ARM. Na

execução com OpenMP, o processador ARM possibilitou ganhos de até 8% comparados com os processadores Intel.

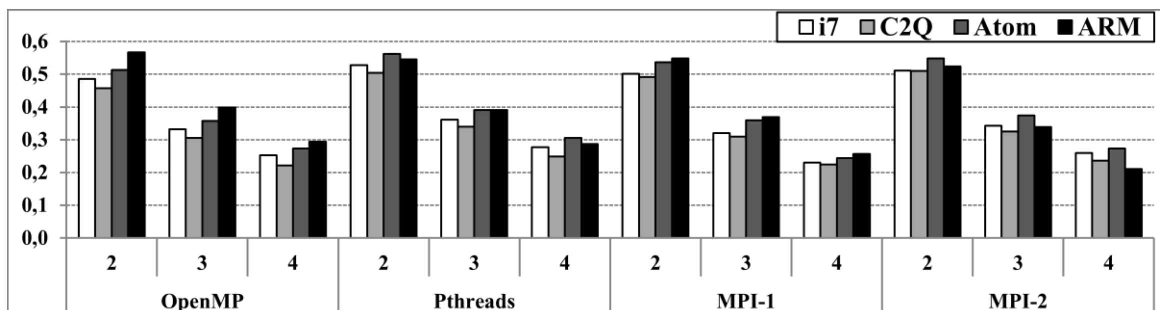
As aplicações MEM-B (Figura 5.12c) apresentaram os piores resultados considerando as três classes de *benchmarks*. Isto porque as aplicações desta classe possuem maior quantidade de comunicação, o que implica em maior energia gasta com a tarefa de comunicação. Assim, a eficiência de tais aplicações passa a ser limitada pela comunicação, o que acaba impactando em escalabilidade pior que as aplicações CPU-B e WMEM-B. Nesta classe, destaca-se novamente a eficiência do OpenMP no processador ARM, que é aproximadamente 5% melhor que os processadores Intel para 2 e 3 *threads*.

A Figura 5.13 apresenta os resultados médios de eficiência obtidos em cada IPP considerando a execução de todas as classes de *benchmark*. Pode-se notar que em todas as

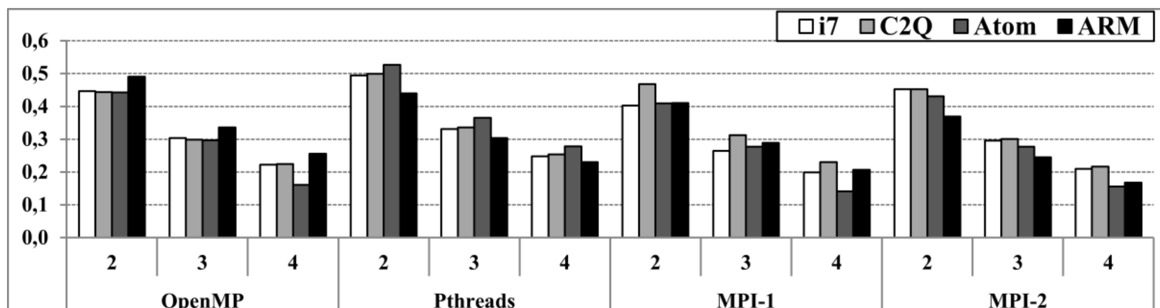
Figura 5.12 - Resultados de Eficiência/Escalabilidade da Energia



a) Aplicações CPU-B

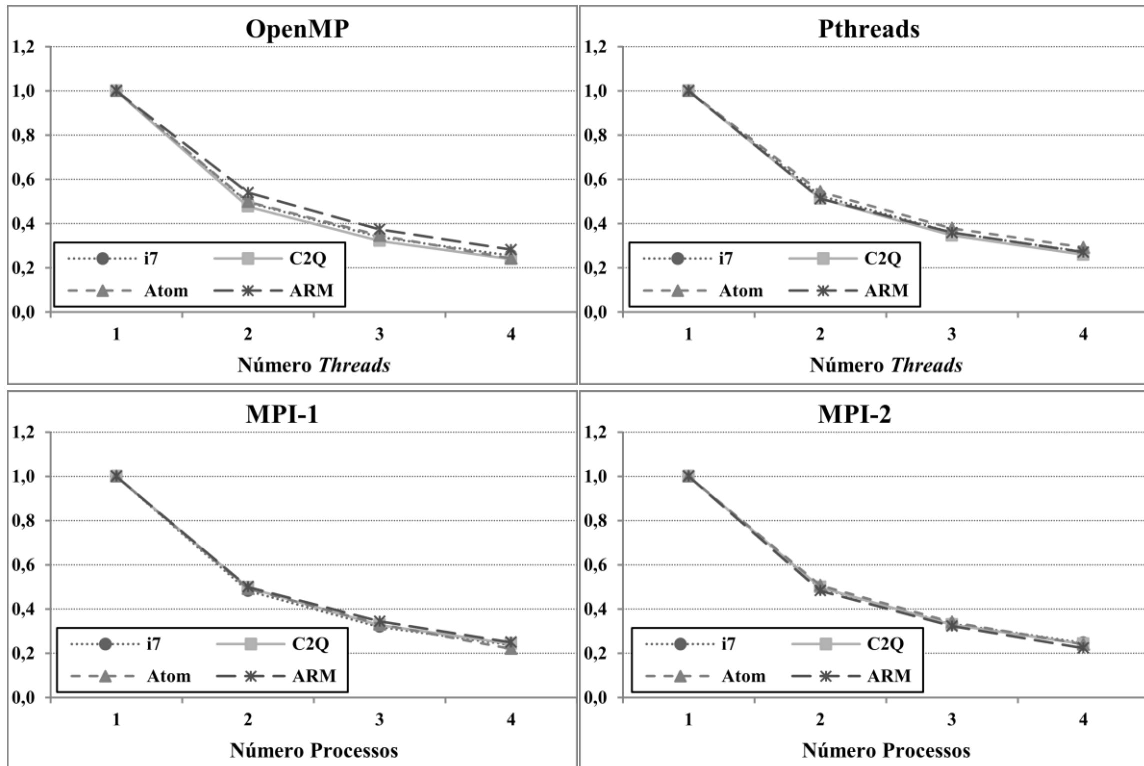


b) Aplicações WMEM-B



c) Aplicações MEM-B

Figura 5.13 - Resultado Médio de Eficiência/Escalabilidade da Energia por IPP



IPPs, conforme ocorre o aumento do número de *threads*/processos, a eficiência da energia decresce. Isto porque o consumo de energia não é dividido entre as *threads*/processos no paralelismo, como ocorre com o desempenho. No entanto, ao contrário da eficiência de desempenho apresentada na Seção 5.5.1, a eficiência da energia é bem similar entre os processadores. Embora exista diferença na eficiência entre os processadores, esta diferença é irrelevante (menos de 3% no caso com maior diferença – comparando MPI-1 nos processadores ARM e Atom com 4 processos). Isto significa que ambos os processadores escalaram de forma similar com relação ao consumo de energia.

5.5.3 *Energy-Delay Product*

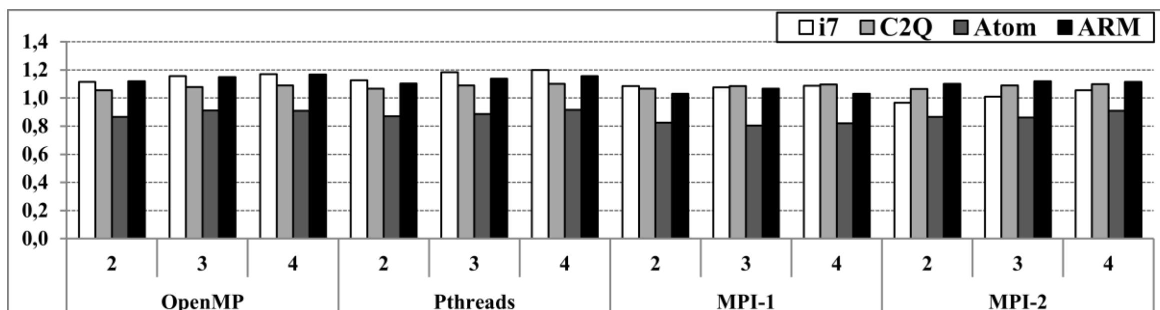
A Figura 5.14 apresenta os resultados da eficiência do EDP para cada IPP em cada processador considerando as três classes de *benchmarks*. Os melhores resultados de eficiência foram obtidos na execução das aplicações CPU-B, com média (entre todos os processadores, números de *threads*/processos e IPP) de 104% de eficiência, comparada com 83% nas aplicações WMEM-B e 69% nas aplicações MEM-B.

Nas aplicações CPU-B (Figura 5.14a), pode-se notar que a maioria das IPPs obteve resultados acima de 100% e que, aumentando o grau de exploração do paralelismo, a

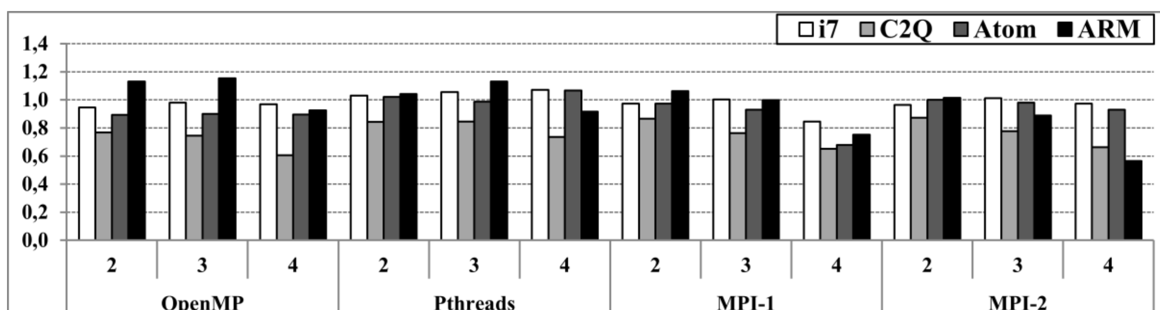
eficiência do EDP melhorou. Isto é possível, pois em alguns casos os resultados de desempenho foram próximos ou acima do ideal e o consumo de energia foi similar ou inferior ao consumo sequencial (vide Seções 5.2 e 5.3). Das IPPs, a que obteve melhor eficiência do EDP em todos os processadores foi Pthreads.

Considerando as aplicações WMEM-B (Figura 5.14b), na maioria das IPPs, a eficiência do EDP não acompanhou o aumento do grau de exploração do paralelismo. Na exploração do paralelismo com 2 e 3 *threads*/processos, o processador ARM obteve melhores resultados em todas as IPPs, com exceção do MPI-2. No entanto, aumentando a exploração de 3 para 4 *threads*/processos, houve perda substancial de aproximadamente 20% na eficiência das IPPs nesse processador. O mesmo não ocorre para Intel Core i7, que na exploração com 4 *threads*/processos a eficiência continuou similar à exploração de TLP com 3

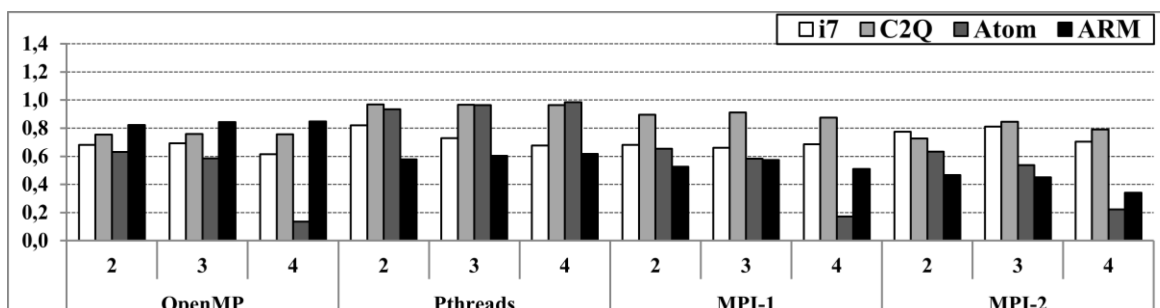
Figura 5.14 - Resultados de Eficiência/Escalabilidade do EDP



a) Aplicações CPU-B



b) Aplicações WMEM-B



c) Aplicações MEM-B

threads/processos, com exceção do MPI-1.

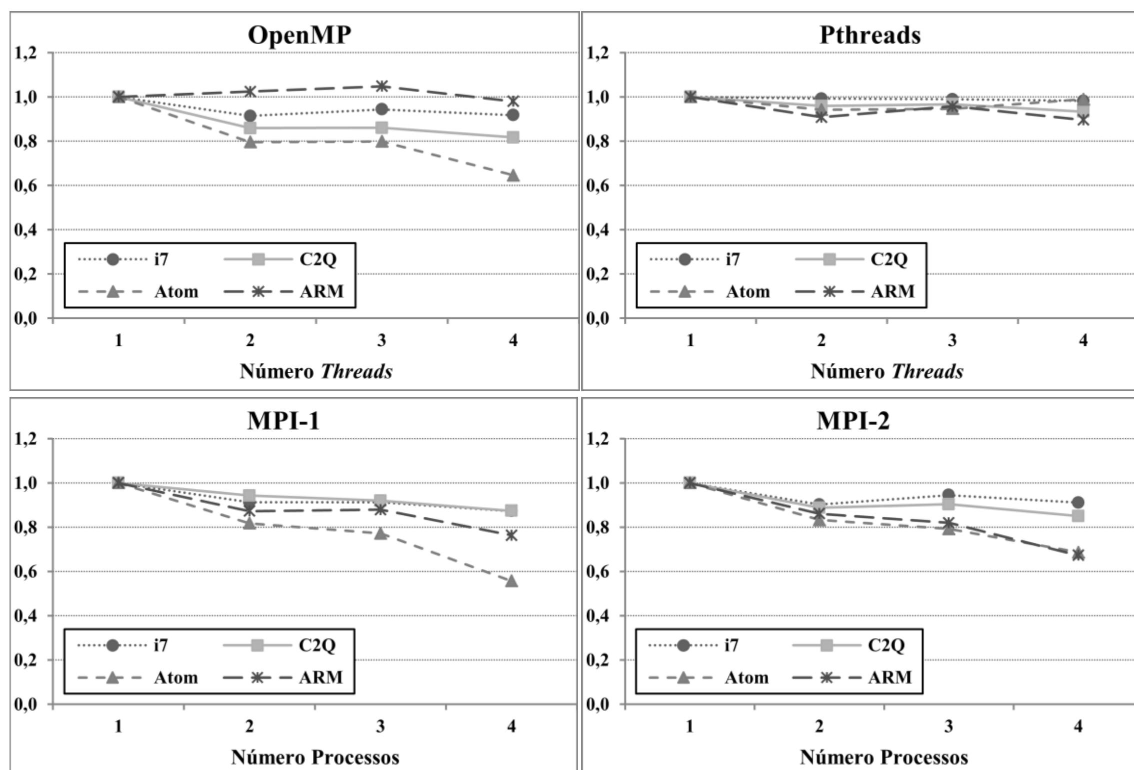
As aplicações MEM-B (Figura 5.14c) apresentaram os piores resultados de eficiência do EDP dentre as três classes, e também a pior escalabilidade quando ocorre o aumento de 3 para 4 *threads*/processos na maioria dos cenários (IPP e processador). Isto porque, obteve os piores resultados de desempenho e de consumo de energia devido à quantidade de comunicações e pontos de sincronização desta classe. Nesta classe, o melhor resultado no geral ocorreu com Pthreads executando nos processadores Core2Quad e Atom, ambos com eficiência e escalabilidade similar. O que mostra que processadores embarcados escalaram de forma similar aos processadores de propósito geral.

A Figura 5.15 apresenta os resultados médios de eficiência obtidos em cada IPP para a execução de todas as classes de *benchmark*. É importante ressaltar que melhor escalabilidade da eficiência não significa melhor desempenho, e sim que melhor escala conforme o grau de exploração do TLP aumenta. Assim, considerando inicialmente o OpenMP, ele obteve os melhores resultados de eficiência no processador ARM. Enquanto que com 2 e 3 *threads*, o ARM obteve ganhos na eficiência, os demais processadores perderam em eficiência. Isto representa que no aumento da exploração do TLP de 1 para 2 e 3 *threads*, o ARM escalou melhor que os demais processadores. Já na execução com 4 *threads*, o ARM escalou pior e perdeu 7% de eficiência enquanto que o Core i7 perdeu somente 2%. No entanto, mesmo com esta perda na eficiência, o ARM continuou com a melhor eficiência do EDP nas execuções com OpenMP.

Nos resultados com Pthreads (Figura 5.15), o Core i7 apresentou melhor eficiência do EDP com 2 e 3 *threads*. No entanto, aumentando o grau de exploração do TLP para 4 *threads*, enquanto Core i7 perdeu 1% de eficiência, o Atom ganhou 4% de eficiência. Assim, na execução com 4 *threads* usando Pthreads, o Atom obteve eficiência 1% melhor que o Core i7; 6% melhor que o Core2Quad; e 9% melhor que o ARM. O ARM obteve os piores resultados de eficiência da IPP Pthreads. Embora na execução com 3 *threads*, tenha obtido resultado similar ao Atom, quando a exploração do TLP passou para 4 *threads*, o ARM perdeu 6% de eficiência do EDP enquanto o Atom ganhou 4%. Esta perda de eficiência no ARM esta relacionada ao pior desempenho e consumo de energia obtido nas classes WMEM-B e MEM-B (vide Seção 5.2 e 5.3).

Considerando MPI-1 e MPI-2, observa-se que a escalabilidade não é similar a das IPPs Pthreads e OpenMP. Isto ocorre, principalmente, pois enquanto a comunicação das *threads* (OpenMP e Pthreads) é realizada através de acessos simultâneos a regiões compartilhadas da memória, em MPI tem-se o custo extra das tarefas de comunicação através de trocas de

Figura 5.15 - Resultados Médio de Eficiência/Escalabilidade do EDP



mensagens. No MPI-1, enquanto os processadores de propósito geral (Core2Quad e Core i7) possuíram os melhores resultados, os processadores para sistemas embarcados apresentaram perdas maiores na eficiência. Este mesmo comportamento ocorre com o MPI-2. Interessante notar que o fato de criar processos em tempo de execução permitiu melhores ganhos de eficiência do EDP nos processadores Atom e Core i7.

5.5.4 Análise Crítica

Com os resultados discutidos nesta seção, mostrou-se que a eficiência do EDP muda conforme a Interface de Programação Paralela, a arquitetura e organização, o número de *threads*/processos e a característica da aplicação utilizada. No nicho de aplicações que possuem uso intensivo da CPU e pouca comunicação (CPU-B), as IPPs alvo deste trabalho, no melhor caso, mostraram-se similar com relação à escalabilidade e eficiência do EDP em cada processador. Considerando a escalabilidade dos processadores, o processador para embarcados ARM Cortex-A9 obteve resultados similares aos processadores de propósito geral Intel Core i7 e Core2Quad. Portanto, para este nicho de aplicações, o desenvolvedor poderá optar entre qual IPP escolher analisando outros eixos, como por exemplo: facilidade de programação, controle da aplicação, etc.

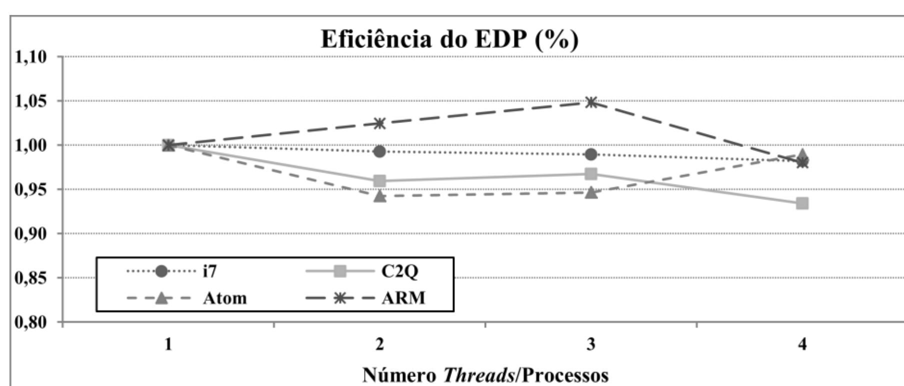
No nicho das aplicações que possuem comunicação semelhante à classe WMEM-B, na execução com 2 e 3 *threads*, o processador mais indicado é o ARM executando a IPP OpenMP, com eficiência de EDP aproximadamente 10% maior que o melhor resultado com os processadores de propósito geral (Pthreads executando no Core i7). No entanto, na exploração do TLP com 4 *threads*, a melhor opção é paralelizar com a IPP Pthreads e executar nos processadores Intel Core i7 ou Atom, que possuem a mesma eficiência.

Nas aplicações com características similares a classe MEM-B, o cenário que obteve melhor eficiência com 2 e 3 *threads* foi utilizando a IPP Pthreads no processador Core2Quad. No entanto, na exploração com 4 *threads*, o processador Atom novamente possuiu o melhor resultado com eficiência do EDP muito próxima a 100%. Se o programador estiver interessado em produtividade, ou seja, escrever códigos paralelos com maior facilidade, pode também utilizar o OpenMP e executá-lo no ARM, que obteve melhores resultados com tal IPP. Entretanto, com eficiência média do EDP 15% menor que Pthreads executando no Atom.

Na eficiência média das três classes, em termos de eficiência e escalabilidade do EDP, o OpenMP é indicado para exploração do paralelismo no ARM; Pthreads indicado para execução no Core i7, Core2Quad e Atom; MPI-1 para os processadores Core i7 e Core2Quad; e MPI-2 o mais indicado para o processador Core i7.

Por fim, a Figura 5.16 mostra o melhor resultado de eficiência do EDP obtido em cada processador, independente da IPP utilizada. Este gráfico mostra o comportamento de cada processador na escalabilidade da eficiência do EDP em seu melhor resultado. Pode-se notar na mesma figura, que o ARM possui melhor eficiência e escalabilidade que os demais processadores na exploração do TLP com 2 e 3 *threads*/processos. Porém, para 4 *threads*/processos, o ARM perdeu 7% de eficiência e assemelhou-se ao Core i7. Já no Atom, embora tenha perdido eficiência do EDP na exploração com 2 e 3 *threads*/processos, o aumento de 3 para 4 *threads*/processos melhorou a eficiência do EDP, fazendo com que

Figura 5.16 – Melhor Eficiência do EDP em cada Processador



atingisse a melhor eficiência com 4 *threads*/processos dentre todos os processadores alvo. Estes resultados mostram que se considerarmos o melhor caso de eficiência do EDP em cada processador, os processadores para embarcados possuíram melhor escalabilidade e eficiência que os de propósito geral. Novamente, cabe ressaltar que melhor eficiência e escalabilidade não significam melhor desempenho, ou seja, menor tempo de execução; e sim que aproveitam melhor os recursos computacionais disponíveis.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho objetivou analisar o impacto no desempenho, consumo de energia e eficiência energética decorrente da extração do paralelismo com diferentes Interfaces de Programação Paralela e investigar em sistemas embarcados (Intel Atom e ARM Cortex-A9) e de propósito geral (Intel Core i7 e Core2Quad). Para tanto, 12 aplicações divididas em três classes de acordo com a quantidade de acessos a memória privada e compartilhada, quantidade de pontos de sincronização e dependência de dados foram desenvolvidas e analisadas. As Interfaces de Programação Paralela usadas para os experimentos representam as largamente utilizadas atualmente: OpenMP, Pthreads, MPI-1 e MPI-2.

Nossos resultados mostram que cada Interface de Programação Paralela possui comportamento diferente na quantidade de instruções executadas e acessos a memória de dados, impactando assim, de forma diferente no desempenho, consumo de energia e eficiência energética dos processadores. No nicho de aplicações com uso intensivo de CPU e pouca comunicação (CPU-*Bound*), observou-se que as IPPs possuem comportamento similar na eficiência energética. Embora Pthreads tenha obtido os melhores resultados de desempenho, consumo de energia e EDP, mostrou-se que para esta classe é possível obter resultados próximos do ideal com pouco esforço de programação através do OpenMP. No comportamento dos processadores, a comparação foi prejudicada pelo impacto das execuções de instruções que influenciaram no resultado do ARM. No entanto, o processador que obteve menor consumo de energia para esta classe foi o Atom: economizou 32% comparado ao Core i7 e Core2Quad; e 26% comparado ao ARM.

Considerando as aplicações WMEM-B, as quais exploram um pouco das características de comunicação e sincronização entre *threads*/processos, mostrou-se que as IPPs possuem comportamento diferente nos processadores embarcados e de propósito geral. Pthreads possuiu eficiência do EDP média de 8% melhor que o melhor caso em OpenMP no processador Core i7; 11% melhor que MPI-1; e 7% melhor que MPI-2. Já no ARM, o melhor caso do OpenMP obteve eficiência 4% melhor que Pthreads; 13% melhor que MPI-1; e 25% melhor que MPI-2. Ademais, no geral, o ARM economizou 60% de energia comparado ao Core i7, 65% comparado ao Core2Quad e 32% comparado ao Atom.

Para as aplicações MEM-B, a diferença na economia de energia do ARM com relação aos processadores de propósito geral aumentou para: 65% comparado ao Core i7 e 68% comparado ao Core2Quad. Já no Atom, a diferença seguiu a mesma da classe anterior: 32%. A diferença na eficiência do EDP das IPPs também aumentou. Pthreads foi 21% mais

eficiente que o melhor caso do OpenMP; 8% que MPI-1 e 18% que MPI-2 no processador Core2Quad (melhor resultado nos processadores de propósito geral). Já nos processadores embarcados, notou-se diferença no comportamento do OpenMP e Pthreads novamente. Pthreads foi melhor no Atom, com eficiência do EDP de aproximadamente 51% melhor que OpenMP, enquanto que, no ARM, OpenMP foi aproximadamente 24% mais eficiente que Pthreads.

Por fim, mostrou-se que embora os processadores de propósito geral possuem melhor desempenho (menor tempo de execução da aplicação), quando aumenta o grau de exploração do TLP, a perda de eficiência do EDP é maior nestes processadores que nos embarcados. Por exemplo, considerando o melhor resultado da eficiência do EDP em cada processador, os processadores embarcados obtiveram os melhores resultados na exploração com 2 e 3 *threads*/processos no ARM Cortex-A9; e com 4 *threads*/processos no processador Atom.

6.1 Trabalhos Futuros

6.1.1 Expansão do Conjunto de *Benchmark*

Conforme pôde ser notado na Seção 4.1, o conjunto de *benchmarks* escolhido para este trabalho possuiu nível de TLP muito próximo do ideal. Uma oportunidade de trabalho futuro é expandir este conjunto para aplicações que permitam cobrir aplicações com diferentes comportamentos (por exemplo, baixo nível de TLP) e considerar outros fatores, como paralelismo no nível de instrução e aplicações que são mais orientadas a fluxo de dados ou de controle.

Também se pretende desconsiderar as aplicações presentes neste trabalho que de certa forma foram injustas com o processador ARM. Conforme pôde ser acompanhado na discussão dos resultados nas Seções 5.1 e 5.2, as aplicações Cálculo do Pi, Conjunto de *Mandelbrot* e Série Harmônica tiveram maior número de instruções executadas e pior desempenho. Isto porque enquanto as operações *sqrt*, *sqrtf*, *mod* são executadas em uma única instrução no Intel, no ARM elas são substituídas por algoritmos menos eficientes. Portanto, surge como alternativa a implementação de novas aplicações que permitam abordar

unicamente as características das Interfaces de Programação Paralela, sem ter a influência de tais características que possam limitar a análise.

6.1.2 Análise de Diferentes Compiladores

Sabe-se que o uso de diferentes compiladores pode ter impacto diferente nos resultados. Outro fator que pode impactar são as otimizações providas por tais compiladores. Neste trabalho, não foi utilizada nenhuma otimização. Portanto, surge como trabalho futuro a exploração do uso de diferentes compiladores (GCC, LLVM – *Low Level Virtual Machine* -, etc.) com diferentes níveis de otimização. Ademais, o uso de diferentes níveis de otimização, como por exemplo, O1, O2, O3 e OS, também serão analisados.

6.1.3 Impacto das Boas Práticas de Programação

Conforme resultados discutidos na Seção 5.1 e 5.2, o impacto das boas práticas de programação teve comportamento diferente nos processadores ARM e Intel. Enquanto que no Intel, o uso de variáveis locais (internas a *main*) contribuiu para impactar de forma negativa no número de instruções executadas, no ARM, aconteceu o contrário. Desta maneira, este assunto envolve verificar se boas práticas de programação levará a melhor eficiência energética; se isto se repetiria para todas as arquiteturas, etc.

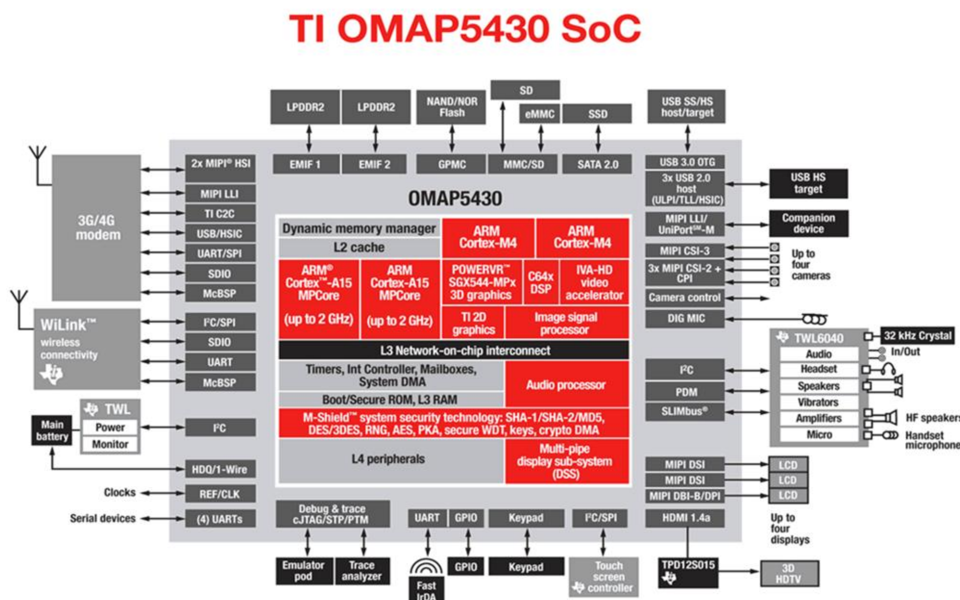
6.1.4 Investigar o Impacto do uso de Comunicações Coletivas

Diferentes são as formas de realizar comunicações coletivas através de trocas de mensagens entre processos: *single-accumulation*, *scatter*, *gather*, *multi-broadcast*, *multi-accumulation* e *total exchange*. Surge como oportunidade de trabalhos futuro, investigar o impacto de tais diretivas de comunicação no desempenho, consumo de energia e eficiência do *Energy-Delay Product* em diferentes arquiteturas.

6.1.5 Arquiteturas Heterogêneas

No cenário em que é necessário aumentar o desempenho com limitações no consumo de energia, arquiteturas heterogêneas surgem como alternativa. Elas mesclam processadores

Figura 6.1 – Texas OMAP 5

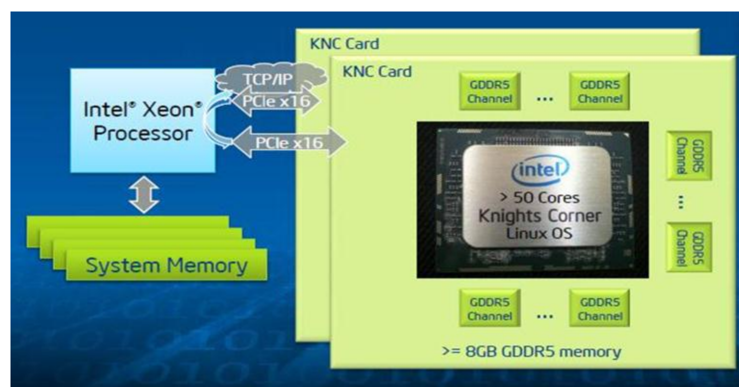


Fonte: Texas Instruments

de propósito geral com processadores de propósito específico. Um dos principais exemplos são os *Systems on Chip* (SoCs) (e.g.: Texas OMAP, *Qualcomm Snapdragon, Apple-7*). Eles são compostos por processadores de propósito geral e aceleradores desenvolvidos para executar as aplicações de interesse (decodificação de vídeo e áudio, por exemplo). A Figura 6.1 mostra o *System on Chip* OMAP 5, da empresa *Texas Instruments*. Nela podemos ver que no SoC existem processadores específicos para decodificação de áudio (*Audio processor*), imagem (*Image signal processor*), acelerador de vídeo (*IVA-HD vídeo accelerator*), que executam tais operações específicas com ótimo desempenho e baixo consumo de energia; processadores de uso geral para executar tarefas que necessitam maior capacidade de processamento (ARM Cortex-A15) e processadores de uso geral com baixo consumo de energia para realizar tarefas simples (ARM Cortex-M4). Portanto, em tempo de projeto, inúmeras combinações de componentes podem ser utilizadas para atingir determinados requerimentos do sistema (e.g.: desempenho, potência, área e energia).

Como pôde ser observado no exemplo anterior, *Systems on Chip* geralmente são heterogêneos em arquitetura. Entretanto, processadores de mesma arquitetura com diferentes organizações também podem ser utilizados para obter diferentes requisitos de processamento. Por exemplo, o ARM Cortex-A15 pode ser usado para executar aplicações que necessitem maior capacidade de processamento, e processadores Cortex-A7, com foco em baixo consumo de energia, que pode ser usado para executar aplicações simples.

Figura 6.2 - Exemplo de combinação de uma CPU com o Intel Xeon Phi



Fonte: Intel®

6.1.5.1 Intel Xeon Phi

A Intel, com o Xeon Phi, também apresenta uma solução heterogênea para aliar computação de alto desempenho com baixo consumo energético através do uso de coprocessadores. Dentre suas principais funções, estão a realização de operações em aritmética de ponto flutuante, computação gráfica, processamento de sinais e *strings*, criptografia entre outras[3].

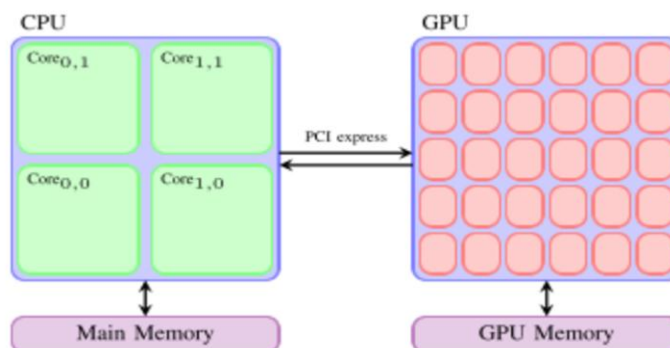
O *Intel Xeon Phi*[4] é baseado na arquitetura Intel *Many Integrated Core* (MIC) (JEFFERS et al., 2013). O *Xeon Phi* possui até 61 núcleos, suportando a execução simultânea de até 244 *threads* para fornecer desempenho de até 1.2 *teraflops* com baixo consumo energético (SHAO et al., 2013) (RAMACHANDRAN et al., 2013). Assim, os trechos de código que necessitam de grande poder computacional podem ser executados no coprocessador *Xeon Phi*, enquanto que aplicações com baixa necessidade computacional são executadas no processador primário/principal. A Figura 6.2 apresenta um exemplo em alto nível de abstração da combinação do processador Intel *Xeon* e do coprocessador Intel *Xeon Phi,* em que a comunicação entre eles é feita por PCI *Express* x16. O protocolo TCP/IP é usado para realizar a comunicação entre o *Xeon* e o *Xeon Phi*.

---

[3] Intel Delivers New Architecture for Discovery with Intel® Xeon Phi™ Coprocessors

[4] http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html

Figura 6.3 – Exemplo de Combinação de uma CPU com uma GPU



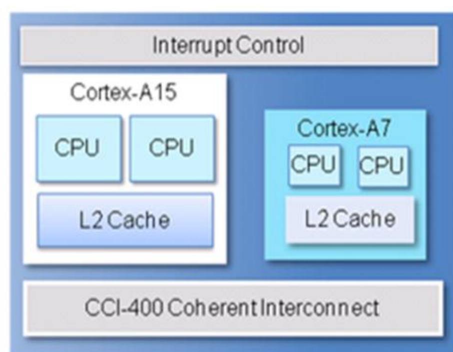
Fonte: (BRODTKORB et. al 2010)

6.1.5.2 Unidades de Processamento Gráfico

Da mesma maneira, as Unidades de Processamento Gráfico (GPUs – *Graphics Processing Units*) tornaram-se uma grande ferramenta para a computação massivamente paralela. Estes processadores gráficos trabalham em conjunto com uma CPU para acelerar aplicações científicas e corporativas. Lançadas pela Nvidia em 2007, as GPUs agora estão mais presentes em *data centers* devido ao uso eficiente de energia (HUANG et al., 2009).

Atualmente, os dois principais fabricantes de GPUs são a NVIDIA e a AMD, que podem atuar em conjunto com CPUs Intel ou AMD. O paralelismo a ser explorado em GPUs é do tipo SIMD (*Single Instruction, Multiple Data*), onde milhares de *threads* executam simultaneamente nas centenas de núcleos presentes na GPU. O programa principal executa na CPU (chamado de *host*) e é o responsável por iniciar as *threads* na GPU (*device*). As GPUs tem sua própria hierarquia de memória e os dados devem ser transferidos da memória principal (do lado do *host*) através de um barramento *PCI Express*. A Figura 6.3 mostra um esquema genérico de um sistema heterogêneo composto por uma CPU e uma GPU. Conforme Abe et al. (2012), as GPUs provem ganhos significantes em desempenho por watt quando comparado com processadores *multicore* tradicionais para determinadas classes de aplicações (principalmente vetoriais e matriciais).

Figura 6.4 – Exemplo de *SoC mobile* heterogêneo



Fonte: ARM

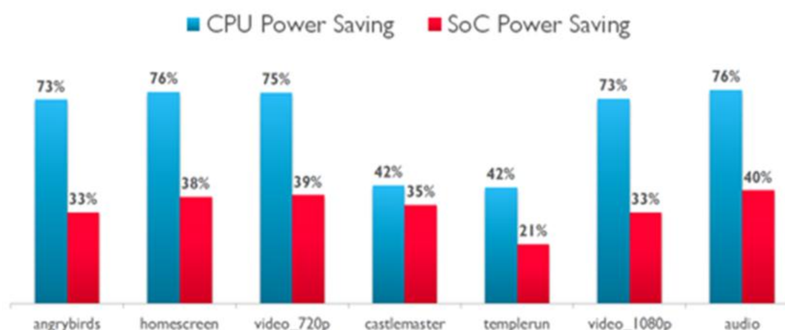
6.1.5.3 Tecnologia big.LITTLE

Se por um lado as empresas estão se adaptando a necessidade de desenvolver processadores com boa eficiência energética, do outro, surgem tecnologias que auxiliam na redução do consumo de energia destas arquiteturas. Um exemplo é a tecnologia ARM *big.LITTLE*, usada para economizar energia em *Systems on Chip* móveis.

A tecnologia *big.LITTLE* combina processadores ARM de alto desempenho com processadores ARM com baixo consumo energético para proporcionar capacidade máxima de performance com consumo de energia relativamente baixo. A Figura 6.4 apresenta um SoC com um processador de alto desempenho, chamado de *big* (Cortex-A15) em conjunto com um de baixo consumo energético, chamado de *LITTLE* (Cortex-A7). Este tipo de tecnologia está presente nos processadores mais atuais, como por exemplo, o *Samsung Galaxy* S4 e S5.

Em conjunto com a tecnologia *big.LITTLE*, a ARM tem desenvolvido um conjunto de alterações para o *kernel* do sistema operacional. Dentre elas, a capacidade do escalonador do sistema operacional decidir em tempo de execução qual processador é mais apropriado (*big* ou *LITTLE*) para executar determinada tarefa. Com este escalonador, o *big.LITTLE* consegue obter reduções no consumo de energia. A Figura 6.5 apresenta o quanto de economia pode ser obtido com a utilização do processador *big.LITTLE.* Nela, podemos observar que a escolha ideal do processador para executar determinada tarefa pode representar a economia de até 76% na CPU sem impactar no desempenho da mesma, o que representa economia de até 40% em todo o sistema multicore.

Figura 6.5 - Redução do Consumo de Energia em Arquiteturas Heterogêneas



Fonte: http://www.arm.com/products/processors/technologies/biglittleprocessing.php)

6.1.5.4 Dynamic Voltage and Frequency Scaling - DVFS

Outra tecnologia utilizada para reduzir o consumo de energia é o *Dynamic Voltage and Frequency Scaling* (DVFS). Ele tem sido amplamente utilizado em dispositivos móveis, computação de propósito geral e de alto desempenho, e recentemente está sendo empregada em processadores gráficos (e.g.: K20c da Nvidia) (RONG et al., 2013). O objetivo de DVFS é diminuir o estado de energia de um componente sem deixar de cumprir a exigência de desempenho. Ou seja, este mecanismo é aplicado sobre o processador, modificando sua frequência e voltagem de operação de modo a economizar energia (KAHNG et al., 2013). Outras podem ser citadas, como: *Clock Gating, Core Power Gating, Retention Nodes,* e *Thermal Management* (LEVERICH et al., 2009) (ANNAVARAM, 2011).

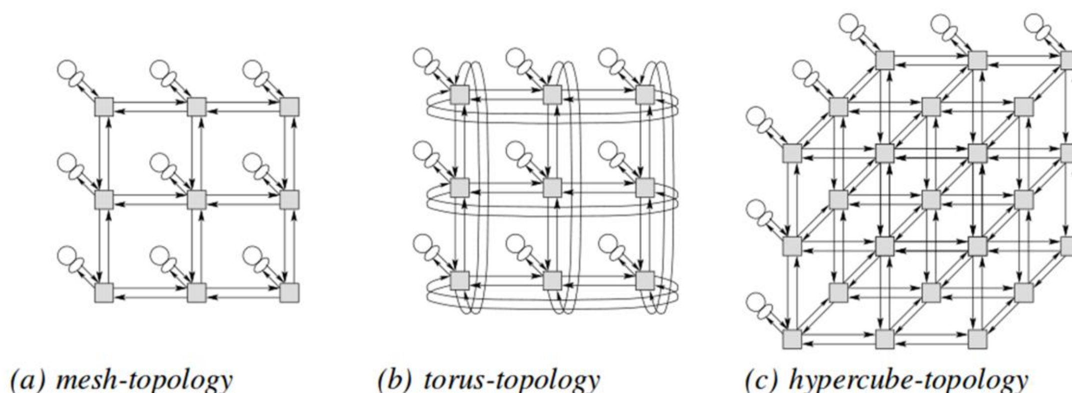
6.1.5.5 Network-on-Chips

A partir da discussão acima, pode-se observar que há um aumento no número de unidades de processamento. Assim, a comunicação entre eles através do uso de barramentos começa a se tornar o gargalo da computação. Desta forma, surgem alternativas para contornar tal gargalo. A principal delas, chamada *Network-on-Chips* (NoC) já começaram a ser utilizadas (e.g.: *TILEPro64 Processor*[5] (ZIMMER et al., 2012)). Uma NoC é uma rede de nós interconectados, onde cada nó está associado a um núcleo de processamento, controlador de memória, etc. As principais vantagens da utilização de NoCs são a regularidade da rede de

---

[5] http://www.tilera.com/products/processors/TILePro_Family

Figura 6.6 – Exemplos de topologia NoC



interconexão, a redução dos efeitos eletromagnéticos e a boa escalabilidade (WELDEZION et al., 2009). NoCs podem ser implementadas considerando diferentes topologias regulares, como *mesh*, *torus* e *hypercube*, conforme apresentado na Figura 6.6. A Intel passou a utilizar alguns conceitos de NoCs em 2008 através do *Quickpath interconnect* (QPI)[6], em diversas versões dos processadores Intel Core i3, i5, i7 e Intel *Xeon.*

Conforme pôde ser observado, o espaço de exploração de projeto mostrado acima é muito vasto. Por exemplo, um SoC que implementa uma NoC dotado de DVFS e que usa a IPP Pthreads pode possuir melhor eficiência energética quando comparada com uma arquitetura heterogênea contendo processadores de alto desempenho e GPUs executando com a IPP CUDA.

---

[6] http://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html

REFERÊNCIAS

ABE, Y.; SASAKI, H.; PERES, M.; INOUE, K.; MURAKAMI, K.; KATO, S.: Power and Performance Analysis of GPU-Accelerated Systems. In: USENIX CONFERENCE ON POWER-AWARE COMPUTING AND SYSTEMS, 2012, Proceedings. Berkeley, CA, USA: USENIX, 2012 p. 10

ADVE, V. S.; VERNON, M. K.: **A Deterministic Model for Parallel Program Evaluate Performance Evaluation**. Techreport in Rice University and University of Wisconsin-Madison. 1998.

AJKUNIC, E.; FATKIC, H.; OMEROVIC, E.; TALIC, K.; NOSOVIC, N.: A Comparison of five Parallel Programming Models for C++. In: MIPRO, May 2012, **Proceedings.** Opatija: IEEE, 2012 p.1780,1784.

AKKAN, H.; LANG, M.; IONKOV, L.: HPC Runtime Support for Fast and Power Efficient Locking and Synchronization. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, Sept. 2013, **Proceedings.** Indianapolis –IN: IEEE, 2013. p. 1-7.

ALTMAN, E. R.; KAELI, D.; SCHEFFER, Y.: Welcome to the Opportunities of Binary Translation**. Computer**, [S.l.]: IEEE, v. 33, nº 3. p. 40-45. March, 2000.

ANDREWS, G.E.; ASKEY, R.; ROY, R.: **Special Functions.** Cambridge University Press. 1999.

ANNAVARAM, M.: A case for guarded power gating for multi-core processors. In: IEEE 17th HPCA, Feb. 2011, **Proceedings**. Washington, DC, USA: IEEE, 2011. p.291,300

ARM Cortex-A9 Technical Reference Manual. Disponível em: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f/DDI0388F_cortex_a9_r2p2_trm.pdf 2010.

BALLADINI, J.; SUPPI, R.; REXACHS, D.; LUQUE, E.: Impact of Parallel Programming Models and CPUs Clock Frequency on Energy Consumption of HPC Systems. In: 9TH IEEE/ACS INTERNATIONAL CONFERENCE ON COMPUTER SYSTEMS AND APPLICATIONS, Dec. 2011, **Proceedings**. Sharm El-Sheikh: IEEE, 2011. p. 16-21.

BLAKE, G.; DRESLINSKI, R.G.; MUDGE, T.; FLAUTNER, K.: Evolution of Thread-Level parallelism in Desktop Applications. In: 37th ANNUAL INTERNATIONAL SYMPOSIUM

ON COMPUTER ARCHITECTURE, 2010, **Proceedings.** New York, NY, USA: ACM, 2010. p. 302-313.

BLEM, E.; MENON, J.; SANKARALINGAM, K.: **A Detailed Analysis of the Contemporary ARM and x86 Architectures**. UW-Madison Technical Report. 2013.

BORKAR, S.; CHIEN, A. A.: The Future of Microprocessors. **Communications of the ACM**, New York, NY, USA: ACM, Vol 54, nº 5, p. 67-77, May, 2011.

BOUMAN, D. S.: **Parallelizing a Skyline Matrix Solver using Orca**. Student Project Report. Disponivel em: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.436&rep=rep1&type=pdf

BRODTKORB, A. R.; DYKEN, C.; HAGEN, T.R.; HJELMERVIK, J. M.: State of the art in Heterogeneous Computing. **Scientific Programming**, Netherlands, IOS Press Amsterdam, vol. 18, nº 1, p. 1-33, January, 2010.

BROWNE, S., DONGARRA, J., GARNER, N., HO, G., MUCCI, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. **The International Journal of High Performance Computing Applications**, Thousand Oaks, CA, USA: Sage Publications, Vol. 14, nº 3, p. 189-204, August, 2000.

BUTENHOF, D. R.: **Programming with POSIX threads**. Addison-Wesley Longman Publishing Co. Boston, USA, 1997.

CAO, Y.; WU, B.; TAO, Y.; SHI, L.: Performance Analysis of Current Parallel Programming Models for Many-core Systems. In: 8th INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE & EDUCATION (ICCSE), April, 2013, **Proceedings**. Colombo, Sri Lanka: IEEE, 2013. p. 132-135

CAPPELLO, F.; GEIST, A.; GROPP, B.; KALE, L.; KRAMER, B.; SNIR, M.: Toward Exascale Resilience. **International Journal of High Performance Computing Applications**. Thousand Oaks, CA, USA: Sage Publications, Inc. Vol. 23, nº 4, p. 374-388, November, 2009.

CERA, M. C., PEZZI, G. P., MATHIAS, E. N., MAILLARD, N., NAVAUX, P. O. A.: Improving the Dynamic Creation of Processes in MPI-2. In: 13th EUROPEAN PVM/MPI USER’S GROUP CONFERENCE ON RECENT ADVANCES IN PARALLEL VIRTUAL MACHINE AND MESSAGE PASSING INTERFACE, 2006, **Proceedings**. Berlin, Heidelber: Springer-Verlag, 2006. p. 242-257.

CERA, M. C., GEORGIOU, Y., RICHARD, O., MAILLARD, N., NAVAUX, P. O. A.: Supporting Malleability in Parallel Architectures with Dynamic Cpusets Mapping and Dynamic MPI. In: 11th INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING AND NETWORKING, 2010, **Proceedings.** Berlin, Heidelberg; Springer-Verlag, 2010.

CHADHA, G.; MAHLKE, S.; NARAYANASAMY, S.: When Less Is More (LIMO): Controlled Parallelism for Improved Efficiency. In: CASES, 2012, **Proceedings**. New York, NY, USA: ACM, 2012. p. 141-150

CHANDRA, R.; DAGUM, L.; KOHR, D.; MAYDAN, D.; McDONALD, J.; MENON, R.: **Parallel Programming in OpenMP**. Morgan Kaufmann, San Francisco, 2001.

CHAPMAN, B.; JOST, G.; VAN DER PAS, G.: **Using OpenMP:** Portable Shared Memory Parallel Programming. Cambridge, MIT Press, 2008.

CHEN, J.; DONG, Y. YANG, X.; WANG, P.: Energy-Constrained OpenMP Static Loop Scheduling. In: HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS, Sept. 2008, **Proceedings**. Dalian: IEEE, 2008. p. 139-146.

CHENEY, W.; KINCAID, D.: **Linear Algebra:** Theory and Applications. [S.l:s.n.], pp. 544-558.

DEEPAK SHEKHAR, T.C.; VARAGANTI, K.; SURESH, R.; GARG, R.; RAMAMOORTHY, R.: Comparison of Parallel Programming Models for Multicore Architectures. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING WORKSHOPS AND PHD FORUM (IPDPSW), May, 2011, **Proceedings.** Shangahai: IEEE, 2011. p.1675 - 1682.

DIJKSTRA, E. W.: A Note on two Problems in Connexion with Graphs. **Numerische Mathematik**, [S. l.], vol. 1, nº 1, p. 269-271. Dec. 1959.

DONG, Y.; CHEN, J.: Energy Optimization on OpenMP Loop Scheduling. **Journal of Software**, [S.l.], vol. 7, nº. 8, p. 1694 – 1705, Nov. 2012.

EGGERS, S. J,; EMER, J. S.; LEVY, H. M.; LO, J. L.; STAMM, R. L.; TULLSEN, D. M.: Simultaneous Multithreading: A Platform for Next-Generation Processors. **IEEE Micro**, [S.l.], vo. 17, nº 5, p. 12 – 19. Sep/Oct 1997.

ESMAEILZADEH, H.; BLEM, E.; AMANT, R. S.; SANKARALINGAM, K.; BURGER, D.: Power Limitations and Dark Silicon Challenge the Future of Multicore. **ACM Transactions**

on Computer Systems. New York, NY, USA: ACM. vol 30, nº 3, Article nº 11, August 2012.

FREDRICKSON, N. R.; AFSAHI, A.; QIAN, Y.: Performance Characteristics of OpenMP Constructs, and Application Benchmarks on a Large Symmetric Multiprocessor. In: 17th ANNUAL INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 2003, **Proceedings.** New York, NY, USA: ACM, 2003. p. 140-149.

FOSTER, I. T.: **Designing and Building Parallel Programs:** Concepts and Tools for Parallel Software Engineering. [S.l.]: Addison-Wesley, 1995.

GAO, C.; GUTIERREZ, A.; DRESLINSKI, R.G.; MUDGE, T.; FLAUTNER, K.; BLAKE, G.: A Study of Thread Level Parallelism on Mobile Devices. In: IEEE ISPASS, March 2014, **Proceedings**. Monterey, CA: IEEE, 2014 pp. 126-127.

GARDNER, M.: Mathematical Games – The Fantastic Combinations of John Conway's new Solitaire Game of Life. **Scientific American**, [S.l.] vol. 223, nº 5, p. 120-123. Oct. 1970.

GE, R.; FENG, X.; SONG, S.; CHANG, H.; LI, D.; CAMERON, K: PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications. **IEEE Transactions on Parallel Distributed Systems**, [S.l.], vol. 21, nº 5, p. 658-671. May, 2010.

GOLDSTON, D.; YILDIRIM, C. Y.: **On the Second Moment for Primes in an Arithmetic Progression.** Mathematics – Number Theory. 2000.

GRAMA A.; KARYPIS, G.; KUMAR, V.; GUPTA, A.: **Introduction to Parallel Computing**. 2nd edn. Addison-Wesley, Boston, 2003.

GROPP, W. et al.: **MPI – The Complete Reference**. Cambridge, MA, MIT Press, 1998.

GROPP, W.; LUSK, E.; THAKUR, R.: **Using MPI-2: Advanced Features of the Message Passing Interface.** MIT Press, Cambridge, MA. 1998.

HANAWA, T.; SATO, M.; LEE, J.; IMADA, T.; KIMURA, H.; BOKU, T.: **Evaluation of Multicore Processors for Embedded Systems by Parallel Benchmark Program using OpenMP**. In Lecture Notes in Computer Science, Springer, Vol 5568, p. 15-27. 2009.

HENNESSY, J. L; PATTERSON, D. A.: **Computer Architecture – A Quantitative Approach (5a ed.)**. Morgan Kaufmann. 2012.

HOCHSTEIN, L.; CARVER, J.; SHULL, F.; ASGARI, S.; BASILI, V.; HOLLINGSWORTH, J. .; ZELKOWITZ, M. V.: Parallel Programmer Productivity: A Case

Study of Novice Parallel Programmers. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2005, **Proceedings.** Washington, DC, USA: IEEE, 2005. p 35

HOROWITZ, M.: Scaling, Power and the Future of CMOS. In: VLSID, 2007, **Proceedings**. Washington, DC, USA: IEEE, 2007. p. 23.

HUA, S.; YANG, Z.: Comparison and Analysis of Parallel Computing Performance Using OpenMP and MPI. **The Open Automation and Control System Journal**, [S.l.], vol. 5, p. 38-44. 2013.

HUANG, S.; XIAO, S.; FENG, W.: On the Energy Efficiency of Graphics Processing Units for Scientific Computing. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING, 2009, **Proceedings.** Washington, DC, USA: IEEE, 2009. p 1-8.

Intel® Atom™ Processor D2000 and N2000 Series Datasheet, Vol. 1 Disponível em: http://www.intel.com/content/www/us/en/processors/atom/atom-d2000-n2000-vol-1-datasheet.html 2012.

Intel® Core™ i7-800/i5-700 Desktop Processors: Datasheet, Vol. 1. Disponível em: http://www.intel.com/content/www/us/en/intelligent-systems/piketon/core-i7-800-i5-700-desktop-datasheet-vol-1.html 2010.

Intel® Core™2Extreme Processor QX9000 Series, Intel® Core™2Quad Processor Q9000, Q9000S, Q8000, and Q8000S Series Datasheet. Disponível em: http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/core2-qx9000-q9000-q8000-datasheet.pdf 2009.

JEFFERS, J.; REINDERS, J.: **Intel Xeon Phi Coprocessor High Performance Programming**. Morgan Kaufmann Publishers Inc. San Francisco, USA, 2013.

JI, J.; WANG, C.; ZHOU, X.: System-Level Early Power Estimation for Memory Subsystem in Embedded Systems. In: FIFTH IEE INTERNATIONAL SYMPOSIUM ON EMBEDDED COMPUTING, Oct. 2008, **Proceedings.** Beijing: IEEE, 2008. p. 370-375.

JOAO, J. A.; SULEMAN, M. A.; MUTLU, O.; PATT, Y. N.: Bottleneck Identification and Scheduling in Multithreaded Applications. In: ASPLOS, 2012, **Proceedings.** New York, NY, USA: ACM, 2012. p. 223-234.

JOHN, N.: **Analyzing PAPI Performance on Virtual Machines.** ICL Technical Report, ICL-UT-13-02, August, 2013.

JOHNSON, M.; MCCRAW, H.; MOORE, S.; MUCCI, P.; NELSON, J.; TERPSTRA, D.; WEAVER, V.; MOHAN, T.: PAPI-V: Performance Monitoring for Virtual Machines. In: CLOUDTECH-HPC, Sept. 2012, **Proceedings.** Pittsburgh, PA: IEEE, 2012. p. 194-199.

KAHNG, A. B.; SEOKHYEONG, K.; KUMAR, R.; SARTORI, J.: Enhancing the Efficiency of Energy-Constrained DVFS Designs. **IEEE Transactions on Very Large Scale Integration Systems**, [S.l.], Vol. 21, no. 10, p. 1769-1782, Oct. 2013.

KASIM, H.; MARCH, V.; ZHANG, R.; SEE, S.: Survey on Parallel Programming Model. In: IFIP INTERNATIONAL CONFERENCE ON NETWORK AND PARALLEL COMPUTING, 2008, **Proceedings.** Berlin, Heidelberg: Springer-Verlag, 2008.p. 266-275.

KORTHIKANTI, V. A.; AGHA, G.: Towards Optimizing Energy Costs of Algorithms for Shared Memory Architectures. In: 22nd ACM SPAA, 2010, P**roceedings.** New York, NY, USA: ACM, 2010. p. 157-165.

KRAWEZIK, G.; CAPPELLO, F.: Performance Comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors. In: SPAA, 2003, **Proceedings.** San Diego, CA, USA: ACM, 2003. p. 118-127.

KUHN, B., PETERSEN, P., O'TOOLE, E.: OpenMP versus threading in C/C++. **Concurrency: Practice and Experience,** [S.l.], vol. nº. p. 1165-1176. 2000.

LEE, K. M.; SONG, T. H., YOON, S-H.; KNOW, K-H.; JEON, J-W.: OpenMP Parallel Programming using Dual-Core Embedded Systems. In: 11th ICCAS, Oct. 2011, **Proceedings.** Gyeonggi-do: IEEE, 2011. p. 762-766.

LEISERSON, C. E.: The Cilk++ Concurrency Platform. **The Journal of Supercomputing**, [S.l.], vol. 51, nº 3, p. 244-257. March, 2010.

LEOPOLD, C.; SÜSS, M.: Observations on MPI-2 Support for Hybrid Master/Slave Applications in Dynamic and Heterogeneous Environments. In: 13th EUROPEAN PVM/MPI USER'S GROUP CONFERENCE ON RECENT ADVANCES IN PARALLEL VIRTUAL MACHINE AND MESSAGE PASSING INTERFACE, 2006, **Proceedings.** Berlin, Heidelberg: Springer-Verlag, 2006. p. 285-292.

LEVERICH, J.; MONCHIERO, M.; TALWAR, V.; RANGANATHAN, P; KOZYRAKIS, C.: Power Management of Datacenter Workloads using Per-Core Power Gating. **IEEE Computer Architecture Letters**, [S.l.], vol. 8, nº 2, p. 48-51. 2009.

LI S.; AHN, J. H.; STRONG, R. D.; BROCKMAN, J. B.; TULLSEN, D. M.; JOUPPI, N. P.: The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing. **ACM Transactions on Architecture and Code Optimization**, New York, NY, USA, vol. 10, nº 1, Article nº 5, April 2013.

LIVELY, C.; WU, X.; TAYLOR, V.; MOORE, S.; CHANG, H-C.; CAMERON K.: Energy and Performance Characteristics of Different Parallel Implementations of Scientific Applications on Multicore Systems. **The International Journal of High Performance Computing Applications**. [S.l.], vol 25 nº 3, p. 342-350. August 2011.

LORENZON, A. F.; ROSSI, F. D.; CERA, M. C.: Analyzing the Impact of MPI-2 Dynamic Creation to the Game of Life Problem. In: 13th SYMPOSIUM ON COMPUTER SYSTEMS (WSCAD-SSC), Oct. 2012, **Proceedings.** Petropolis, RJ: IEEE, 2012. p. 133-140.

LORENZON, A. F.; ROSSI, F. D.; CERA, M. C.: **Um Estudo sobre o Desenvolvimento de Aplicações com Criação Dinâmica de Processos em MPI**. 2013. 115 f. Trabalho de Conclusão de Curso (Ciência da Computação) – UNIPAMPA, Alegrete-RS, 2013.

LORENZON, A. F.; ROSSI, F. D.; OLIVEIRA, A. B.: **Analysis of Parallel Programming Interfaces through Determination of Histograms Similarity**. IN IADIS INTERNATIONAL CONFERENCE APPLIED COMPUTING, Nov. 2011, **Proceedings.** Rio de Janeiro, RJ: [s.n], 2011. p. 446-450.

LUK, C-K.; COHN, R.; MUTH, R.; PATIL, H.; KLAUSER, A.; LOWNEY, G.; WALLACE, S.; REDDI, V. J.; HAZELWOOD, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In PLDI, 2005, **Proceedings**. New York, NY, USA: ACM, 2005. p. 190-200.

MAILLARD, N.; CERA, M. C.: **Message-Passing Interface Avançado**. In: 10th Escola Regional de Alto Desempenho – SBC. Capítulo de Livro. 2010.

MÁLLON, D. A.; TABOADA, G. L.; TEIJEIRO, C.; TOURIÑO, J.; FRAGUELA, B. B.; GÓMEZ, A.; DOALLO, R.; MOURIÑO, J. C.: Performance Evaluation of MPI, UPC and OpenMP on Multicore Architecture. In: EUROPVM/MPI, 2009, **Proceedings.** Berin, Heidelberg: Springer-Verlag, 2009. p. 174-184.

MAROWKA, A.: Analytic Comparison of Two Advanced C Language-Based Parallel Programming Models**. In:** ISPDC/HETEROPAR, July 2004, **Proceedings.** [S. l.]: IEEE, 2004. p. 284-291.

MCCRAW, H., TERPSTRA, D., DONGARRA, J., DAVIS, K., MUSSELMAN R.: Beyond the CPU: Hardware Performance Counter Monitoring on Blue Gene/Q**.** In: INTERNATIONAL SUPERCOMPUTING CONFERENCE, 2013, **Proceedings.** [S. l.]: Springer-Verlag, 2013. p. 213-225.

MICHIELSE, P.: Parallel Multigrid using PVM. **Supercomputer**, Amsterdam, vol 10, nº 1-2, p. 63-69, Nov. 1995.

MOLKA, D.; HACKENBERG, D.; SCHONE, R.; MULLER, M.S.: Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In 18th INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, Sept. 2009, **Proceedings.** Raleigh, NC: IEEE. p. 261-270.

NANJEGOWDA, R.; HERNANDEZ, O.; CHAPMAN, B.; JIN, H. H.: **Scalability Evaluation of Barrier Algorithms for OpenMP**. In Lecture Notes in Computer Science, Vol. 5568, pp 42-52, 2009.

OLIVEIRA, A. B.; SCHARCANSKI, J: Vehicle Counting and Trajectory Detection based on Particle Filtering**.** In: XXIII SIBGRAPI, Aug. 2010, **Proceedings**. Gramado, RS, BR: IEEE, 2008. p. 376-383.

PADUA, D. A.: **Unified Parallel C**. In Encyclopedia of Parallel Computing. pp. 2103, Springer, 2011.

PATEL, I.; GILBERT, J.R.: An Empirical Study of the Performance and Productivity of Two parallel Programming Models. In: IEEE IPDPS, April, 2008, **Proceedings.** Miami, FL: IEEE, 2008. p. 1- 7.

PAUDEL, J.; AMARAL, J. N.: Using the Cowichan Problems to Investigate the Programmability of X10 Programming System**.** In: ACM SIGPLAN, 2011, **Proceedings**. New York, NY, USA: ACM, 2011. Article no. 4.

PELETTI, F.; POGGLIALI, A.; BERTOZZI, D.; BENINI, L.; MARCHAL, P.; LOGHI, M.; PONCINO, M.: Energy-Efficient Multiprocessor System-on-Chip for Embedded Computing: Exploring Programming Models and Their Architectural Support. **IEEE Transactions on Computers**, [S.l.], vol.56, no.5, p.606-621, May, 2007.

PORTERFIELD, A. K.; OLIVIEAR, S. L.; BHALACHANDRA, S.; PRINS, J.: Power Measurement and Concurrency Throttling for Energy Reduction in OpenMP Programs. In: IEEE 27th INTERNATIONAL SYMPOSIUM ON PARALLEL & DISTRIBUTED

PROCESSING WORKSHOPS AND PHD FORUM, May, 2013, **Proceedings.** Cambridge, MA: IEEE, 2013. p. 884-891.

PRESS, W. H.; TEUKOLSKY, S. A.; VETTERLING, W. T., FLANNERY, B. P.: **Numerical Recipes 3rd Edition: The Art of Scientific Computing.** Cambridge University Press, 2007.

RAMACHANDRAN, A.; VIENNE, J.; VAN DER WIJNGAART, R.; KOESTERKE, L.; SHARAPOV, I.: Performance Evaluation of NAS Parallel Benchmarks on Intel Xeon Phi. In: 42nd INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Oct. 2013, **Proceedings.** Lyon: IEEE, 2013. p. 736-743.

RAUBER, T.; RUNGER, G.: **Parallel Programming for Multicore and Cluster Systems**. Springer, 2010.

ROBISON, A. D.: **Intel ® Threading Building Blocks (TBB)**. Encyclopedia of Parallel Computing. P. 2029, Springer, 2011.

RONG, G.; VOGT, R.; MAJUNDER, J.; ALAM A.; BURTSCHER, M.; ZILIANG, Z.: Effects of Dynamic Voltage and Frequency Scaling on a K20 GPU. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Oct. 2013, **Proceedings.** Lyon: IEEE, 2013. p. 826-833.

SHAO, Y.S.; BROOKS, D.: Energy Characterization and Instruction-level Energy model for Intel's Xeon Phi Processor. In: IEEE INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, 2013, **Proceedings**. Piscataway, NJ, USA: IEEE, 2013. p. 389-394.

TANENBAUN, A. S.; WOODHUL, A. S.: **Operating Systems: Design and Implementation**. Prentice Hall, 2009.

SILVA J. A. da.; REBELLO, V. E. F.: A Hybrid Fault Tolerance Scheme for EasyGrid MPI Applications. In: 9th INTERNATIONAL WORKSHOP ON MIDDLEWARE FOR GRIDS, CLOUDS AND E-SCIENCE, 2011, **Proceedings.** New York, USA: ACM, 2011. ARTICLE, No. 4.

STAMATAKIS, A.; OTT, M.: **Exploiting Fine-Grained Parallelism in the Phylogenetic Likelihood Function with MPI, Pthreads, and OpenMP: A Performance Study**. In Pattern Recognition in Bioinformatics, Lecture Notes in Computer Science. Vol 5265, pp. 424-435. 2008.

STANISIC, L.; VIDEAU, B.; CRONSIOE, J.; DEGOMME, A.; MARANGOZOVA-MARTIN, V.; LEGRAND, A.; MÉHAUT, J-F.: Performance Analysis of HPC Applications on Low-Power Embedded Platforms**.** In: DATE, March 2013, **Proceedings.** Grenoble, France: IEEE, 2013. p. 18-22.

SULEMAN, M. A.; QURESHI, M. K.; PATT, Y. N.: Feedback-driven Threading: Power-efficient and High-Performance Execution of Multi-threaded Workloads on CMPs. In ASPLOS XIII, 2008, **Proceedings.** New York, NY, USA: ACM, 2008. p. 277-286.

TERUEL, X.; BARTON, C.; DURAN, A.; MARTORELL, X., AYGUADE, E.; UNNIKRISHNAN, P.; ZHANG, G.; SILVEIRA, R.: OpenMP Tasking Analysis for Programmers. In: CONFERENCE OF THE CENTER OF ADVANCED STUDIES ON COLLABORATIVE RESEARCH, 2009, **Proceedings.** New York, USA: ACM, 2009. p. 32-42.

TRISTAM W., BRADSHAW K.: Investigating the Performance and Code Characteristics of Three Parallel Programming Models for C++. In: SOUTHERN AFRICA TELECOMMUNICATION NETWORKS AND APPLICATIONS CONFERENCE (SATNAC), Jul. 2010, **Proceedings**. [S.l.:s.n], 2010.

ZECENA, I.; ZONG, Z.; GE, R.; JIN, T.; CHEN, Z.; QIU, M.: Energy Consumption Analysis of Parallel Sorting Algorithms Running on Multicore Systems. In: INTERNATIONAL GREEN COMPUTING CONFERENCE, June, 2012, **Proceedings.** San Jose, CA: IEEE, 2012. p. 1-6.

ZIMMER, C.; MUELLER, F.: Low Contention Mapping of Real-Time Tasks onto TilePro 64 Core Processors. In: IEEE 18th REAL TIME AND EMBEDDED TECHNOLOGY AND APPLICATIONS SYMPOSIUM, 2012, **Proceedings**. Washington, DC, USA: IEEE, 2012. p. 131-140.

WAHLÉN, N.: **A Comparison of Different Parallel Programming Models for Multicore Processors**. 2010. 51 f. Bachelor of Science Thesis, KTH Information and Communication Technology. Stockholm, Sweden, 2010.

WEAVER, V.; TERPSTRA, D.; MCCRAW, H.; JOHNSON, M.; KASICHAYANULA, K.; RALPH, J.; NELSON, J.; MUCCI, P.; MOHAN, T.; MOORE, S.: PAPI 5: Measuring Power, Energy, and the Cloud**.** In: IEEE INTERNATIONAL SYMPOSIUM ON PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE, April, 2013, **Proceedings.** Austin, TX: IEEE, 2013. p. 124-125.

WEHNER, M.; OLIKER, L.; SHALF, J.: A Real Cloud Computing. **IEEE Spectrum**, [S.l.], vol.46, no. p. 2429, 2009.

WELDEZION, A.Y.; GRANGE, M.; PAMUNUWA, D.; ZHONGHAI LU; JANTSCH, A.; WEERASEKERA, R.; TENHUNEN, H., Scalability of network-on-chip communication architecture for 3-D meshes. In: 3RD ACM/IEEE INTERNATIONAL SYMPOSIUM ON NETWORKS-ON-CHIP, May, 2009, **Proceedings.** San Diego, CA: IEEE, 2009. p.114,123.

WILSON, G., V.; BAL, H. E.: Using the Cowichan Problems to Assess the Usability of Orca. **IEEE Parallel and Distributed Technology: Systems and Applications**, [S.l.], vol. 4, nº 3, p. 36-44, Fall 1996.

YAVITS, L.; MORAD, A.; GINOSAR, R.: The Effect of Communication and Synchronization on Amdahl's Law in Multicore Systems. **Elsevier Parallel Computing: Systems and Applications**, [S.l.], vol. 40, no. 1, p. 1-16, January, 2013.

APÊNDICE A – ESTRATÉGIA DE PARALELIZAÇÃO

A.1 Calculo do Pi

A Figura A.1 apresenta a implementação sequencial da função `calculaPi`. A implementação paralela com o **OpenMP** se deu através da inserção da diretiva:

```
#pragma omp parallel for schedule() private(i) reduce(+:pi)
```

entre as linhas 2 e 3. Os valores contidos interno a diretiva *schedule* definem como as iterações serão atribuídas entre as *threads* na região paralela. A definição da variável *i* como privada é necessária para que cada *thread* possa controlar suas próprias iterações sem a interferência das demais. Por fim, a diretiva *reduce(+:pi),* faz com que, ao final da região paralela, a *thread* mestre faça a soma dos valores da variável *pi* calculados em cada *thread*.

Já em **Pthreads**, a paralelização se deu através da divisão do número total de pontos (Num_Pontos) entre o número de *threads* de forma estática. Por exemplo, considerando que Num_Pontos é igual a 100 e existem 4 *threads*, a *thread0* computará as iterações 0 até 24; a *thread1*, as iterações 25 até 49 e assim por diante, sempre em paralelo. Para fazer a redução, foi utilizado as variáveis do tipo *mutex* ao final do laço for.

A principal diferença da paralelização com Pthreads para **MPI-1** é a inserção da função MPI_Reduce() ao final do laço for. Tal função tem o mesmo objetivo que a diretiva *reduce* do OpenMP. Por fim, **MPI-2** diferencia-se de MPI1-1 somente pela criação dinâmica de processos.

Figura A.1 Algoritmo Sequencial Calculo do Pi

```
void calculaPi(){
1.  int Num_Pontos;
2.  double pi=0;
3.  for(int i=0; i<Num_Pontos; i++){
4.      pi += 4/(4*i+1);
5.      pi -= 4/(4*i+3);
6.  }
}
```

A.2 Conjunto de Mandelbrot

A Figura A.2 apresenta a implementação sequencial da função `calculaMandelbrot`. A paralelização se deu através da divisão do número de iterações do laço for da linha 2 entre as *threads*. Para tal, a implementação paralela com o **OpenMP** se deu através da inserção da diretiva:

```
#pragma omp parallel for schedule() private(i,j)
```

em que as iterações serão distribuídas entre as *threads* conforme o *schedule* utilizado. Ademais, as variáveis privadas *i* e *j* são necessárias para que uma *thread* não interfira na computação da outra.

Por se tratar de uma aplicação com grande desbalanceamento de carga, em **Pthreads**, o número de iterações do laço for da linha 2 foi dividido entre as *threads* buscando o melhor balanceamento de carga possível.

A paralelização com **MPI-1** deu-se da mesma forma que em Pthreads, no entanto, com a adição da comunicação entre os processos ao fim do laço *for* da linha 6. Nesta comunicação, os processos que não correspondem ao processo com identificador *0* enviam seus dados computados (através da função `MPI_Send`) para o processo 0 (que recebe os resultados através do `MPI_Recv`).

Novamente, a principal diferença da implementação em MPI-2 para MPI-1 continua sendo a adição da criação dinâmica de processos antes da computação da função calculaMandelbrot.

Figura A.2 Algoritmo Sequencial Conjunto de Mandelbrot

```
void calculaMandelbrot(){
1.  int ALTURA, LARGURA;
2.  for(int i=0; i<ALTURA; i++){
3.      for(int j=0; j<LARGURA; j++){
4.          calculaPonto(i,j);
5.      }
6.  }
}
```

A.3 Série Harmônica

A Figura A.3 apresenta a implementação sequencial da função `serieHarmonica`. A paralelização se deu através da divisão do número de iterações do laço for da linha 2 entre as *threads*. Entretanto, ao final do mesmo *for* (linha6), é necessário fazer a redução dos valores internos ao vetor, isto porque cada *thread* computará valores separados das posições do vetor. Para tal, a implementação paralela com o **OpenMP** se deu através da inserção da diretiva:

```
#pragma omp parallel for schedule() private(i,j)
```

antes do laço for da linha 2. Devido ao OpenMP não possuir operações do tipo *reduce* para vetores, foi adicionada uma seção critica (*#pragma omp critical*) após o fim do laço principal (linha 6) para cada *thread* atualizar o valor final do vetor *serie*.

Na implementação com **Pthreads**, as iterações do laço for da linha 2 foram divididas entre as *threads* visando o melhor balanceamento de carga. Para realizar o *reduce* após o final da computação paralela, na linha 6, utilizou-se variáveis do tipo *mutex* para possibilitar que cada *thread* atualize o valor final do vetor *serie*.

Com **MPI-1**, a divisão da carga de trabalho deu-se da mesma forma que em Pthreads. No entanto, MPI fornece redução de valores em vetores através da função MPI_Reduce. Portanto, ela foi utilizada para a operação de *reduce*.

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.

Figura A.3 Algoritmo Sequencial Série Harmônica

```
void serieHarmonica(){
1.  long unsigned int n, d, serie[d];
2.  for(long unsigned int i=1; i<n; i++){
3.      for(long unsigned int j=0; j<d; j++){
4.          Computa serie[j];
5.      }
6.  }
}
```

A.4 Dijkstra

A Figura A.4 apresenta a implementação sequencial da função `calculaDijkstra`. A paralelização se deu através da divisão do número de vértices a serem computados (laço *for* da linha 2). Para tal, a implementação paralela com o **OpenMP** se deu através da inserção da diretiva `#pragma omp parallel for schedule() private(i)` antes do laço for da linha 2. Os valores contidos interno a diretiva *schedule* definem como as iterações serão atribuídas entre as *threads* na região paralela. A definição da variável *i* como privada é necessária para que cada *thread* possa controlar e computar suas próprias iterações sem a interferência das demais.

A paralelização em **Pthreads** deu-se através da divisão do número de vértices (numVertices) a serem procurados pelo número de *threads*. Por ex. considerando que o numVertices é igual a 100 e existem 4 *threads*, a *thread0* computará as iterações 0 até 24; a *thread1*, as iterações 25 até 49 e assim por diante, sempre em paralelo.

A distribuição da carga de trabalho em **MPI-1** deu-se da mesma forma que em Pthreads. No entanto, ao final do laço for da linha 2, faz-se necessário "juntar" o resultado obtido pela computação de cada processo. Para tal, cada processo diferente de 0 envia seus resultados para o processo 0 através das diretivas MPI_Send/MPI_Recv.

Por fim, na implementação em **MPI-2**, adicionou-se a criação dinâmica de processos com relação a paralelização em MPI-1.

Figura A.4 Algoritmo Sequencial Dijkstra

```
void calculaDijkstra(){
1.  int numVertices, matrizVertices[][];
2.  for(int i=0; i<numVertices; i++){
3.      Computa distância de i para os demais //
        Vertices da matrizVetices[][];
4.  }
}
```

A.5 Similaridade entre Histogramas

A Figura A.5 apresenta a implementação sequencial da função `calculaSimilaridade`. A paralelização se deu através da divisão das iterações do laço *for* da linha 2 que percorre as linhas da matriz. Para tal, a implementação paralela com o **OpenMP** se deu através da inserção da diretiva `#pragma omp parallel for schedule() private(i,j)` antes do laço *for* da linha 2. Assim, as iterações serão divididas entre as *threads* conforme o *schedule* definido. As variáveis privadas *i* e *j* são necessárias para que uma *thread* não interfira na computação das demais.

A paralelização envolvendo **Pthreads** partiu da divisão da altura (ALTURA) da imagem entre o número de *threads*. Por ex. considerando que a ALTURA é igual a 1024 e existem 4 *threads*, a *thread0* computará as iterações 0 até 255; a *thread1*, as iterações 256 até 511 e assim por diante, sempre em paralelo.

Embora a divisão da carga de trabalho em **MPI-1** seja igual a aplicada em Pthreads, existem duas comunicações adicionais em MPI-1. A primeira comunicação consiste em o processo *P0* enviar para os demais processos as porções da imagem (*matriz*) alvo em que cada processo irá computar e o histograma da imagem a ser procurado (*histo*). No momento em que os processos finalizam a computação da sua parcela de trabalho, ambos devem retornar ao processo *P0* (que também está computando) o resultado de sua computação. Tais comunicações foram implementadas com as funções `MPI_Send` e `MPI_Recv`.

Na paralelização com **MPI-2**, adicionou-se a criação dinâmica de processos comparada a paralelização realizada em MPI-1.

Figura A.5 Algoritmo Sequencial Similaridade entre Histogramas

```
void calculaSimilaridade(){
1.  float alvo[ALTURA][LARGURA], histo[];
2.  for(int i=0; i<ALTURA; i++){
3.      for(int j=0; j<LARGURA; j++){
4.          calculo da similaridade entre alvo[i][j] e histo[];
5.      }
6.  }
}
```

A.6 Decomposição-LU

A Figura A.6 apresenta a implementação sequencial da função `decomposicaoLU`. A paralelização se deu através da divisão da computação das linhas e colunas (laço *for* das linhas 5 e 7) entre as *threads*/processos. O laço *for* da linha 3, responsável por calcular o elemento da diagonal não será paralelizado pois ele possui dependência de dados na escrita dos valores na matriz. Portanto, durante a paralelização, esta etapa será computada por somente uma *thread/*processo.

A implementação paralela com o **OpenMP** se deu através da inserção das diretivas: `#pragma omp parallel private(i, k)` antes do laço *for* da linha 2 – assim as *threads* serão criadas uma única vez e não a cada iteração (*i*); `#pragma omp master` antes do laço *for* da linha 3 – para garantir que a computação da calcula diagonal será realizada por uma única *thread*, neste caso, a *thread master*, enquanto que as demais *threads* aguardam em uma barreira implícita; e `#pragma omp for schedule()` antes do laço *for* das linhas 5 e 7 – para dividir a carga de trabalho entre as *threads* conforme o *schedule* definido.

Com **Pthreads**, todas as *threads* executam a função decomposicaoLU. Nela, somente a *thread* 0 computa o laço for da linha 3, enquanto que as demais aguardam em uma barreira antes da linha 5. Isto porque a computação só poderá prosseguir com a finalização da computação do for da linha 3. A divisão da carga de trabalho entre as *threads* nos laços *for* das linhas 5 e 7 se deu através da divisão do numero de iterações pelo número de *threads*, buscando sempre o melhor balanceamento da carga de trabalho.

Com **MPI-1**, foi utilizada divisão da carga de trabalho similar a da utilizada em Pthreads. No entanto, a principal diferença está na necessidade de comunicação para atualizar os dados

Figura A.6 Algoritmo Sequencial Decomposição-LU

```
void decomposicaoLU(){
1.  float A[N][N];
2.  for(int i=1; i<N; i++){
3.      for(int k=0; k<i; k++)
4.          calcula diagonal A[i][i];
5.      for(int k=i+1; k<N; k++)
6.          calcula linha A[i][k];
7.      for(int k=i+1; k<N; k++)
8.          calcula coluna A[k][i];
9.  }
}
```

computados no laço for das linhas 5 e 7. Assim, após a computação do laço for da linha 7, todos os processos se comunicam, a fim de que cada processo mantenha atualizada a sua matriz. Para esta comunicação, foi utilizada a função de comunicação coletiva `MPI_Bcast`.

Já em **MPI-2**, adicionou-se a criação dinâmica de processos, e devido a isso, a comunicação entre os processos para atualização dos dados após a computação do laço for da linha 7 foi realizada através de funções de comunicação ponto a ponto `MPI_Send` e `MPI_Irecv`.

Ademais, esta aplicação foi paralelizada seguindo a proposta apresentada por David Bouman em (BOUMAN, 1995).

A.7 Método de Jacobi

A Figura A.7 apresenta a implementação sequencial da função `jacobi`. A paralelização se deu através da divisão das iterações do laço for da linha 4 entre as *threads/*processos. A dependência de dados existente devido a computação em uma única matriz (A) foi resolvida adicionando uma matriz auxiliar (Anew), que conterá o valor correto ao final de cada computação. Para retornar os dados atualizados para a matriz original A, no final da computação do laço for, na linha 7, é realizada a troca de dados entre as duas matrizes (*swap(A, Anew)*), a qual deve ser executada por um único processo.

A paralelização com **OpenMP** deu-se através do uso da diretiva `#pragma omp parallel private(i, j)` 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 7.

Já com **Pthreads**, as iterações do laço for da linha 4 foram divididas entre as *threads* buscando obter o melhor balanceamento da carga de trabalho. Por ex. considerando que o N é igual a 100 e existem 4 *threads*, a *thread0* computará as iterações 0 até 24; a *thread1*, as iterações 25 até 49 e assim por diante, sempre em paralelo.

Com **MPI-1** a distribuição da carga de trabalho entre os processos seguiu o mesmo padrão que Pthreads. No entanto, inicialmente o processo *P0* envia a parcela de dados que cada processo irá computar através das funções `MPI_Send/MPI_Irecv` antes da linha 3. 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 MPI-1. No entanto, ao invés da utilização do MPI_Bcast para atualizar os dados entre todos

Figura A.7 Algoritmo Sequencial Método de Jacobi

```
void jacobi(){
1.  float A[N][M], Anew[N][M];
2.  int TOTAL_ITERACOES;
3.  for(int iter=0; iter<TOTAL_ITERACOES; iter++){
4.      for(int i=0; i<N; i++)
5.          for(int j=0; j<M; j++)
6.              calcula A[i][j] e armazena em Anew[i][j];
7.      swap(A, Anew);
8.  }
}
```

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.8 Multiplicação de Matrizes

A Figura A.8 apresenta a implementação sequencial da função `multiplicaMatriz`. A paralelização se deu através da divisão das iterações do laço *for* da linha 2 que percorre as linhas da matriz. Para tal, a implementação paralela com o **OpenMP** se deu através da inserção da diretiva:

```
#pragma omp parallel for schedule() private(i, j, k)
```

antes do laço *for* da linha 2. Assim, as iterações serão divididas entre as *threads* conforme o *schedule* definido. As variáveis privadas *i* e *j* são necessárias para que uma *thread* não interfira na computação das demais.

A paralelização envolvendo **Pthreads** partiu da divisão da altura (N) da matriz entre o número de *threads*. Por ex. considerando que N é igual a 1024 e existem 4 *threads*, a *thread0* computará as iterações 0 até 255; a *thread1*, as iterações 256 até 511 e assim por diante, sempre em paralelo.

Embora a divisão da carga de trabalho em **MPI-1** seja igual à aplicada em Pthreads, existem duas comunicações adicionais em MPI-1. A primeira comunicação consiste em o processo *P0* enviar para os demais processos 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 *P0* (que também está computando) o resultado de sua computação. Tais comunicações foram implementadas com 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`.

Na implementação com **MPI-2**, adicionou-se a criação dinâmica de processos à implementação realizada com MPI-1.

Figura A.8 Algoritmo Sequencial Multiplicação de Matriz

```
void multiplicaMatriz(){
1.  float A[N][N], B[N][N], C[N][N];
2.  for(int i=0; i<N; i++)
3.      for(int j=0; j<N; j++)
4.          for(int k=0; k<N; k++)
5.              calcula A[i][j];
}
```

A.9 Jogo da Vida

A Figura A.9 apresenta a implementação sequencial da função `jogodavida`. A paralelização se deu através da divisão das iterações do laço *for* da linha 4 entre as *threads/*processos. A dependência de dados existente devido a computação em uma única matriz (*Soc*) foi resolvida adicionando uma matriz auxiliar (*SocNew*), que conterá o valor correto ao final de cada computação. Para retornar os dados atualizados para a matriz original *SocNew*, no final da computação do laço for, na linha 7, é realizada a troca de dados entre as duas matrizes (*swap(Soc, SocNew)*), a qual deve ser executada por um único processo.

A paralelização com **OpenMP** deu-se através do uso da diretiva `#pragma omp parallel private(i, j)` 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 7.

Já com **Pthreads**, as iterações do laço for da linha 4 foram divididas entre as *threads* buscando obter o melhor balanceamento da carga de trabalho. Por ex. considerando que o N é igual a 100 e existem 4 *threads*, a *thread0* computará as iterações 0 até 24; a *thread1*, as iterações 25 até 49 e assim por diante, sempre em paralelo. A função *swap* foi executada somente pela *thread0*, enquanto que as demais ficam aguardando em uma barreira.

Com **MPI-1** a distribuição da carga de trabalho entre os processos seguiu o mesmo padrão que Pthreads. No entanto, inicialmente o processo *P0* envia a parcela de dados que cada processo irá computar através das funções `MPI_Send/MPI_Irecv` antes da linha 3. 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

Figura A.9 Algoritmo Sequencial Jogo da Vida

```
void jogodavida(){
1.  int Soc[N][M], SocNew[N][M];
2.  int Total_Geracoes;
3.  for(int iter=0; iter<Total_Geracoes; iter++)
4.      for(int i=0; i<N; i++)
5.          for(int j=0; j<M; j++)
6.              evolui(Soc[i][j], SocNew[i][j]);
7.      swap(Soc, SocNew);
8.  }
}
```

MPI-1. No entanto, 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.10 Gram-Schmidt

A Figura A.10 apresenta a implementação sequencial da função `gramSchmidt`. A paralelização se deu através da divisão das iterações dos laços *for* das linhas 3, 6, 8 e 10 entre as *threads*/processos. Nos laços *for* das linhas 3 e 8 nota-se que existe a alteração da variável *tmp*, que será utilizada nos laços *for* seguintes. Portanto, ao final destes laços *for* faz-se necessário atualizar *tmp* em todas as *threads*/processos através da operação de redução. Ademais, para evitar problemas de consistência dos dados, a execução de cada laço *for* só pode começar após a execução do laço *for* anterior estiver terminada. Portanto, faz-se necessário a utilização de barreiras temporais ao fim destes laços *for*.

A paralelização com **OpenMP** se deu através do uso da diretiva `#pragma omp parallel for private(i) reduction(+:tmp)` no laço *for* da linha 3 e 8. Nos laços *for* das linhas 6 e 10 foi inserida mesma diretiva sem a operação de redução. No OpenMP, já existe uma barreira temporal ao final de cada laço paralelizado.

Já com **Pthreads,** as iterações destes laços foram divididas entre as *threads* buscando obter o melhor balanceamento da carga de trabalho. Por exemplo, considerando que N é igual a 100 e existem 4 *threads*, a *threads0* computará as iterações 0 até 24; a *thread1*, as iterações 25 até 49; e assim por diante, sempre em paralelo. Como barreira temporal, foi utilizada a função *pthread_barrier_wait* e o *mutex* foi utilizado para fazer as operações de redução na variável *tmp*.

Figura A.10 Algoritmo Sequencial Gram-Schmidt

```
void gramSchmidt(){
1.  float Q[N][M], tmp=0;
2.  for(k=0; k<M; k++){
3.      for(i=0; i<N; i++)
4.          tmp += Q[i][k]*Q[i][k];
5.      tmp = sqrt(tmp);
6.      for(i=0; i<N; i++)
7.          Q[i][k]/=tmp;
8.      for(j=k+1; j<M; j++){
9.          tmp=0;
8.          for(i=0; i<N; i++)
9.              tmp += Q[i][k]*Q[i][j];
10.         for(i=0; i<N; i++)
11.             Q[i][j] -= tmp*Q[i][k];
12.     }
13. }
}
```

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 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 no MPI-2.

A.11 Ordenação Par-Impar

A Figura A.11 apresenta a implementação sequencial da função `parImpar`. A paralelização se deu através da divisão das iterações dos laços *for* das linhas 3 e 5 entre as *threads*/processos. Entretanto, de acordo com a característica do algoritmo, a computação do laço *for* da linha 5 só poderá iniciar após a finalização por completo da computação do *for* da linha 4. Para tanto, é necessário inserir uma barreira antes da linha 3. O mesmo é necessário para não iniciar a computação do laço *for* da linha 3 antes da conclusão do *for* da linha 5.

Com **OpenMP** a paralelização ocorreu com a inserção das seguintes diretivas: `#pragma omp parallel private(i, j)` antes da linha 2 – para criar as *threads* uma única vez; e `#pragma omp for schedule()` antes dos laços *for* das linhas 3 e 5. Ao final de cada laço paralelo, as *threads* sincronizam, portanto não é necessária a inserção de diretivas explicitas de sincronização.

Já em **Pthreads**, as iterações dos laços *for* das linhas 3 e 5 foram divididas entre as *threads* buscando obter o melhor balanceamento da carga de trabalho. Para realizar a barreira entre as *threads*, foi utilizada a função *pthread_barrier_wait()*, antes do laço *for* das linhas 3 e 5. Assim, a computação destes laços sempre irá iniciar de forma sincronizada.

A paralelização com **MPI-1** deu-se da mesma forma que com Pthreads. No entanto, inicialmente *P0* envia para os demais processos (`MPI_Send/MPI_Irecv`) a parcela de dados que cada um irá computar conforme atribuição da carga de trabalho. Após a computação do laço *for* da linha 3 e 5 inseriu-se 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 a paralelização com MPI-1.

Figura A.11 Algoritmo Sequencial Ordenação Par-Impar

```
void parImpar(){
1.  int vetor[N];
2.  for(int i=0; i<N/2; i++){
3.      for(int j=1; j<N-1; j=j+2)
4.          ordenaFaseImpar();
5.      for(int j=0; j<N-1; j=j+2)
6.          ordenaFasePar();
7.  }
}
```

A.12 Turing Ring

A Figura A.12 apresenta a implementação sequencial da função `calculaTuring`. A paralelização se deu através da divisão das iterações dos laços *for* das linhas 4 e 7 entre as *threads*/processos. Entretanto, de acordo com a característica do algoritmo, a computação do laço *for* da linha 7 só poderá iniciar após a finalização por completo da computação do *for* da linha 4. Para tanto, é necessário inserir uma barreira antes da linha 7. O mesmo é necessário para não iniciar a computação do laço *for* da linha 4 antes da conclusão do *for* da linha 7.

Com **OpenMP** a paralelização ocorreu com a inserção das seguintes diretivas: `#pragma omp parallel private(iter, i, j)` antes da linha 3 – para criar as *threads* uma única vez; e `#pragma omp for schedule()` antes dos laços *for* das linhas 4 e 7. Ao final de cada laço paralelo, as *threads* sincronizam, portanto não é necessária a inserção de diretivas explicitas de sincronização.

Já em **Pthreads**, as iterações dos laços *for* das linhas 4 e 7 foram divididas entre as *threads* buscando obter o melhor balanceamento da carga de trabalho. Para realizar a barreira entre as *threads*, foi utilizada a função *pthread_barrier_wait()*, antes do laço *for* das linhas 4 e 7. Assim, a computação destes laços sempre irá iniciar de forma sincronizada.

A paralelização com **MPI-1** deu-se da mesma forma que com Pthreads. No entanto, existe a necessidade de atualizar a parcela de dados computados em cada processo. Assim, após a computação do laço *for* da linha 7 inseriu-se funções de comunicação (`MPI_Send`/`MPI_Irecv`) entre os processos. Já em **MPI-2**, adicionou-se a criação dinâmica de processos e a divisão da carga de trabalho e comunicação continuou a mesma aplicada no MPI-1.

Figura A.12 Algoritmo Sequencial *Turing Ring*

```
void calculaTuring(){
1.  float TR[N][M];
2.  int TOTAL_ITERACOES;
3.  for(int iter=0; iter<TOTAL_ITERACOES; iter++){
4.      for(int i=0; i<N; i++)
5.          for(int j=0; j<M; j++)
6.              iterage posicao TR[i][j];
7.      for(int i=0; i<N; i++)
8.          for(int j=0; j<M; j++)
9.              ajusta sociedade TR;
10.  }
}
```

APÊNDICE B – RESULTADOS OBTIDOS EM CADA APLICAÇÃO

B.1 Cálculo do Número Pi

Figura B.1 – Resultados Cálculo do Número Pi (escala logarítmica)

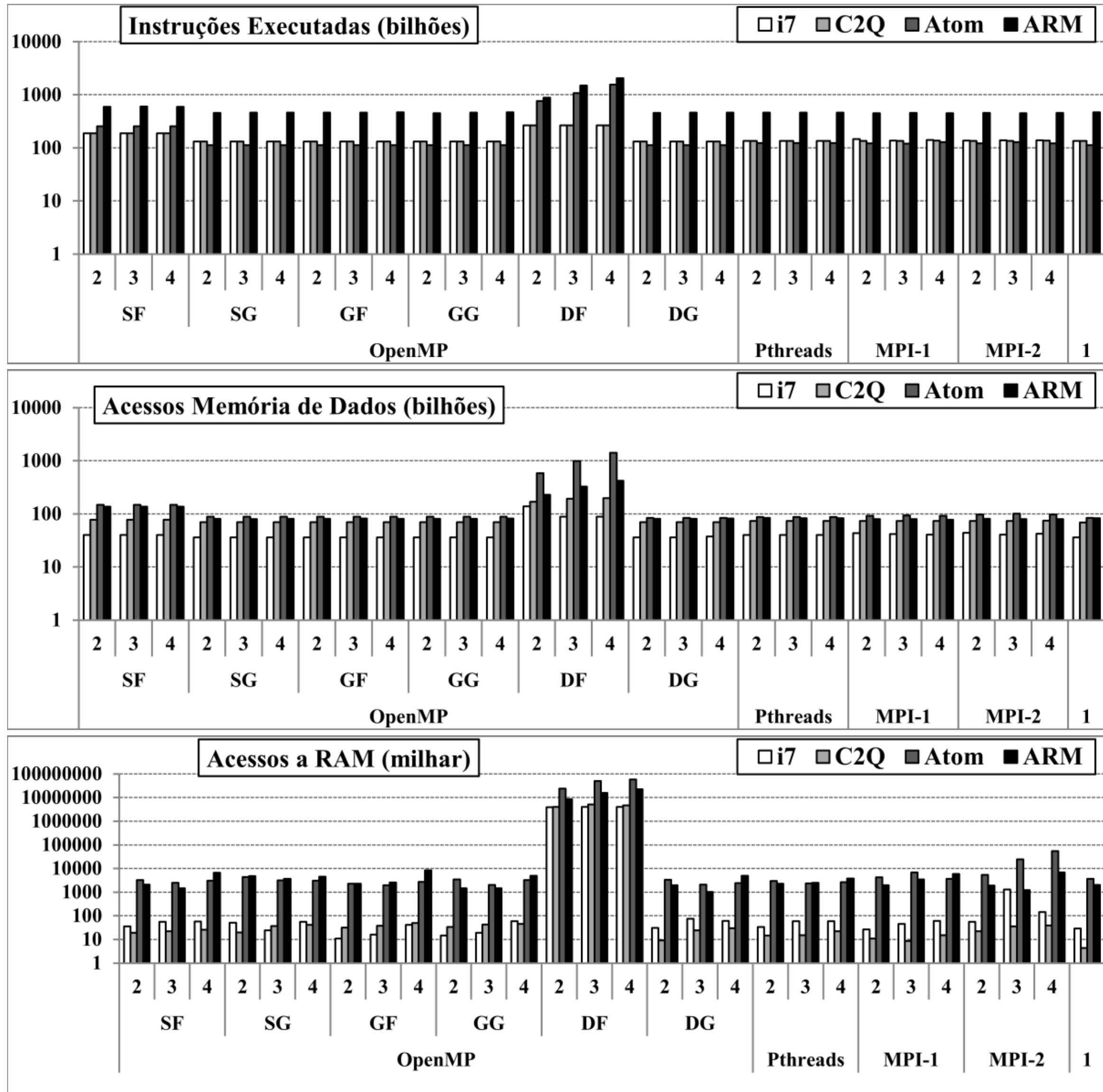
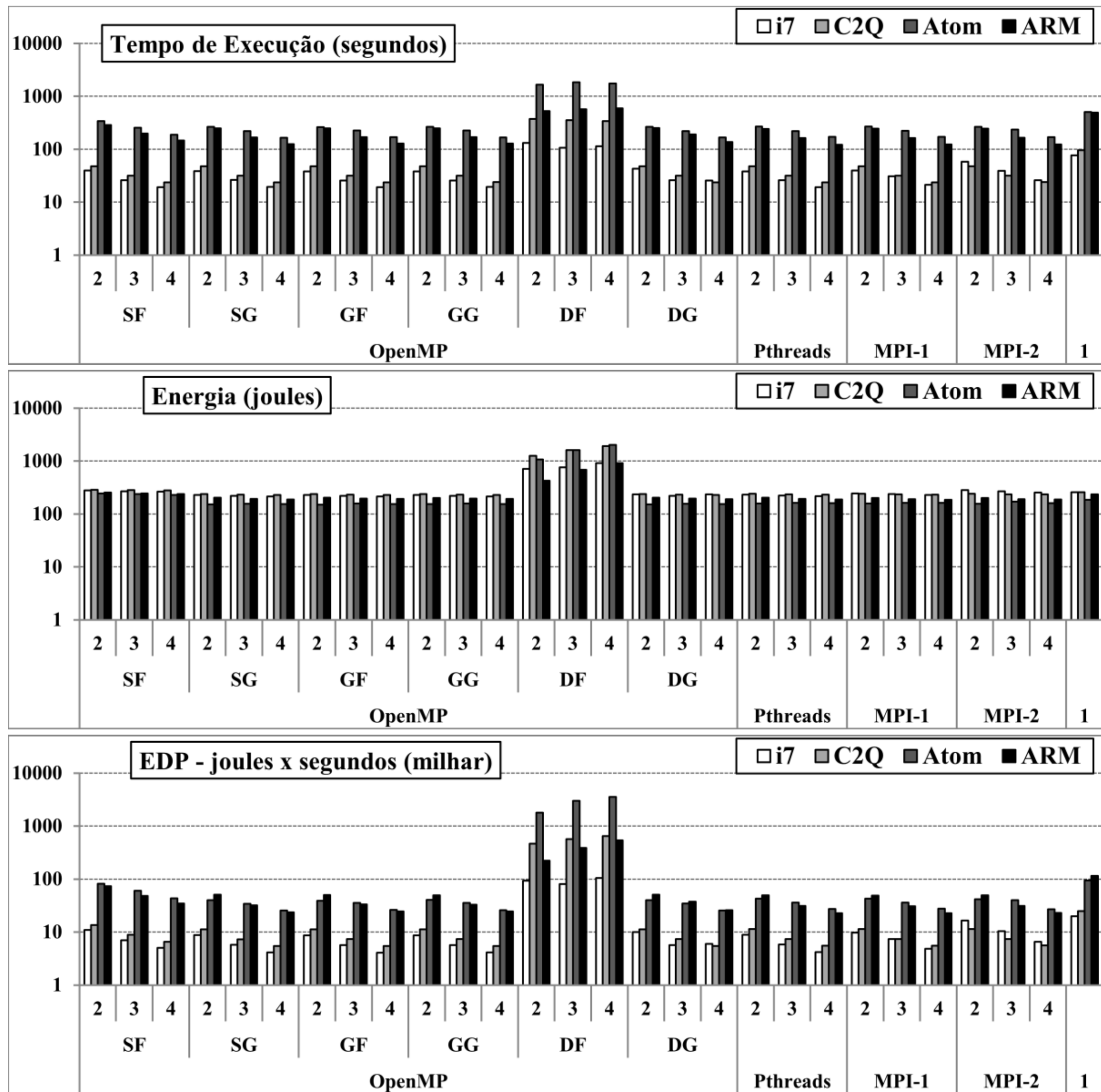


Figura B.2 – Resultados Cálculo do Número Pi (escala logarítmica) - Continuação



B.2 Conjunto de *Mandelbrot*

Figura B.3 – Resultados Conjunto de *Mandelbrot*

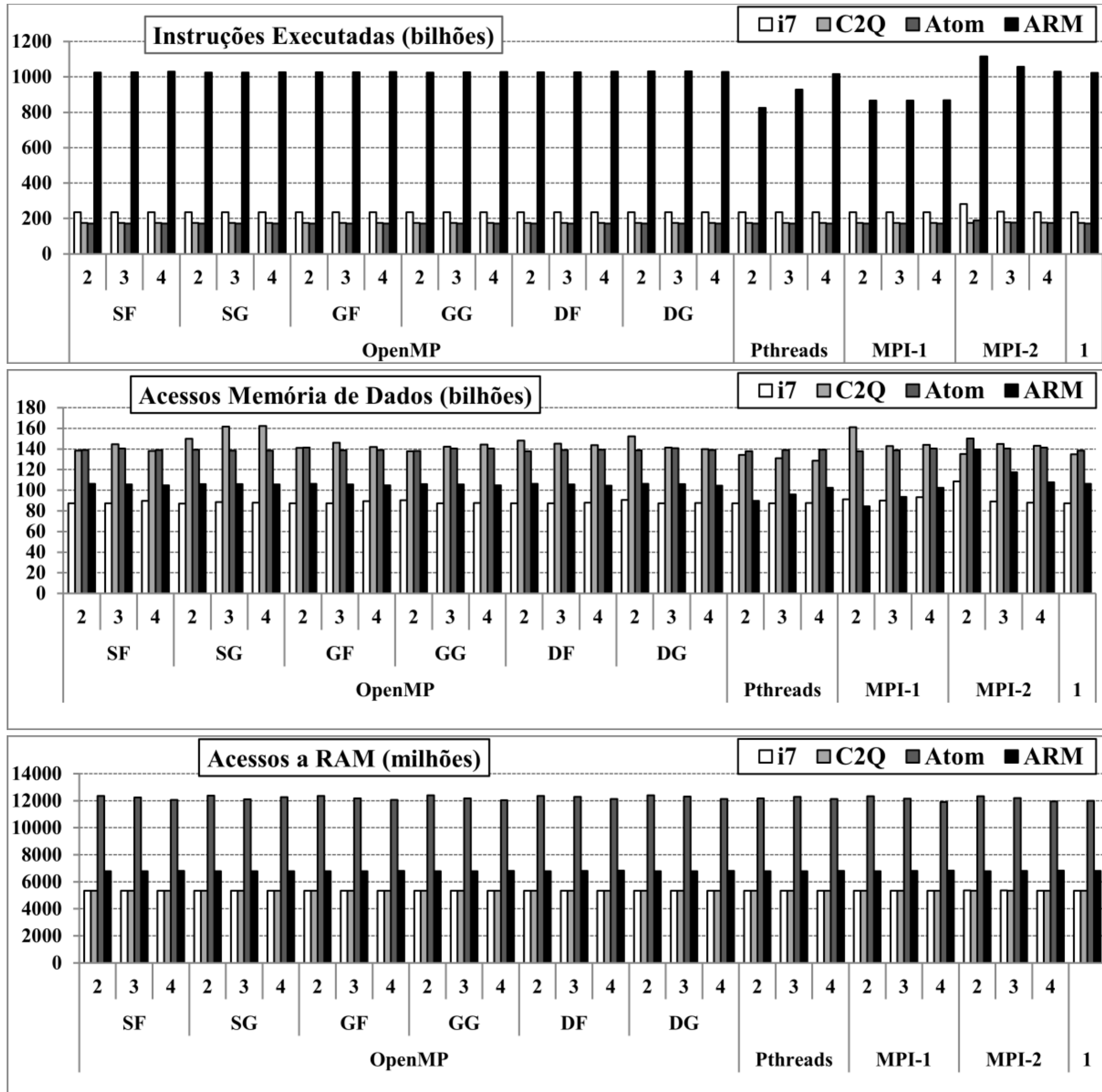
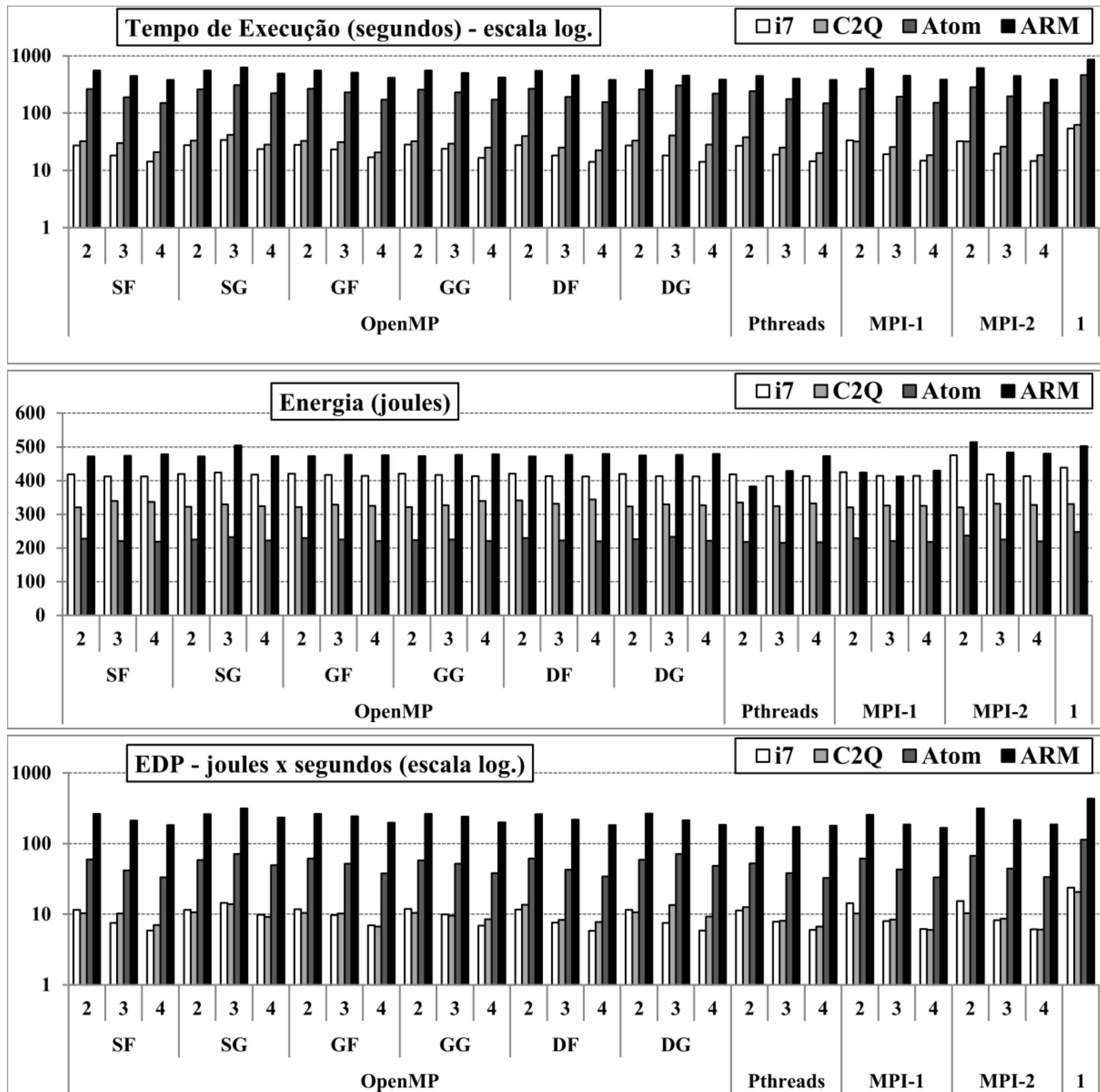


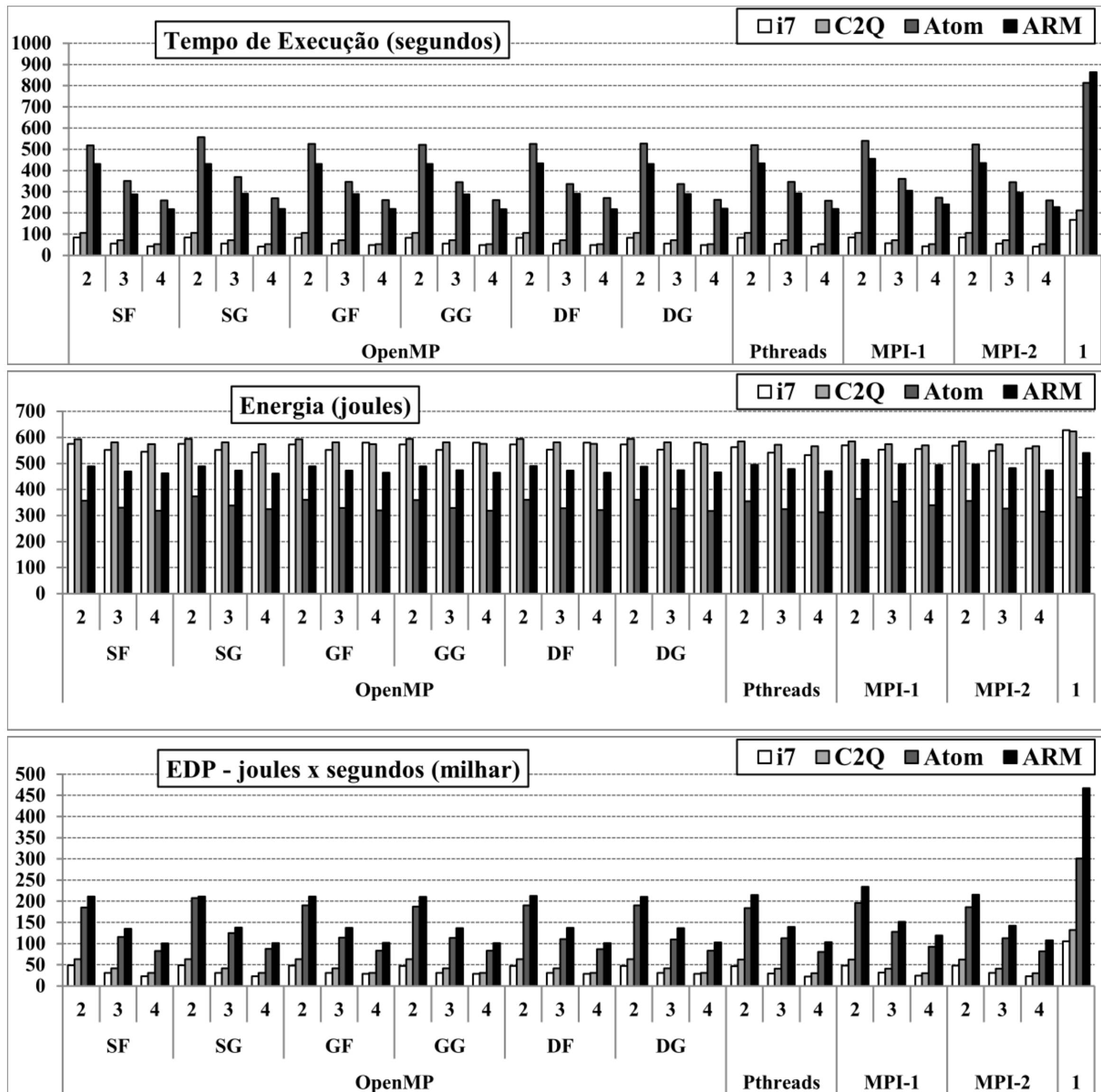
Figura B.4 – Resultados Conjunto de *Mandelbrot* - Continuação



B.3 Série Harmônica

Figura B.5 – Resultados Série Harmônica

Figura B.6 – Resultados Série Harmônica - Continuação



B.4 *Dijkstra*

Figura B.7 – Resultados *Dijkstra*

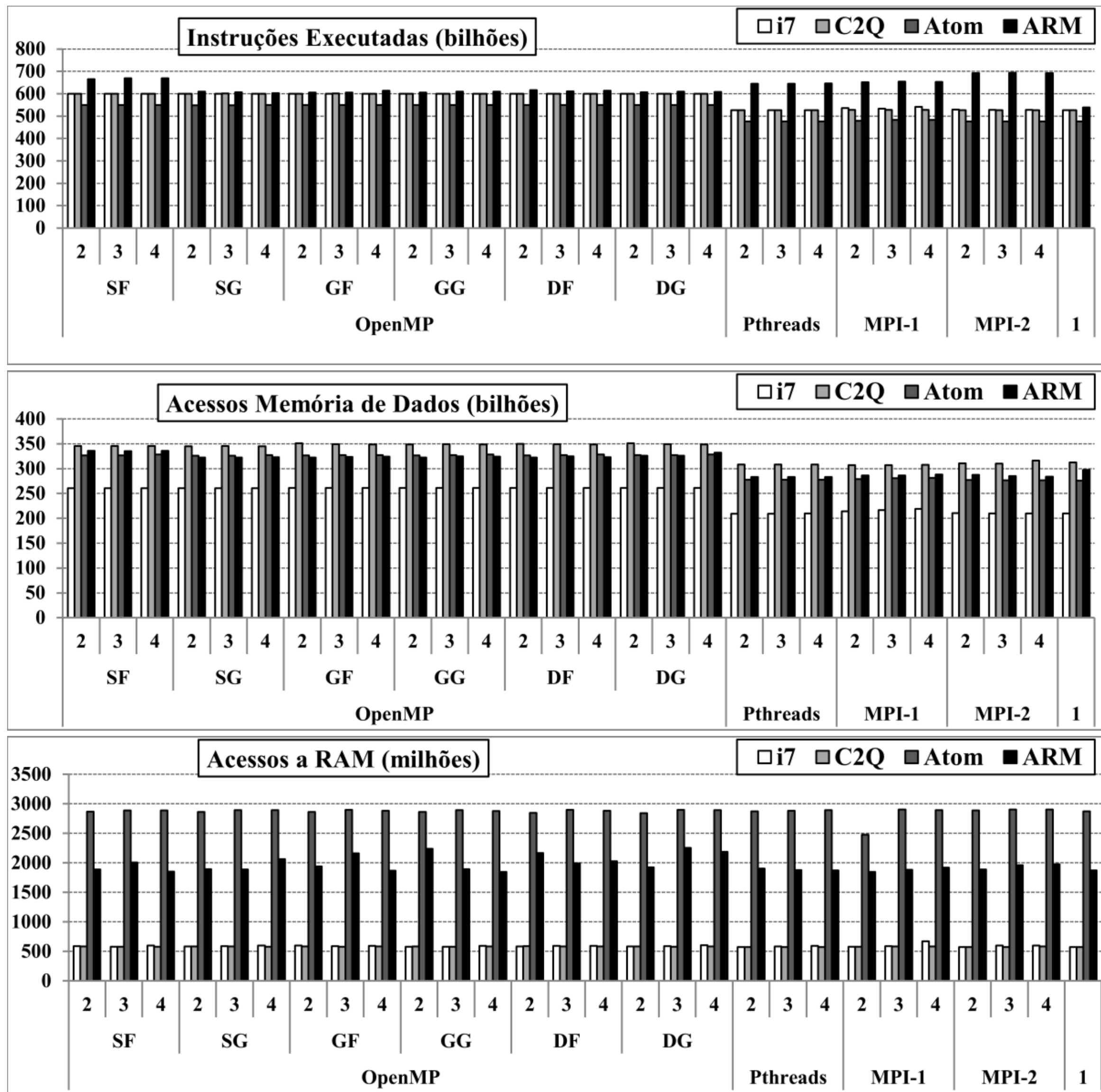
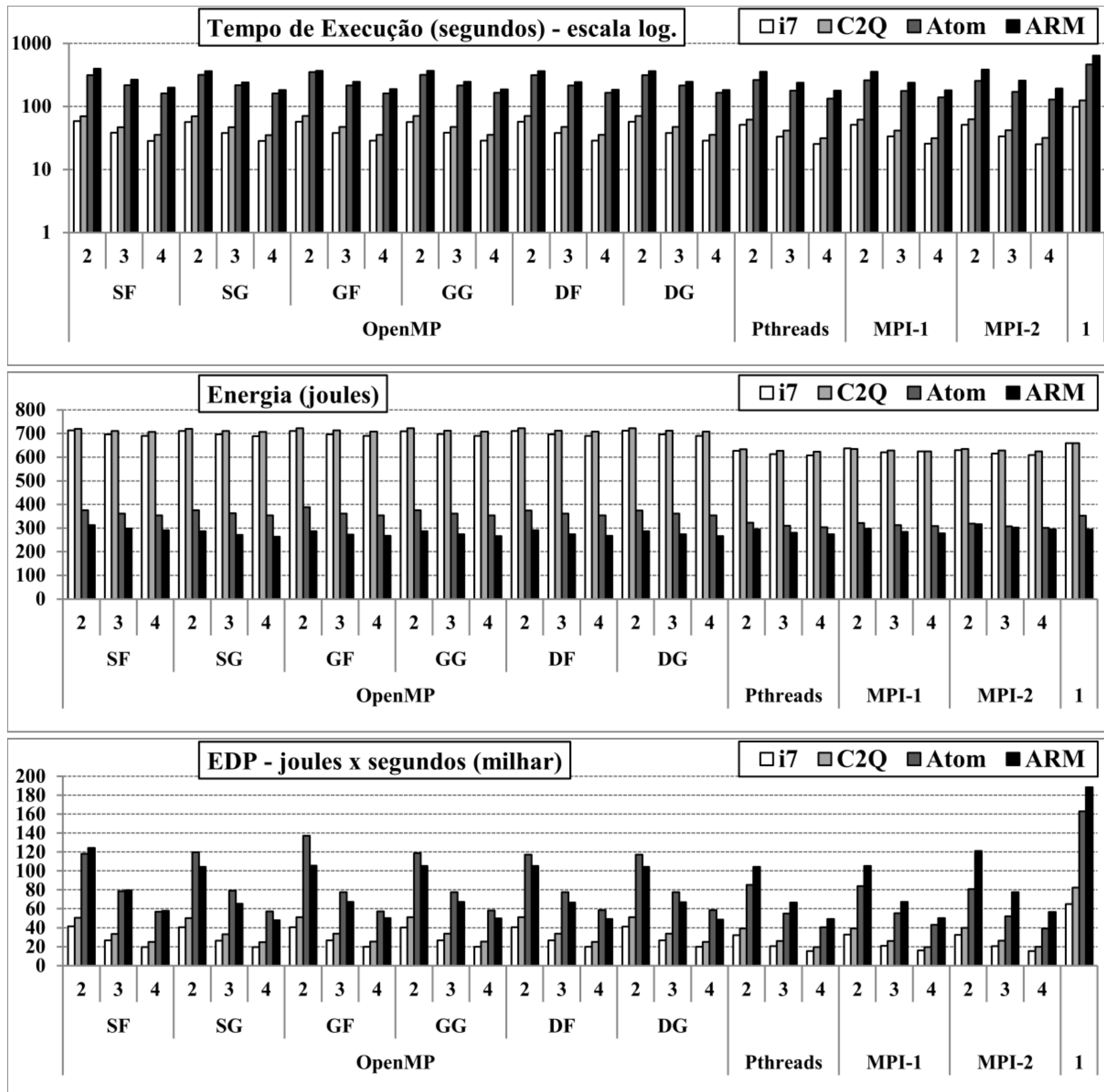


Figura B.8 – Resultados *Dijkstra* - Continuação



B.5 Similaridade entre Histogramas

Figura B.9 – Resultados Similaridade entre Histogramas

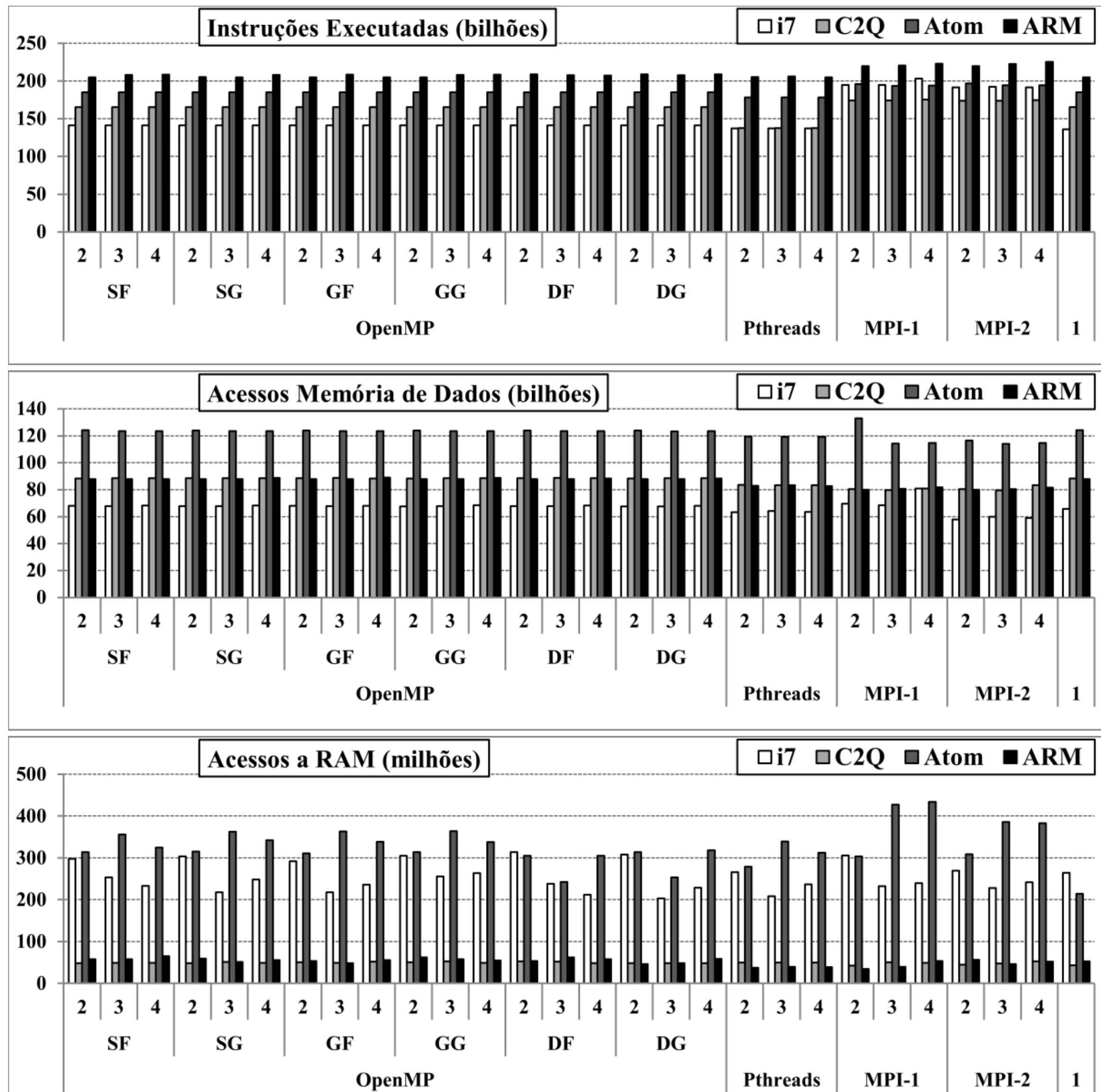
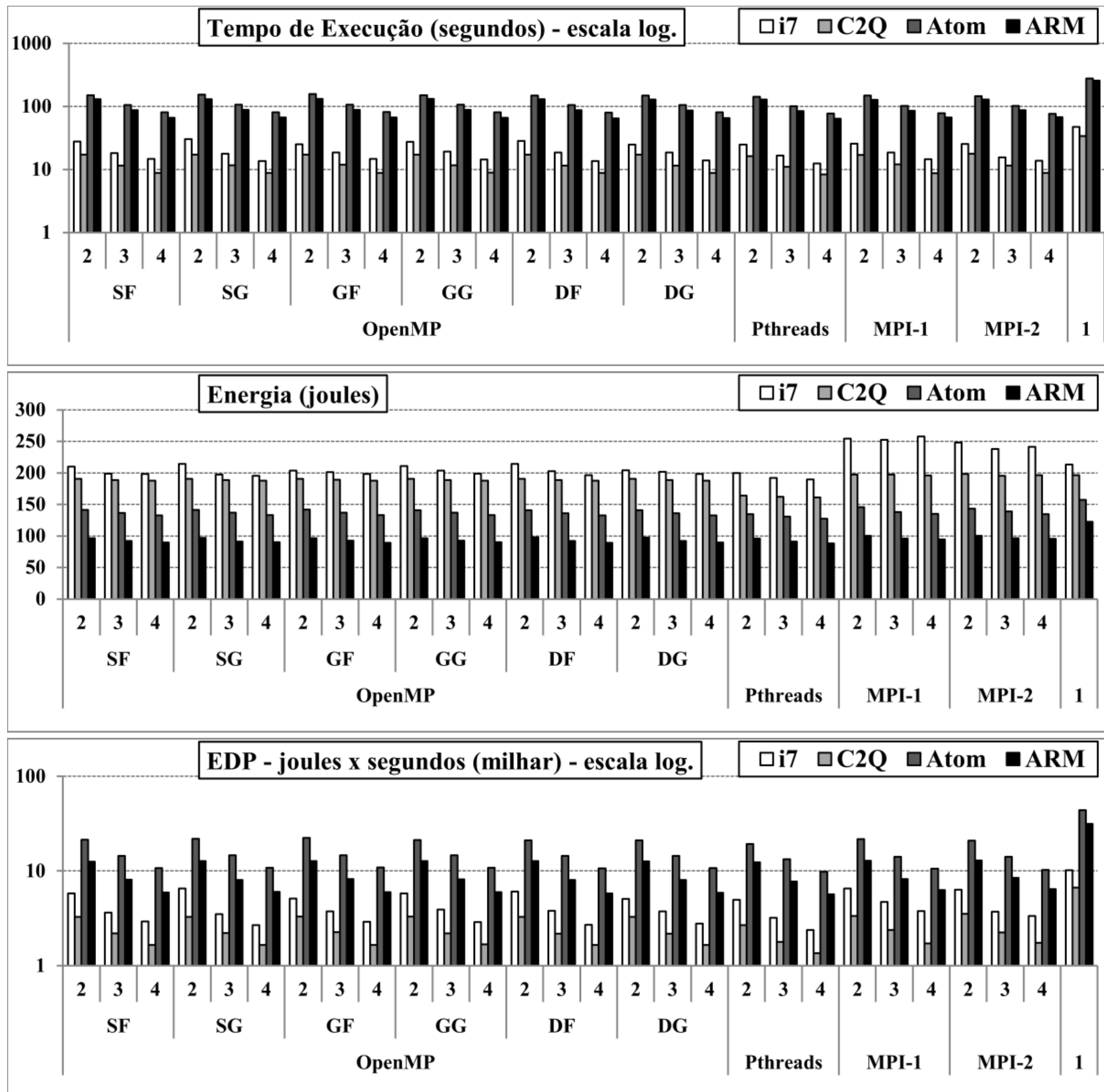


Figura B.10 – Resultados Similaridade entre Histogramas - Continuação



B.6 Decomposição-LU

Figura B.11 – Resultados Decomposição-LU

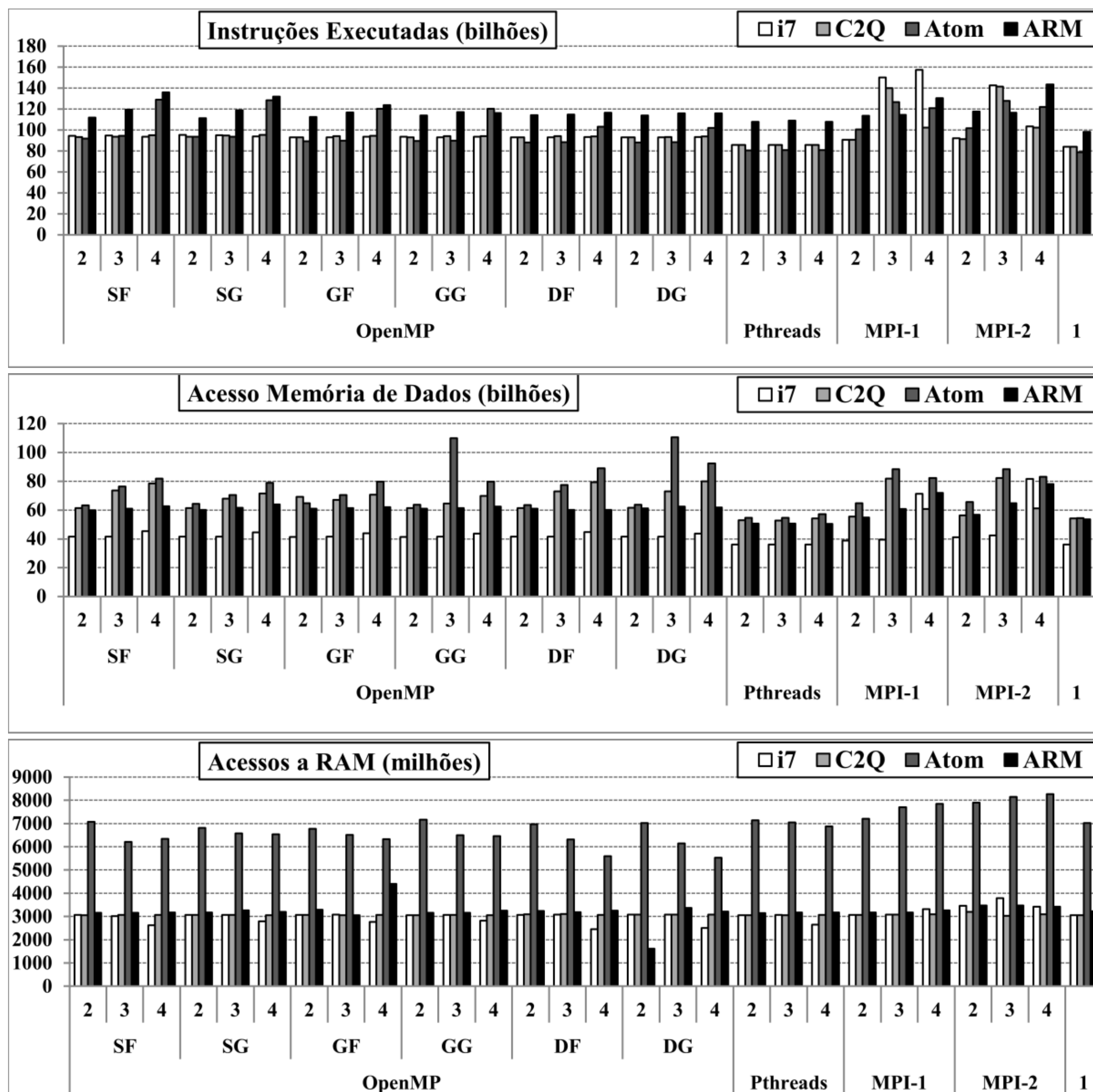
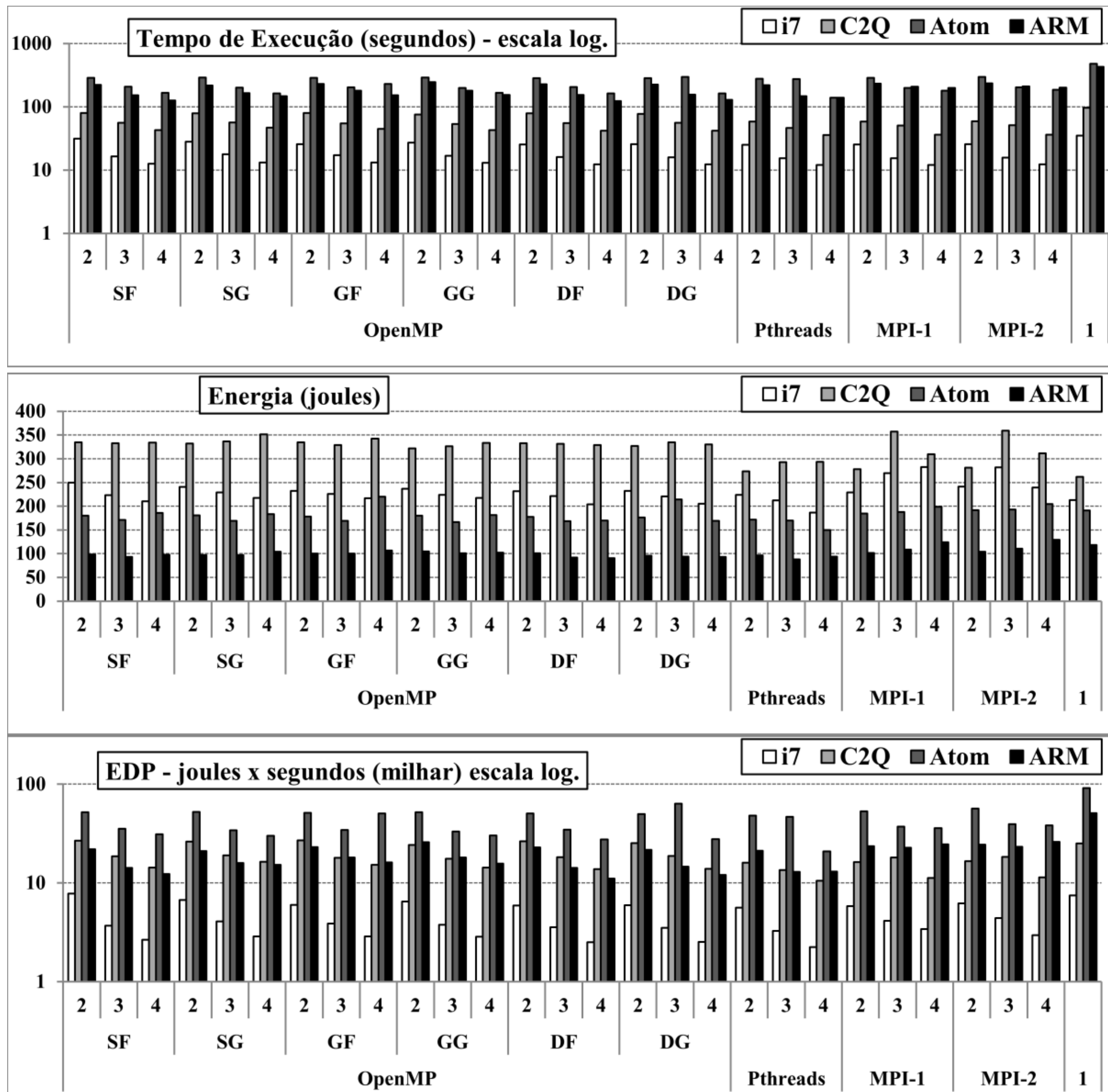


Figura B.12 – Resultados Decomposição-LU - Continuação



B.7 Método de Jacobi

Figura B.13 – Resultados Método de Jacobi

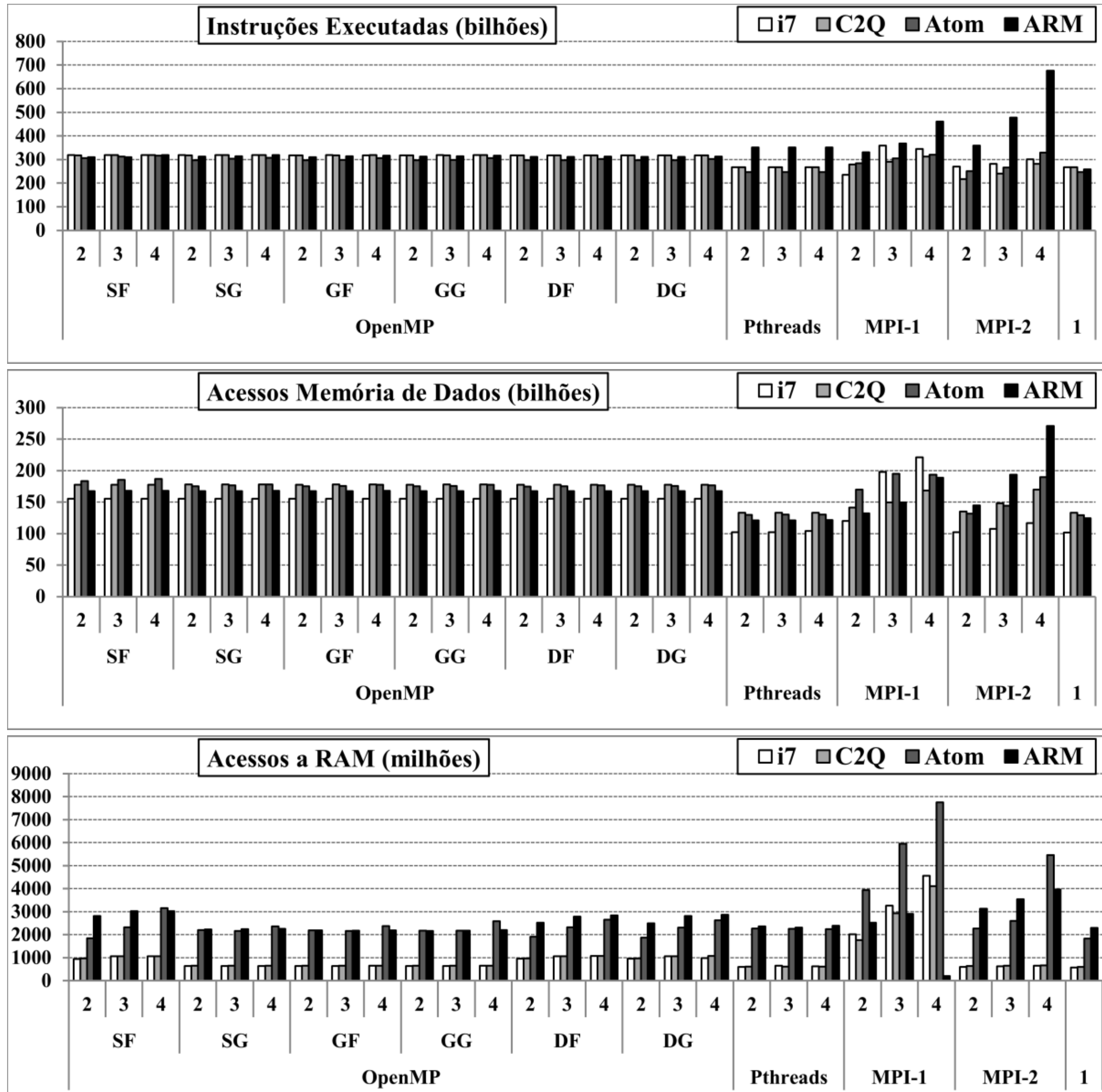
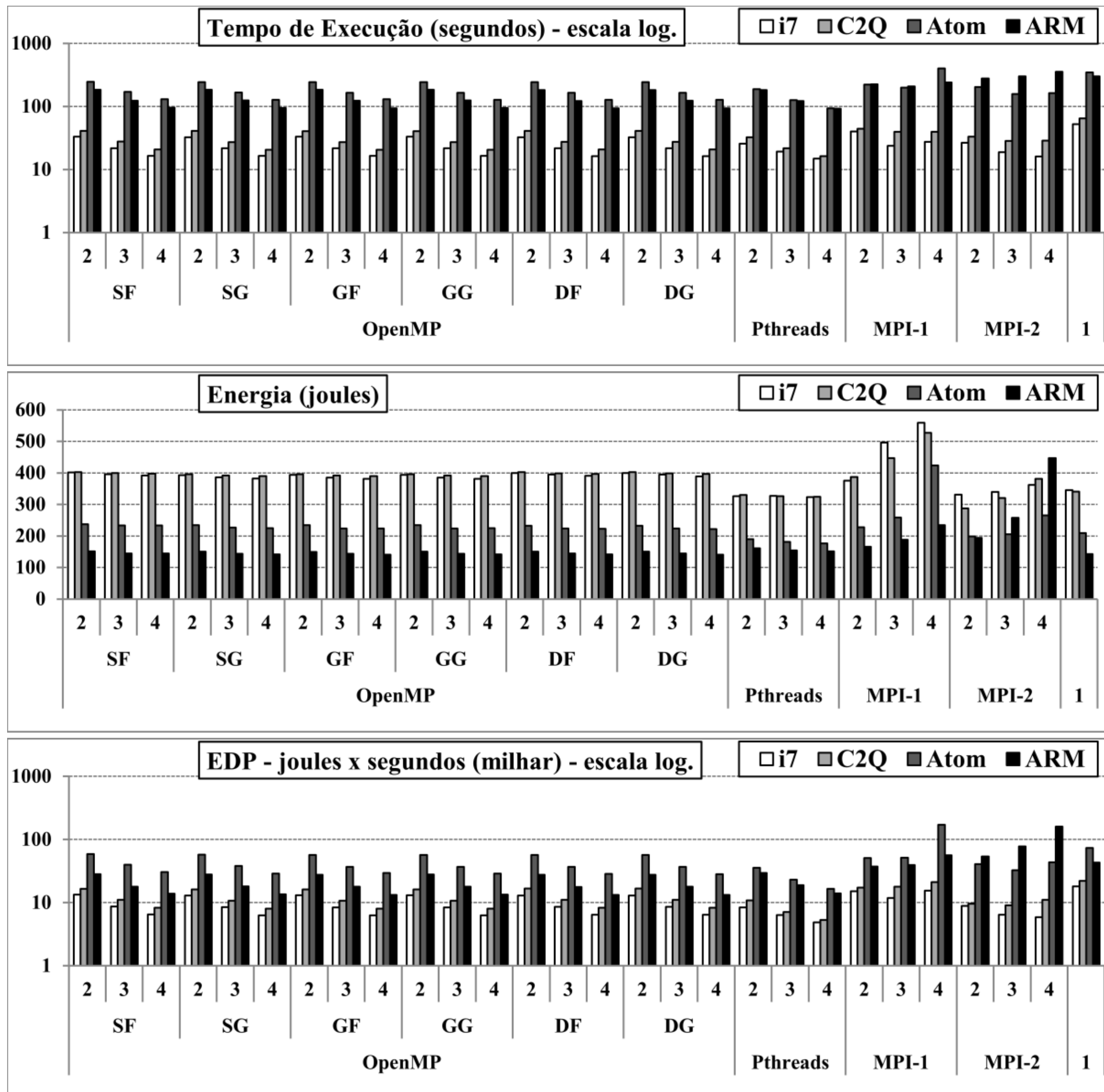


Figura B.14 – Resultados Método de Jacobi - Continuação



B.8 Multiplicação de Matriz

Figura B.15 – Resultados Multiplicação de Matriz

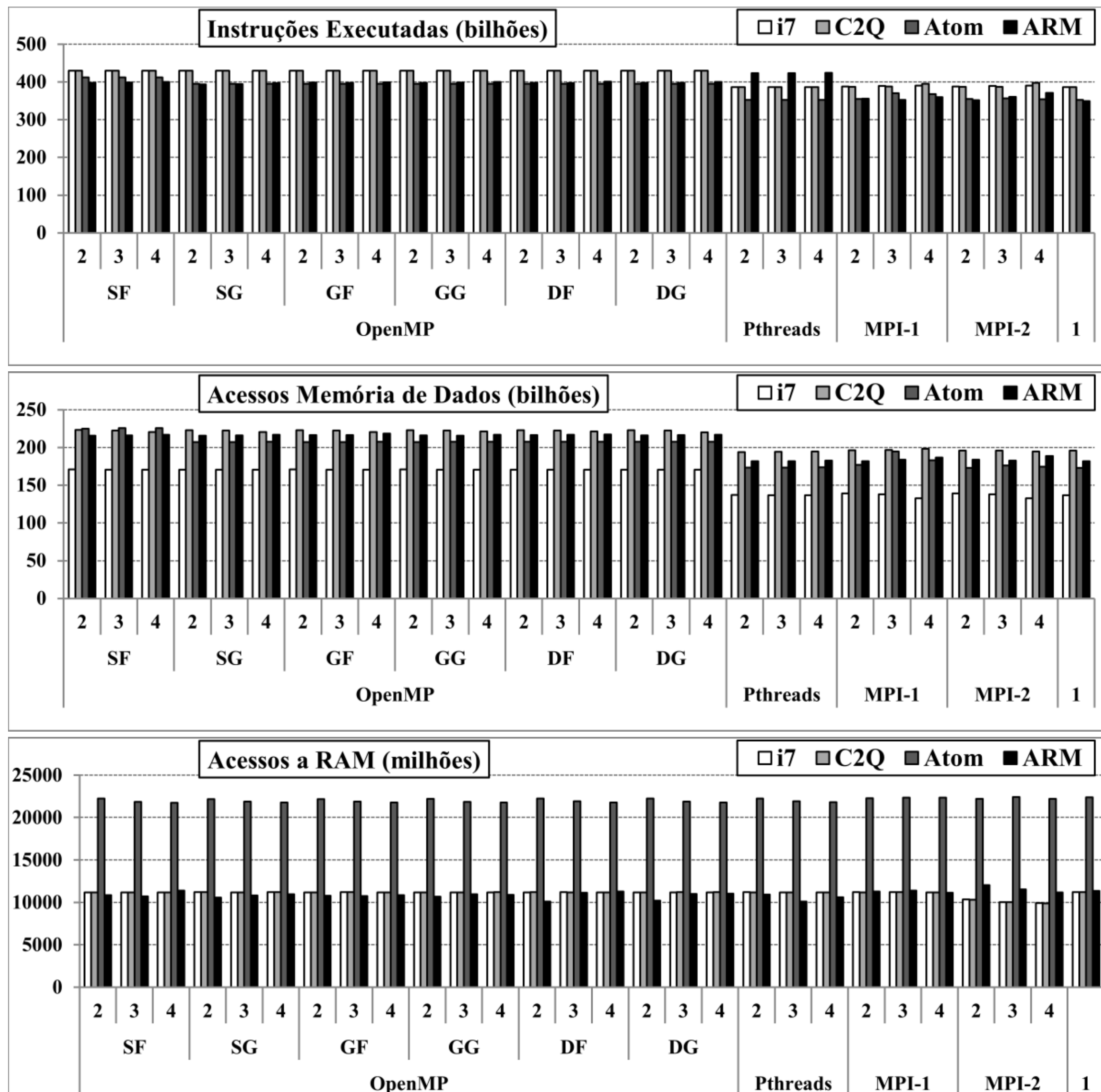
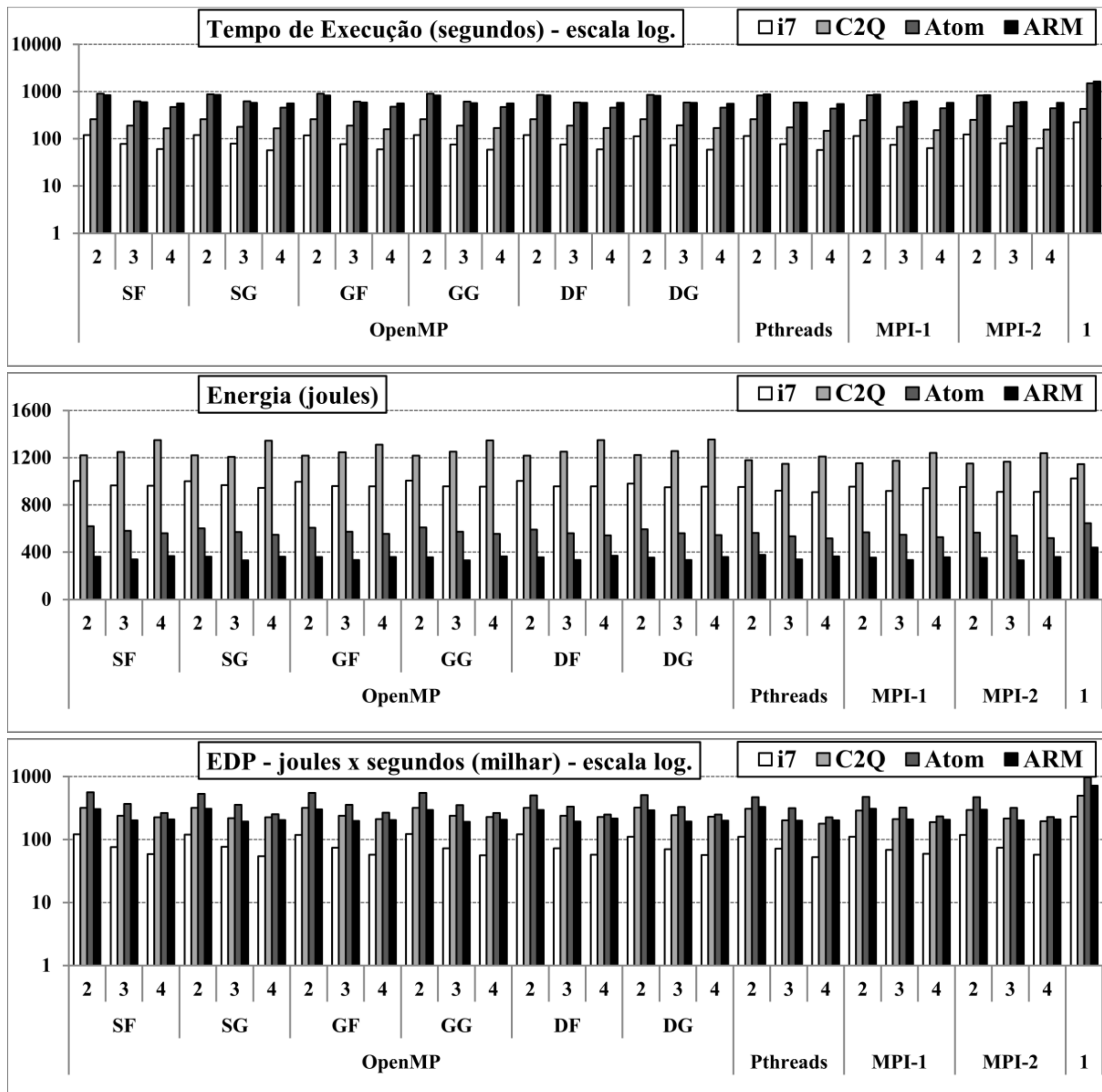


Figura B.16 – Resultados Multiplicação de Matriz - Continuação



B.9 Jogo da Vida

Figura B.17 – Resultados Jogo da Vida

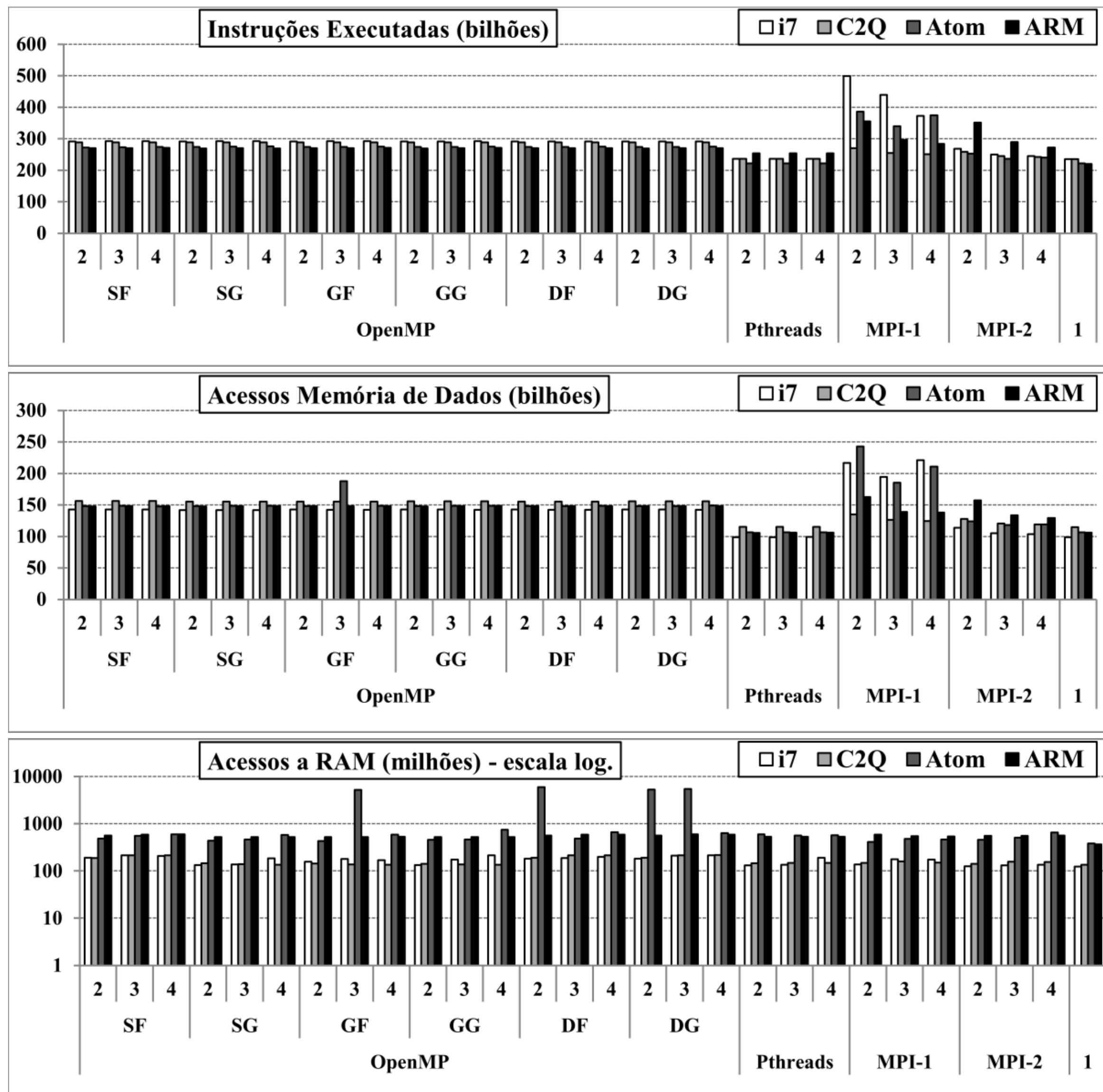
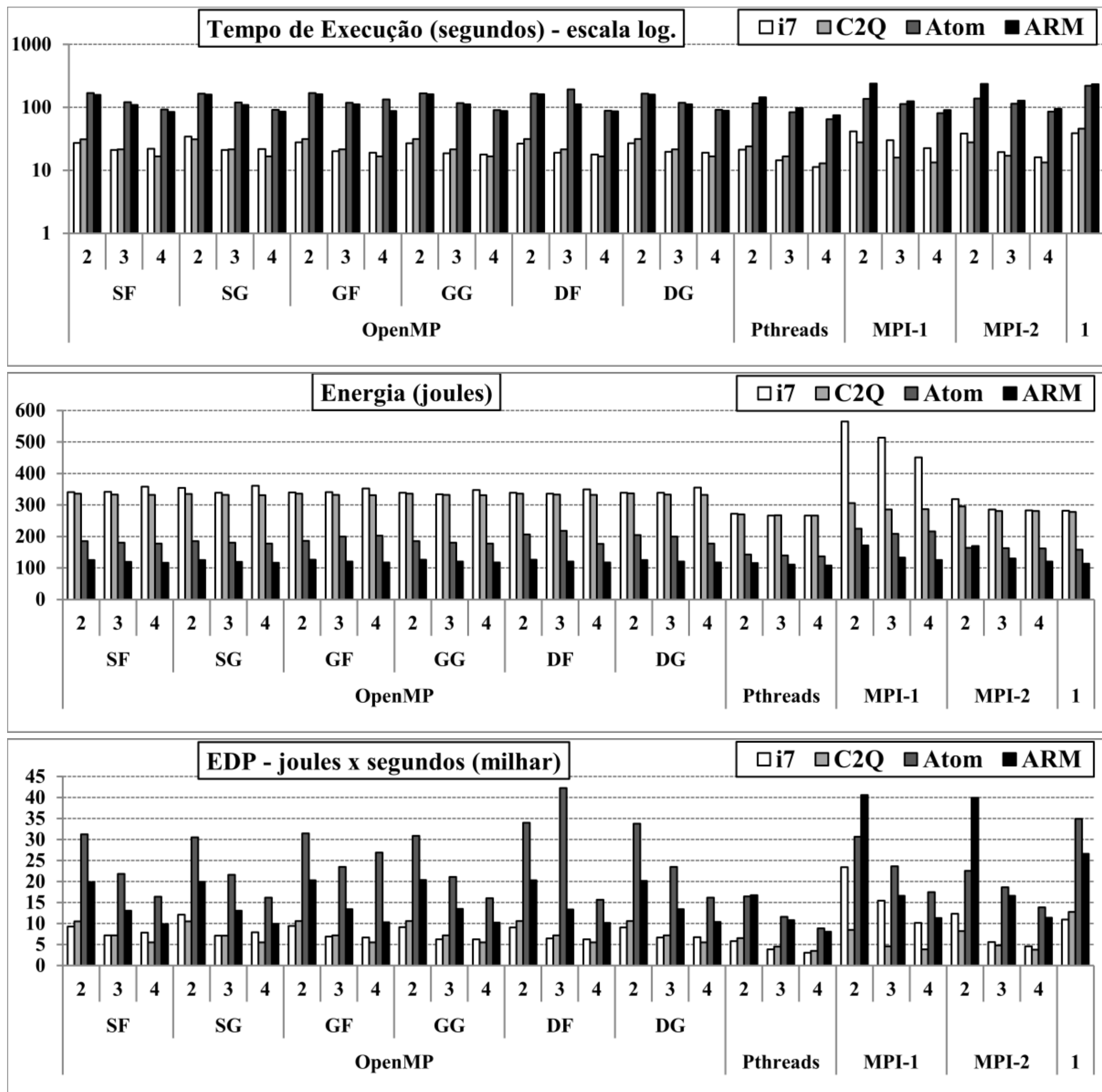


Figura B.18 – Resultados Jogo da Vida - Continuação



B.10 Gram-Schmidt

Figura B.19 – Resultados Gram-Schmidt

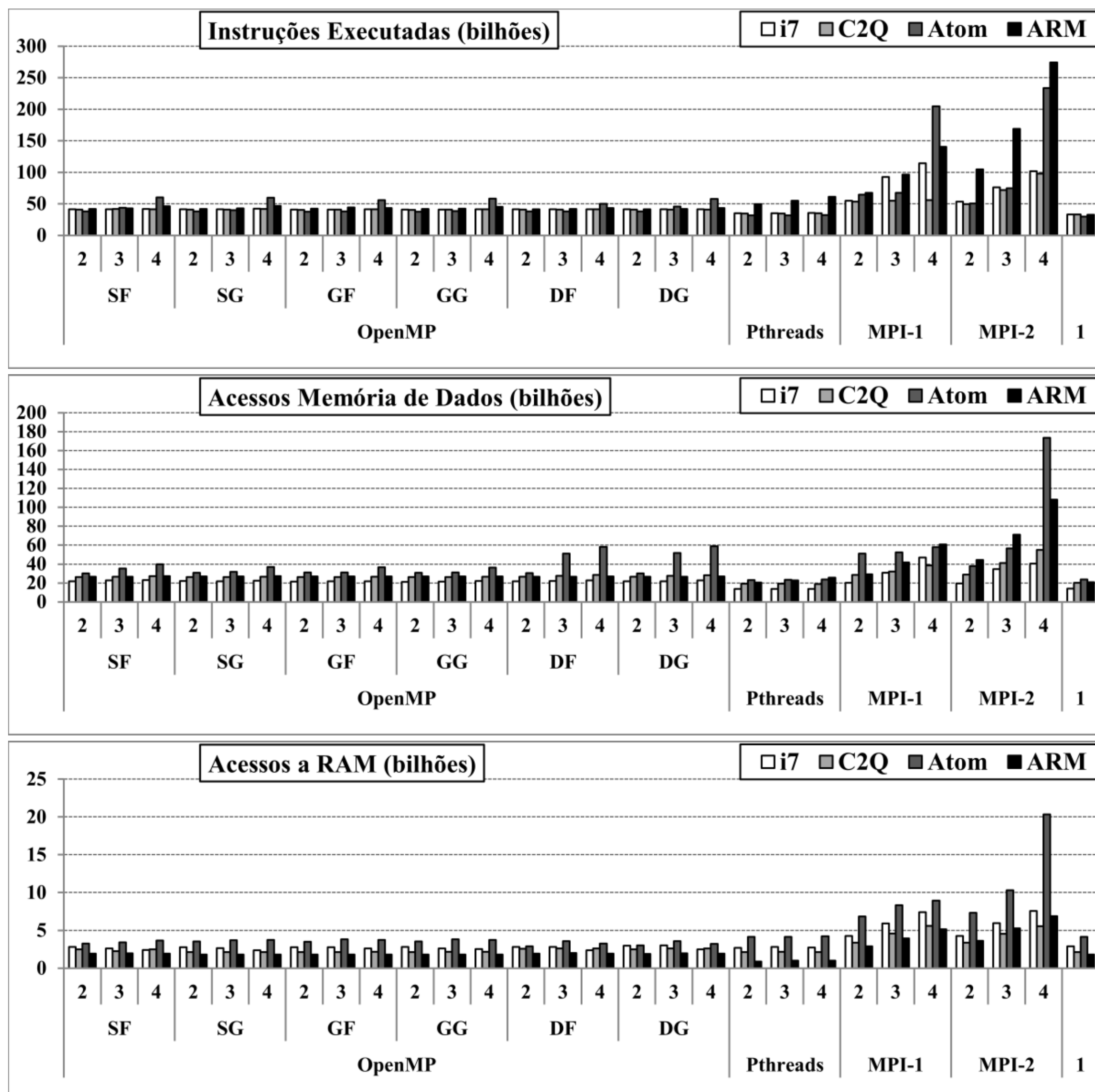
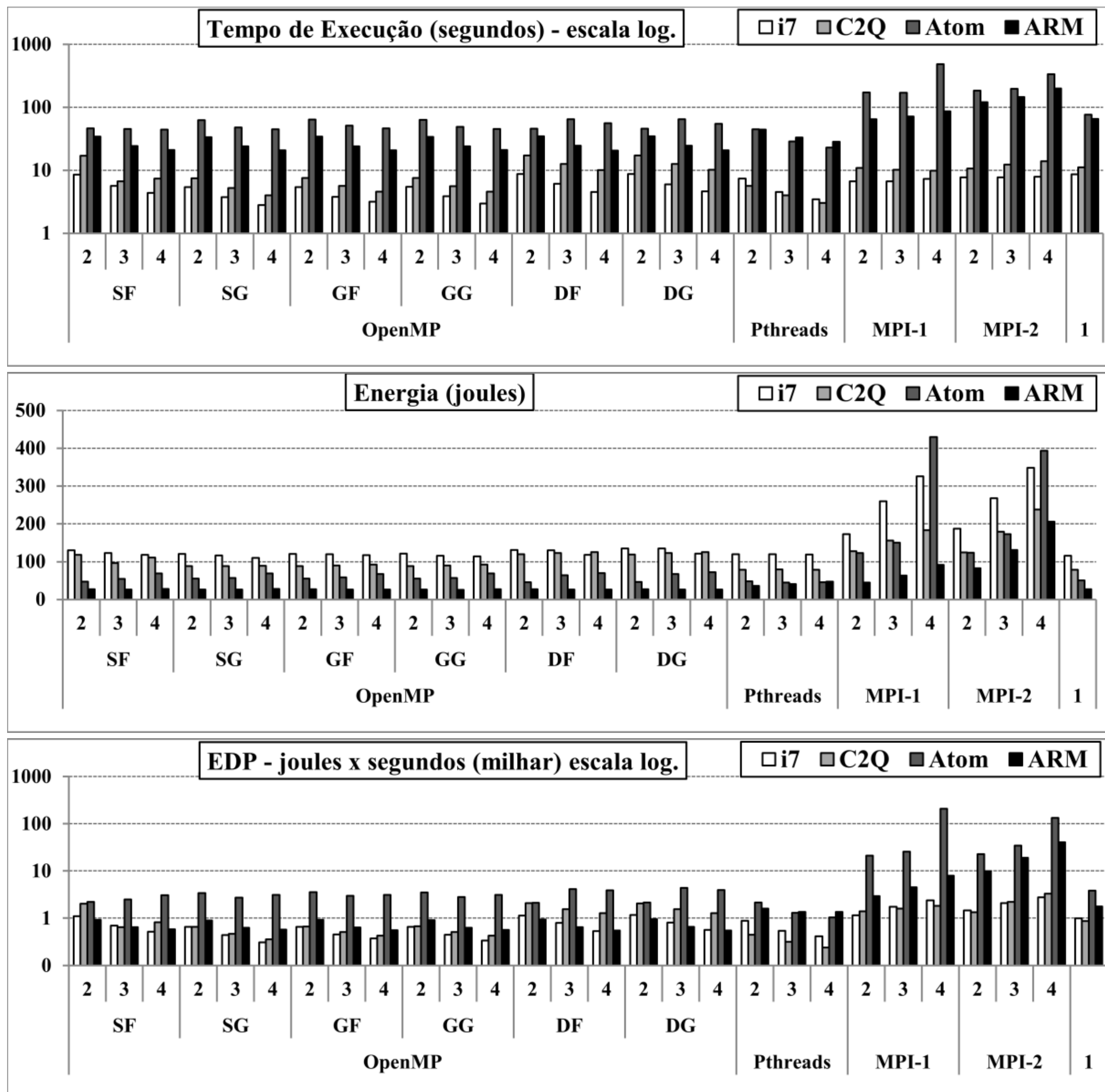


Figura B.0.20 – Resultados Gram-Schmidt - Continuação



B.11 Ordenação Par-Impar

Figura B.21 – Resultados Ordenação Par-Impar

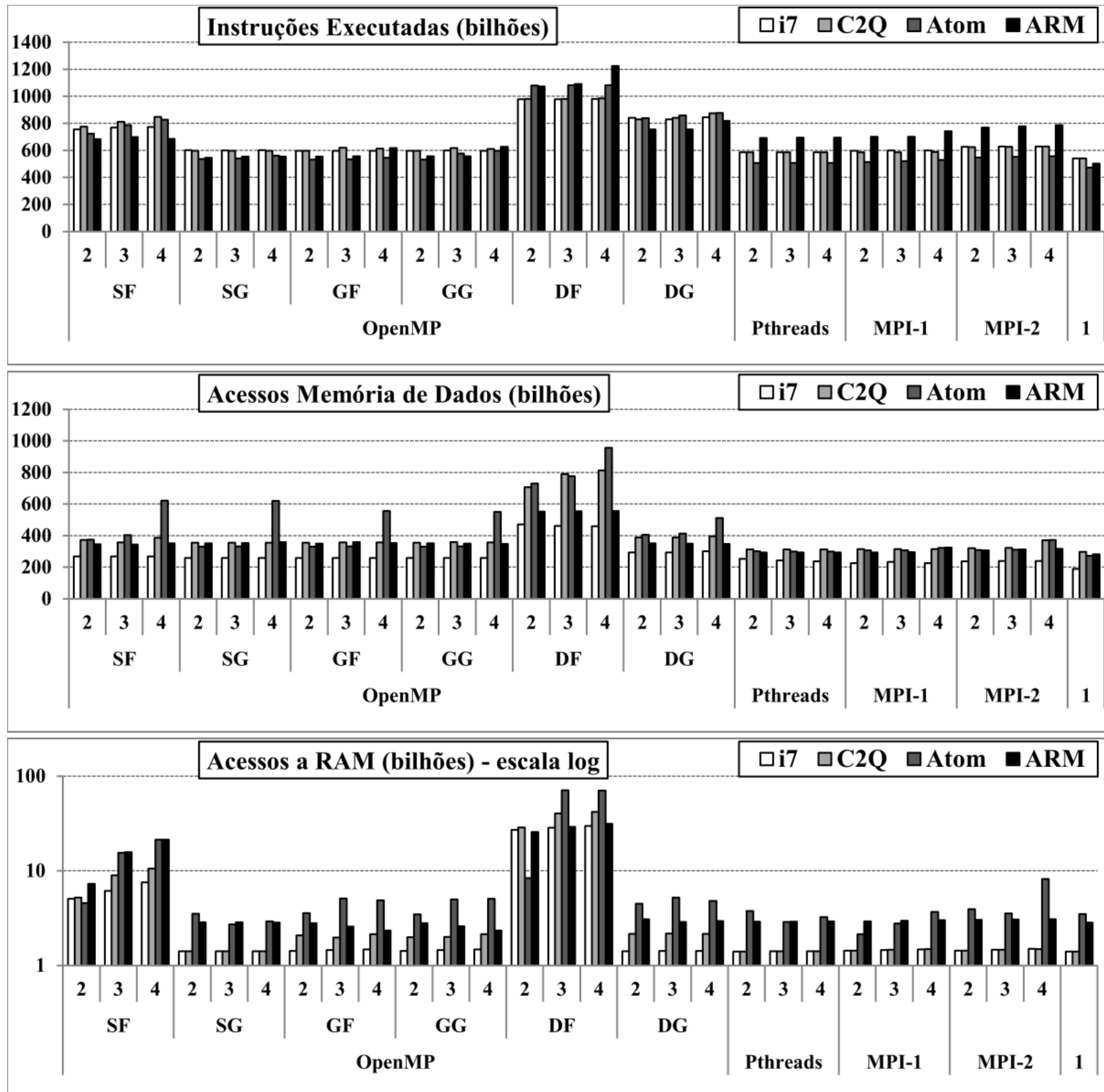
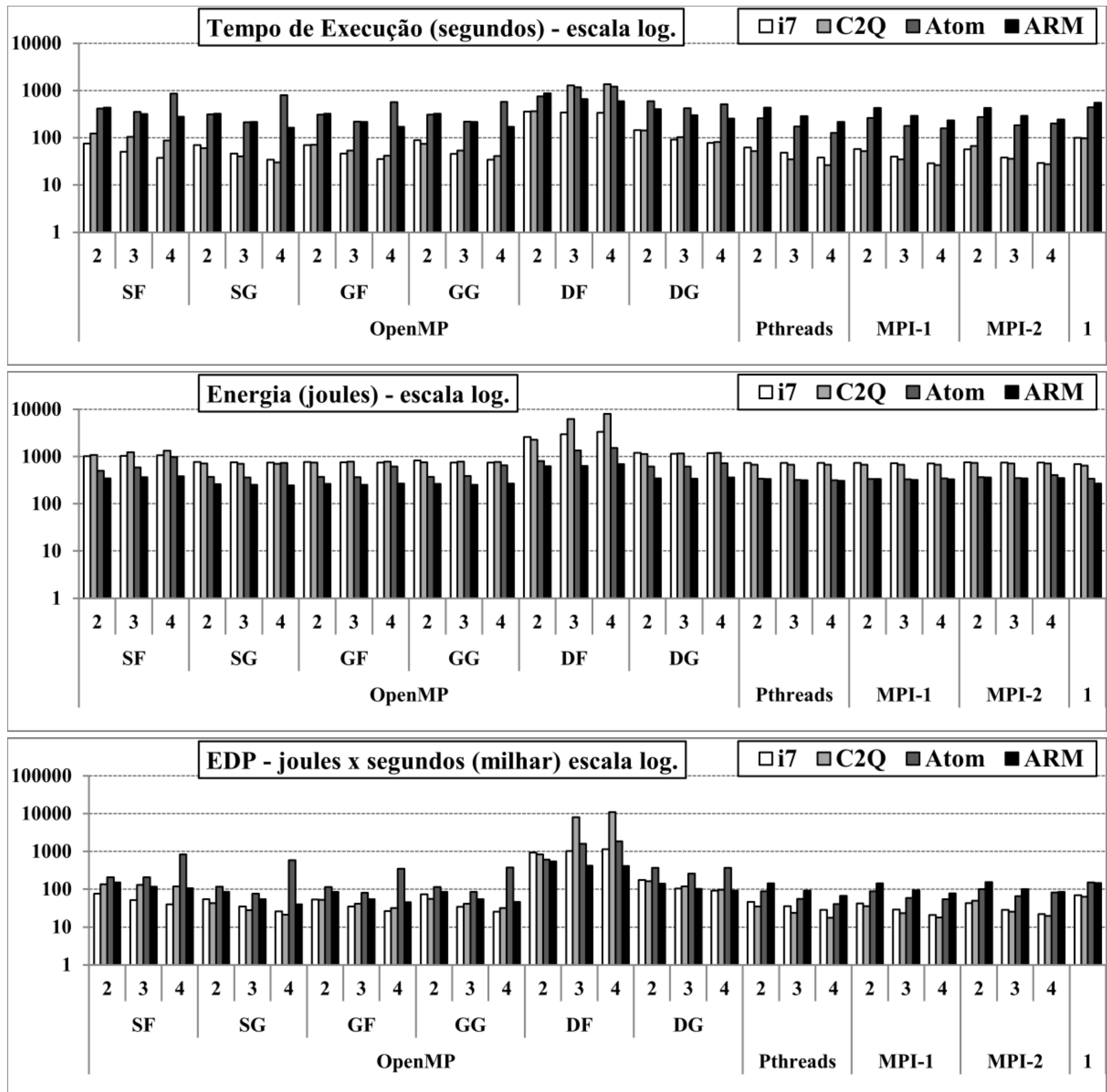


Figura B.22 – Resultados Par-Impar - Continuação



B.12 *Turing Ring*

Figura B..23 – Resultados *Turing Ring*

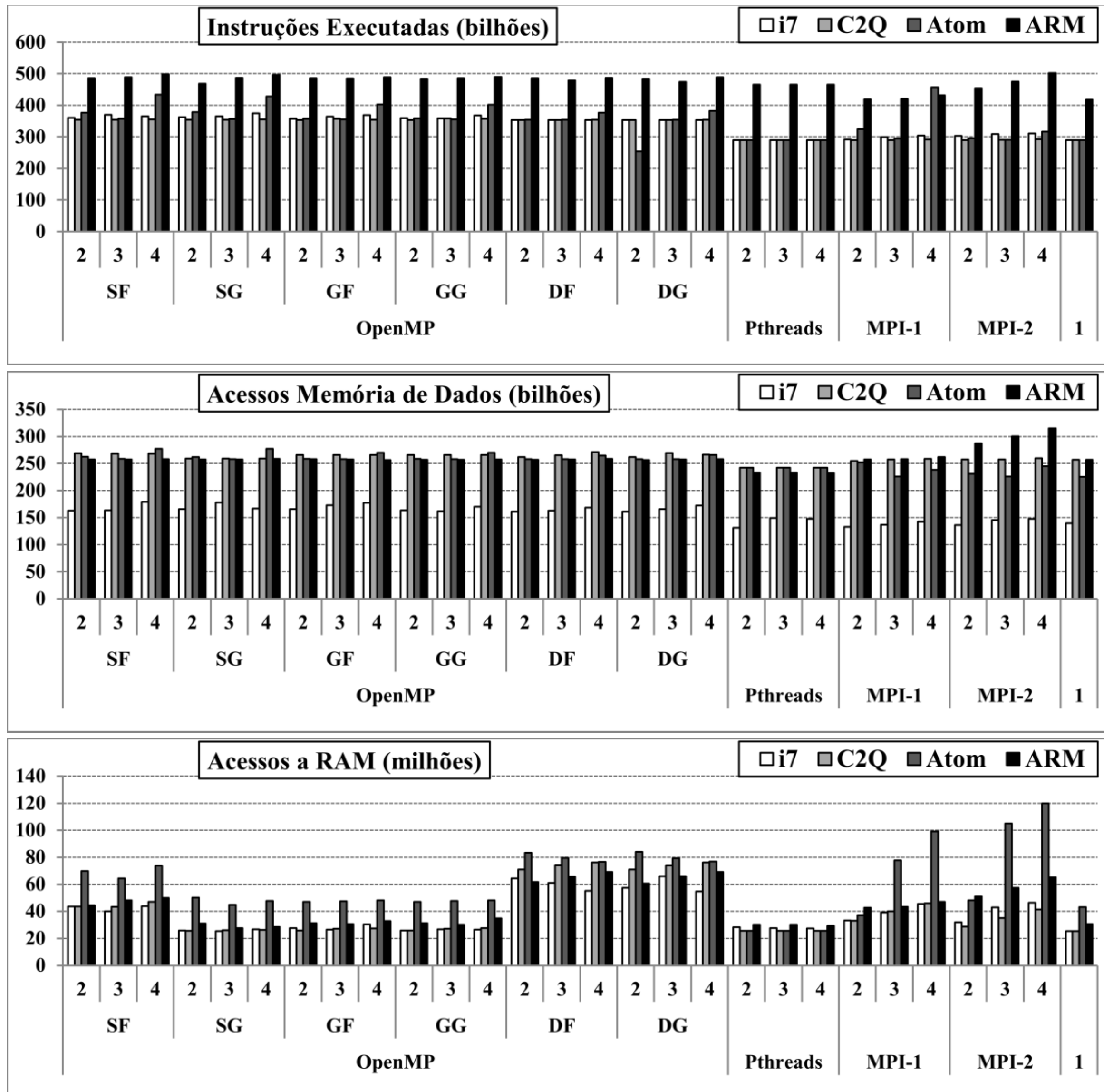


Figura B.24 – Resultados *Turing Ring* - Continuação