

UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Rafael Gauna Trindade

**ESCALONADOR ADAPTATIVO PARA LAÇOS PARALELOS EM
PROCESSADORES MULTINÚCLEO ASSIMÉTRICOS**

Santa Maria, RS
2020

Rafael Gauna Trindade

**ESCALONADOR ADAPTATIVO PARA LAÇOS PARALELOS EM PROCESSADORES
MULTINÚCLEO ASSIMÉTRICOS**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração em Ciência da Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Ciência da Computação**. Defesa realizada por videoconferência.

ORIENTADOR: Prof. João Vicente Ferreira Lima

Santa Maria, RS
2020

Trindade, Rafael Gauna
Escalonador Adaptativo para Laços Paralelos em
Processadores Multinúcleo Assimétricos / Rafael Gauna
Trindade.- 2020.
72 p.; 30 cm

Orientador: João Vicente Ferreira Lima
Dissertação (mestrado) - Universidade Federal de Santa
Maria, Centro de Tecnologia, Programa de Pós-Graduação em
Ciência da Computação , RS, 2020

1. Algoritmos Adaptativos 2. Escalonadores 3. Laços
Paralelos 4. Processadores Multinúcleo Assimétricos 5.
Computação Paralela I. Ferreira Lima, João Vicente II.
Título.

Sistema de geração automática de ficha catalográfica da UFSM. Dados fornecidos pelo autor(a). Sob supervisão da Direção da Divisão de Processos Técnicos da Biblioteca Central. Bibliotecária responsável Paula Schoenfeldt Patta CRB 10/1728.

Declaro, RAFAEL GAUNA TRINDADE, para os devidos fins e sob as penas da lei, que a pesquisa constante neste trabalho de conclusão de curso (Dissertação) foi por mim elaborada e que as informações necessárias objeto de consulta em literatura e outras fontes estão devidamente referenciadas. Declaro, ainda, que este trabalho ou parte dele não foi apresentado anteriormente para obtenção de qualquer outro grau acadêmico, estando ciente de que a inveracidade da presente declaração poderá resultar na anulação da titulação pela Universidade, entre outras consequências legais.

©2020

Todos os direitos autorais reservados a Rafael Gauna Trindade. A reprodução de partes ou do todo deste trabalho só poderá ser feita mediante a citação da fonte.

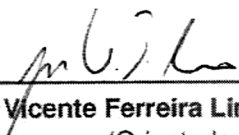
End. Eletr.: rtrindade@inf.ufsm.br

Rafael Gauna Trindade

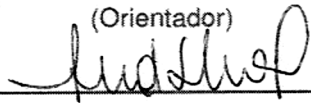
**ESCALONADOR ADAPTATIVO PARA LAÇOS PARALELOS EM PROCESSADORES
MULTINÚCLEO ASSIMÉTRICOS**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração em Ciência da Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Ciência da Computação**.

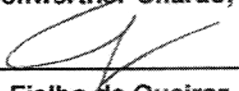
Aprovado em 30 de março de 2020:



João Vicente Ferreira Lima, Dr. (UFSM)
(Orientador)



Andrea Schwertner Charão, Dra. (UFSM)



Leonardo Fialho de Queiroz, PhD. (Atos)

Santa Maria, RS
2020

DEDICATÓRIA

Dedico este trabalho a todos os(as) pesquisadores(as) e demais profissionais da ciência, que trabalham continuamente em prol do conhecimento, destacando uma das melhores características do ser humano: a curiosidade.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais e irmãos, pelo amor, atenção e apoio incondicional que prestaram durante a pós-graduação e também durante toda a minha vida.

Agradeço ao meu professor orientador, João, e aos professores que comporam a banca, Andrea e Leonardo, que contribuíram de forma insubstituível para a minha formação e a de meus colegas de programa.

Agradeço pelo companheirismo e força passados pelos meus amigos e colegas de Laboratório, com os quais convivi durante estes dois anos no Laboratório de Sistemas de Computação. Agradecimentos especiais aos mais próximos: Ana, Bruno, Iago, Lucas e Maurício.

Agradeço aos meus demais amigos, que me acompanharam e me deram forças desde o começo. Agradecimentos especiais aos mais próximos: Amanda, Bruna, Caroline, Cassiano, Felipe, João, Laura, Linda, Márian, Otávio, Pedro, Raul e Vinícius.

Por fim, agradeço à Universidade Federal de Santa Maria, pelo ambiente de ensino de excelência, amigável e acolhedor que proporciona aos seus alunos e que me proporcionou durante esses dois anos de programa.

*Do not parallelize what does not matter!
I talked it to too many users who just parallelized every loop in their program...
that's not a good idea, all right?*

(Ruud van der Pas, 2019)

RESUMO

ESCALONADOR ADAPTATIVO PARA LAÇOS PARALELOS EM PROCESSADORES MULTINÚCLEO ASSIMÉTRICOS

AUTOR: Rafael Gauna Trindade

ORIENTADOR: João Vicente Ferreira Lima

A crescente demanda por potência computacional e eficiência energética em computação móvel desencadeou o desenvolvimento de processadores heterogêneos com núcleos especializados para diferentes tipos de tarefas computacionais, como processadores ARM big.LITTLE, que tem núcleos distintos que aliam desempenho a baixo consumo energético. Tal diferença na composição dos núcleos nesse tipo de processadores acaba induzindo uma assimetria no desempenho computacional desses sistemas, tornando complicada a tarefa de prever o comportamento de aplicações paralelas com relação a desempenho quando usados todos os seus núcleos. Essa assimetria pode ser percebida em aplicações que façam uso de laços paralelos, um recurso de programação paralela que permite dividir a carga de trabalho de uma rotina iterativa entre os núcleos presentes em um processador. Escalonadores de laços paralelos que não são projetados de modo a evitar a perda de desempenho em processadores multinúcleo assimétricos (AMPs) podem comprometer a implementação de soluções de *software* construídas para esse tipo de arquitetura. Esta dissertação apresenta a proposta de implementação de um escalonador para laços paralelos que utiliza um algoritmo adaptativo para distribuição da carga de trabalho entre as *threads*, visando extrações de desempenho mais eficientes em AMPs. O escalonador utiliza-se de roubo de trabalho paralelo e extração de trabalho sequencial o mais livre de travas quanto for possível para fazer frente às demais soluções existentes. Sua implementação foi realizada na linguagem C++, com possibilidade de portabilidade para a linguagem C. Com a finalidade de avaliar o desempenho da solução, uma análise foi realizada sobre o conjunto de *benchmarks* NAS e quatro aplicações científicas distintas bem consolidadas na literatura relacionada, sobre dois ambientes assimétricos embarcados reais, contra duas soluções existentes (OpenMP e Intel TBB). A análise mostra que o escalonador consegue extrair mais desempenho em determinados casos e se aproxima muito das melhores soluções na maioria dos casos restantes, com potencial de escalabilidade em teoria maior para casos onde o sobrecusto de escalonamento se torna um empecilho nas demais soluções.

Palavras-chave: Algoritmos Adaptativos. Escalonadores. Laços Paralelos. Processadores Multinúcleo Assimétricos. Computação Paralela. Computação Heterogênea. Computação de Alto Desempenho. Roubo de Trabalho. OpenMP. Intel TBB.

ABSTRACT

AN ADAPTIVE SCHEDULER FOR PARALLEL LOOPS ON ASYMMETRIC MULTICORE PROCESSORS

AUTHOR: Rafael Gauna Trindade

ADVISOR: João Vicente Ferreira Lima

The growing demand for computing power and energy efficiency in mobile computing has triggered the development of heterogeneous processors with specialized cores for different types of computational tasks, such as ARM big.LITTLE processors, which have different cores that combine performance with low energy consumption. Such difference in the composition of the cores in this kind of processors ends up inducing an asymmetry in the computational performance of these systems, making complicated the task of predicting the behavior of parallel applications in relation to performance when using all their cores. This asymmetry can be detected in applications that use parallel loops, a parallel programming feature that allows to divide the workload of an iterative routine between the cores present in a processor. Parallel loop schedulers that are not designed to prevent loss of performance in asymmetric multi-core processors (AMPs) can compromise the implementation of software solutions designed for this type of architecture. This dissertation presents the implementation proposal of a scheduler for parallel loops that uses an adaptive algorithm to distribute the workload among the threads, aiming at a better extraction of performance in AMPs. The scheduler uses of parallel work-stealing and lock-free as possible sequential extraction of work to face other existing solutions. Its implementation was carried out in the C++ language, with the possibility of portability to the C language. In order to evaluate the performance of the solution, an analysis was performed on the set of NAS benchmarks and four distinct well-established scientific applications in related literature, over two real asymmetric embedded environments, against two existing solutions (OpenMP and Intel TBB). The analysis shows that the scheduler is able to extract more performance in certain cases and is very close to the best solutions in most of the remaining cases, with greater scalability potential in theory for cases where the scheduling overhead becomes an obstacle in the other solutions.

Keywords: Adaptive Algorithms. Schedulers. Parallel Loops. Asymmetric Multicore Processors. Parallel Computing. Heterogeneous Computing. High Performance Computing. Work Stealing. OpenMP. Intel TBB.

LISTA DE GRÁFICOS

Gráfico 2.1 – Aceleração teórica paralela de uma arquitetura paralela heterogênea em relação à uma homogênea de mesmo custo	19
Gráfico 2.2 – Tempo de execução do conjunto de benchmarks SPEC OMP	20
Gráfico 3.1 – Tempo médio de execução com o NAS <i>Parallel Benchmark</i>	53
Gráfico 3.2 – Eficiência paralela com o NAS <i>Parallel Benchmark</i>	54
Gráfico 3.3 – Aceleração paralela com o NAS <i>Parallel Benchmark</i>	55
Gráfico 3.4 – Tempo médio de execução com as aplicações científicas selecionadas .	59
Gráfico 3.5 – Eficiência paralela com as aplicações científicas selecionadas	60
Gráfico 3.6 – Aceleração paralela com as aplicações científicas selecionadas	61

LISTA DE ILUSTRAÇÕES

Ilustração 2.1 – Diagrama da Taxonomia de Flynn e Rudd (1996).	17
Ilustração 2.2 – Diagrama exemplo de um AMP.	18
Ilustração 2.3 – Classificação de algoritmos híbridos.	22
Ilustração 2.4 – Caso exemplo de paralelização de laço com OpenMP	23
Ilustração 2.5 – Caso exemplo de paralelização de laço com a diretiva <code>taskloop</code> do OpenMP	25
Ilustração 2.6 – Caso exemplo de paralelização de laço com biblioteca Intel TBB	26
Ilustração 3.1 – Fluxograma do ciclo de vida de uma <i>thread</i>	34
Ilustração 3.2 – Caso exemplo de extração de trabalho em um sub-intervalo particular	34
Ilustração 3.3 – Escalonador Adaptativo – Algoritmo de Extração de Trabalho Sequencial	35
Ilustração 3.4 – Caso exemplo de um conflito durante extração de trabalho	36
Ilustração 3.5 – Escalonador Adaptativo – Algoritmo de Extração de Trabalho Paralelo	37
Ilustração 3.6 – Exemplos de divisão inicial e roubo de trabalho	40
Ilustração 3.7 – Diagrama UML das classes utilizadas pelo escalonador.	41
Ilustração 3.8 – Esquema de redução em árvore empregado pelo escalonador.	41
Ilustração 3.9 – Comparação de desempenho de tipos atômicos do C++ com outros tipos de trava.	42
Ilustração 3.10 – Assinaturas dos métodos de laço e redução paralelas	43
Ilustração 3.11 – Caso exemplo de paralelização de laço com a API proposta	43

LISTA DE TABELAS

Tabela 3.1 – Dispositivos Embarcados Utilizados na Avaliação	48
Tabela 3.2 – Núcleos de CPU por Configuração de Cluster	48
Tabela 3.3 – Aplicações e Kernels do NAS Parallel Benchmark Usadas na Avaliação .	49
Tabela 3.4 – Aplicações Científicas Usadas na Avaliação	51
Tabela 3.5 – Tempos de Execução Sequencial dos <i>Benchmarks</i> , em Segundos	52

LISTA DE ABREVIATURAS E SIGLAS

<i>AMP</i>	<i>Asymmetric Multicore Processor</i> (Processador Multinúcleo Assimétrico)
<i>APME</i>	Aceleração Paralela Máxima Esperada
<i>CPU</i>	<i>Central Processing Unit</i> (Unidade de Processamento Central)
<i>HPC</i>	<i>High Performance Computing</i> (Computação de Alto Desempenho)
<i>ISA</i>	<i>Intruction Set Architecture</i> (Conjunto de Instruções de Arquitetura)
<i>PU</i>	<i>Processing Unit</i> (Unidade de Processamento)
<i>SMP</i>	<i>Symmetric Multicore Processor</i> (Processador Multinúcleo Simétrico)

SUMÁRIO

1	INTRODUÇÃO	13
2	FUNDAMENTAÇÃO TEÓRICA	15
2.1	ARQUITETURAS PARALELAS	15
2.1.1	Taxonomia de Arquiteturas Paralelas	15
2.2	PROCESSADORES MULTINÚCLEO ASSIMÉTRICOS	16
2.2.1	ARM big.LITTLE™	17
2.2.2	Efeitos da Assimetria no Desempenho	18
2.3	ALGORITMOS ADAPTATIVOS	21
2.4	ESCALONADORES DE LAÇOS PARALELOS	22
2.4.1	OpenMP	23
2.4.2	Intel TBB	25
2.5	TRABALHOS RELACIONADOS	27
2.6	SUMÁRIO	31
3	ESCALONADOR ADAPTATIVO	32
3.1	LAÇO PRINCIPAL DO ESCALONADOR	32
3.2	ALGORITMOS DE EXTRAÇÃO SEQUENCIAL E PARALELA	35
3.3	OTIMIZAÇÃO	38
3.4	IMPLEMENTAÇÃO	39
3.4.1	API para Laços Paralelos	42
3.5	SUMÁRIO	44
3.6	ANÁLISE DE DESEMPENHO	45
3.7	METODOLOGIA	45
3.7.1	Métricas	46
3.7.2	Ambientes	47
3.7.3	Benchmarks	49
3.8	RESULTADOS EXPERIMENTAIS	51
3.8.1	Desempenho Sequencial	51
3.8.2	NAS <i>Parallel Benchmark</i>	52
3.8.3	Aplicações Científicas	57
3.9	SUMÁRIO	63
4	CONCLUSÃO	64
	REFERÊNCIAS BIBLIOGRÁFICAS	67

1 INTRODUÇÃO

Com a demanda crescente da indústria por dispositivos computacionais que combinem boa capacidade de processamento com eficiência energética – um aumento substancial em vendas de smartphones e tablets, por exemplo –, novos tipos de necessidades surgiram, e com elas novas tecnologias de hardware foram propostas para poder supri-las. O hoje antigo conceito de processador multinúcleo simétrico (*Symmetric Multicore Processor*, em inglês – SMP), mesmo com ajustes dinâmicos de consumo de energia e frequência de operação (i.e. DVFS), acaba contido por limitações estruturais uma vez que é difícil criar um núcleo dinâmico o suficiente para acomodar tanto baixo consumo de energia quanto bom poder de processamento. Como uma consequência de sua adaptabilidade à diversidade, Processadores Multinúcleo Assimétricos (*Asymmetric Multicore Processor*, em inglês – AMP) prometem ser benéficos para uma vasta gama de cenários de uso: núcleos diferentes em um AMP podem ser otimizados para consumo/desempenho, ou para diferentes domínios de aplicação, ou para explorar diferentes níveis de paralelismo (MITTAL, 2016).

A principal vantagem no uso de AMPs, do ponto de vista de Computação de Alto Desempenho (*High Performance Computing*, em inglês – HPC) se justamente dá pelas características híbridas de seus núcleos. Tarefas sequenciais ou com o desempenho determinado pelas computações em si (*CPU-bound*) obterão melhores resultados sendo executados em núcleos rápidos, ao passo que tarefas paralelas ou com o desempenho determinado pelos acessos à memória (*memory bound*) possuem benefícios em serem executadas em núcleos lentos (FEDOROVA et al., 2009). Estudos anteriores mostram que se é possível ganhar em torno de até 63% de desempenho em comparação a SMPs de mesma área de *chip* e consumo energético (KUMAR et al., 2004). Em um cenário de computação de alto-desempenho, essa flexibilidade associada a uma maior eficiência energética atrai a atenção, uma vez que esta última é um dos desafios na atual corrida pelo *hexascale*: construir computadores rápidos que consumam menos energia e realizem mais computações. Assim, em paralelo a já tradicional Top500¹ – lista semestral dos 500 supercomputadores mais rápidos do mundo – foi criada a Green500², uma lista dos 500 supercomputadores com a maior taxa de eficiência energética (em bilhões de operações de ponto flutuante (GFlop) por Watt) do mundo, ressaltando a sua importância para a área de HPC. E AMPs possibilitam o melhor destes dois mundos: apesar de possuírem características bem próprias em relação a eficiência energética, não estão limitadas a tal. Como consistem de ambientes de múltiplos núcleos, facilmente se concebe cenários em que abrir mão de eficiência energética em prol de desempenho pode se tornar conveniente.

¹<<http://top500.org>>

²<<http://green500.org>>

Entretanto extrair desempenho destas plataformas pode apresentar desafios devido a assimetria nelas contida. Essa possível diferença em capacidade de processamento entre diferentes tipos de núcleo pode levar a problemas de desempenho se o desenvolvedor de uma aplicação paralela não estiver consciente da assimetria do ambiente onde ela será executada. Durante a criação deste tipo de aplicação, desenvolvedores podem optar por utilizar laços paralelos, uma estratégia de programação paralela que permite dividir a carga de trabalho computacional de uma estrutura iterativa entre os núcleos disponíveis em uma máquina, geralmente de uma forma simples de ser entendida, uma vez que essas estruturas podem ser representadas por um intervalo numérico divisível. Um programador OpenMP, por exemplo, pode escolher entre alguns escalonadores de laços paralelos providos pela especificação da API e implementados em diferentes versões da *runtime* distribuídas em diferentes compiladores (GCC, LLVM, ICC, Nanos, IBM XL, entre outros). Entretanto, o uso descuidado desses escalonadores pode criar problemas de desempenho, uma vez que eles não foram projetados visando arquiteturas assimétricas.

Este trabalho propõe o desenvolvimento de um escalonador de laços paralelos que usa um algoritmo adaptativo para melhor adaptar uma carga de trabalho à ambientes de processamento assimétrico. Tal adaptabilidade é alcançada de forma inconsciente em relação aos recursos de processamento: o escalonador não diferencia processadores com características distintas, abrindo mão de ajustes específicos para processadores com desempenhos diferentes. Os objetivos específicos deste trabalho são:

- Apresentar um estudo na literatura de trabalhos anteriores que visavam a criação de escalonadores adaptativos, que serviram como base para o desenvolvimento do novo escalonador.
- Propor o desenvolvimento de um novo escalonador adaptativo, que possibilite adaptabilidade a diferentes ambientes de execução (respeitando a arquitetura à qual a aplicação foi construída), roubo de trabalho e baixo sobrecusto de escalonamento;
- Avaliar o desempenho dos escalonadores em um conjunto de *benchmarks* paralelos bem-estabelecidos em ambientes assimétricos reais e comparar o seu desempenho com os escalonadores existentes.

Esta dissertação está organizada da seguinte maneira: o Capítulo 2 fornece conceitos importantes abordados pelo trabalho, como processadores multinúcleo assimétricos e seu impacto no desempenho de aplicações paralelas, algoritmos adaptativos e escalonadores de laços paralelos existentes que hoje compõem o estado da arte, ao final apresentando os trabalhos relacionados encontrados na literatura. Os algoritmos e detalhes da implementação do escalonador proposto estão presentes no Capítulo 3. Uma análise de desempenho do escalonador sobre aplicações científicas e de *benchmarks* em plataformas assimétricas foi descrita no Capítulo 3.6. Por fim, o Capítulo 4 apresenta uma discussão sobre este trabalho e os resultados obtidos.

2 FUNDAMENTAÇÃO TEÓRICA

Esse capítulo apresenta as definições de Processadores Multinúcleo Assimétricos e algoritmos adaptativos de acordo com a literatura, e introduz alguns escalonadores de laços paralelos já existentes. Ao final, uma relação de trabalhos relacionados encontrados na literatura é apresentada e comentada.

2.1 ARQUITETURAS PARALELAS

O advento da multiplicação de unidades de processamento (*Processing Unit*, em inglês – PU) no mesmo chip trouxe muitas mudanças no campo de computação de alto desempenho. Dependendo de algoritmo e arquitetura, é possível acelerar uma aplicação por volta de N vezes para N núcleos idênticos, podendo inclusive ultrapassar-se esse valor se o desenvolvedor souber como otimizar seus trechos de código visando especificidades da arquitetura. Extrair o máximo de desempenho de uma arquitetura para uma aplicação é uma tarefa de importância vital para a indústria e áreas de pesquisa científica, como trabalhos de sequenciamento de DNA, previsão do tempo, simulação de dinâmica de fluidos, entre outros.

Processadores multinúcleo podem possuir diferenças entre seus núcleos componentes: diferenças em frequência, consumo de energia, tamanho e hierarquia de cache, especialidade, ISA e micro-arquitetura podem ser encontradas internamente em muitos tipos de processadores. À essa classe de processadores a classificação de arquitetura heterogênea é atribuída.

2.1.1 Taxonomia de Arquiteturas Paralelas

Flynn e Rudd (1996) desenvolveram uma classificação para arquiteturas de computadores baseados em como eles se comportam levando em consideração dados e instruções. Assumindo que um computador executa um conjunto de instruções em um conjunto de dados e que este computador pode possuir múltiplos fluxos dos mesmos, quatro grandes classes podem ser desenhadas através da combinação dessas características:

SISD – *Single Instruction and Single Data* (uma instrução e um dado, em inglês). Modelo clássico de processamento sequencial, correspondente à arquitetura de Von Neumann, apesar de poder possuir características de processamento concorrente (FLYNN; RUDD, 1996). O modelo de Von Neumann pode ser considerado limitado

em termos de desempenho, uma vez que ele projeta o uso de um único barramento para ler e/ou escrever instruções e dados (BACKUS, 1978);

MIMD – *Multiple Instructions and Multiple Data* (múltiplas instruções e múltiplos dados, em inglês), pode ser considerado o modelo clássico de paralelismo, onde cada PU pode conter seu próprio conjunto de instruções e processá-las concorrentemente a outras PUs em cima de seu próprio conjunto de dados. Este modelo é facilmente encontrado na maioria dos CPUs produzidos, e pode ser identificado em MICs (acrônimo para "muitos núcleos integrados", em inglês) e inclusive GPGPUs (acrônimo para "unidade de processamento gráfico de propósito geral", em inglês) (DIETZ; YOUNG, 2010), assim como pode ser emulado em arquiteturas SIMD (DIETZ; COHEN, 1992);

SIMD – *Single Instruction and Multiple Data* (uma instrução e múltiplos dados, em inglês), um único fluxo de instruções é executado sobre múltiplos fluxos de dados. Este modelo foi popularizado pela adoção de sua implementação em GPGPUs, e pode ser encontrado também em unidades vetoriais dentro de núcleos de arquiteturas de CPU modernas;

MISD – *Multiple Instructions and Single Data* (múltiplas instruções e um dado, em inglês), modelo incomum onde várias instruções são executadas sobre o mesmo conjunto de dados. Há poucas aplicações práticas conhecidas desta classe de arquitetura, como por exemplo o uso de arrays sistólicos para encontrar o maior divisor comum de dois polinômios diferentes de zero (BRENT; KUNG, 1984).

A Ilustração 2.1 mostra um diagrama-exemplo que ilustra conceitualmente as quatro classes descritas. Este trabalho foca em arquiteturas MIMD, que origina problemas de escalonamento por definição, uma vez que cada processador pode realizar tarefas distintas.

2.2 PROCESSADORES MULTINÚCLEO ASSIMÉTRICOS

Quando este trabalho faz referências a AMPs, a definição referenciada é a proposta por Fedorova et al. (2009): Um processador multinúcleo assimétrico consiste de núcleos que possuem o mesmo conjunto de instruções de arquitetura (*Instruction Set Architecture – ISA*) mas entregam desempenhos diferentes e tem características de consumo energético diferentes. É importante enfatizar que AMP está situado em uma subclasse de processamento heterogêneo: todo AMP tem habilidade de processamento heterogêneo por definição, mas nem todos processamento heterogêneo se é realizado por um AMP, como casos de execuções concorrentes entre CPUs e GPUs ou MICs, por exemplo.

Entretanto é comum que a definição ou a nomenclatura de AMP varie de autor para autor. Como constatado por Mittal (2016) em sua pesquisa, apesar do fato que "Processa-

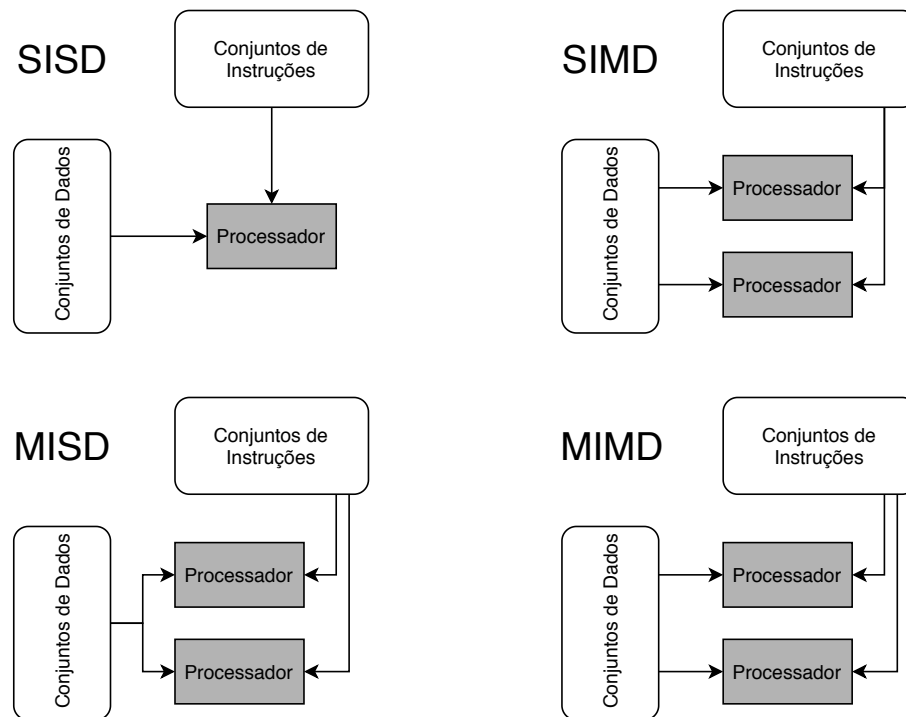


Ilustração 2.1 – Diagrama da Taxonomia de Flynn e Rudd (1996).

Fonte: Adaptado de (FLYNN; RUDD, 1996).

“Multinúcleo Assimétrico” é o termo mais largamente usado para descrever processadores com tipos de núcleos heterogêneos, diversos autores usam terminologias diferentes para descrever AMPs dependendo da origem da assimetria.

2.2.1 ARM big.LITTLE™

A tecnologia ARM big.LITTLE consiste em uma arquitetura de AMPs que foram projetados para atender uma demanda de computação móvel que era guiada por dois fatores (ARM Limited, 2013b): 1) se fazia necessária uma entrega maior de desempenho para suportar aplicações móveis de última geração – que crescem em complexidade e peso computacional – mas sem ultrapassar alguns limites térmicos; 2) um consumo muito baixo de energia, uma vez que esses processadores na maioria dos casos são alimentados com baterias pequenas com tempo de duração de carga limitado.

Essa arquitetura de processamento heterogêneo é composta de dois tipos de núcleo: “LITTLE”, projetados para máxima eficiência energética, e “big”, projetados para prover um maior desempenho computacional. Usualmente esses processadores são coerentes e compartilham do mesmo ISA (ARM Limited, 2013b). Dessa forma uma tarefa pode ser dinamicamente alocada para qualquer dos dois tipos de núcleo dependendo dos requisitos instantâneos da mesma. Diferenças nas micro-arquiteturas internas o permite

prover diferentes taxas de consumo energético e desempenho: os tipos de núcleo podem apresentar essas diferenças em atributos como tamanho e tipo de pipelines, frequência, tamanho de cache L1 e L2, número de núcleos, entre outros. O uso de uma interconexão coerente de cache permite ao sistema garantir a coerência dos dados nas memórias cache e demais dispositivos do sistema sem precisar realizar acessos constantes e lentos à memória principal (Ilustração 2.2).

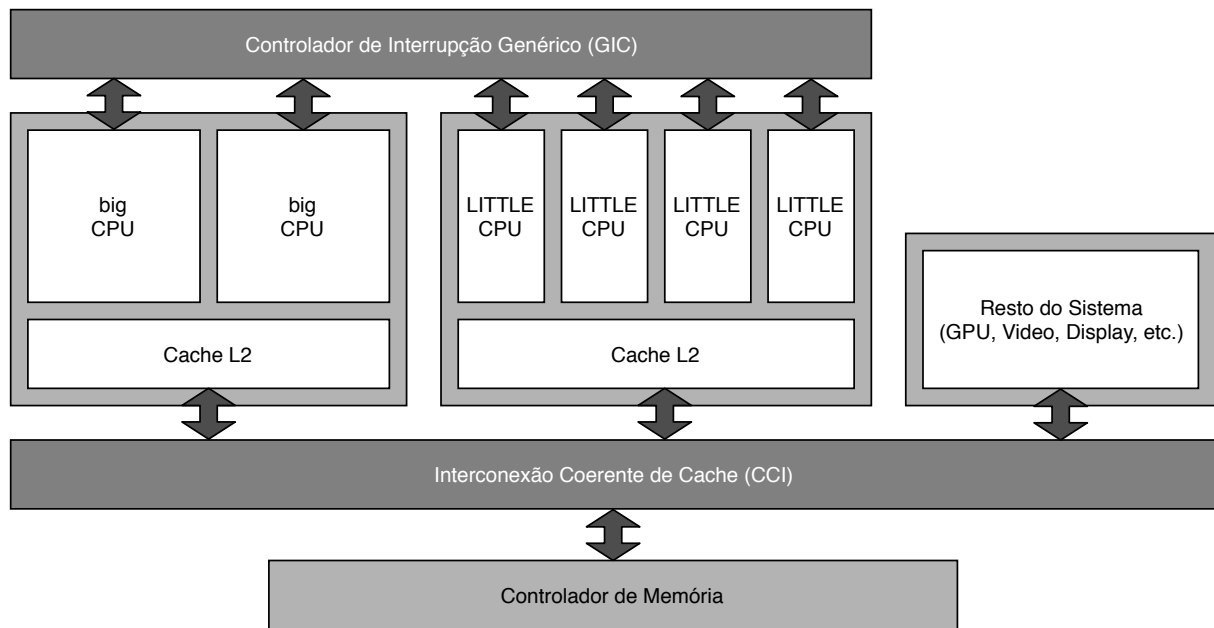


Ilustração 2.2 – Diagrama exemplo de um processador multinúcleo assimétrico: um sistema big.LITTLE típico com 2 núcleos rápidos (big) e 4 núcleos lentos (LITTLE).

Fonte: Adaptado de (ARM Limited, 2013b).

2.2.2 Efeitos da Assimetria no Desempenho

A inclusão de núcleos mais rápidos em sistemas – trazendo consigo assimetria e heterogeneidade – surge com a promessa de, além de possibilitar mais desempenho em tarefas especializadas de cada tipo de núcleo, permitir um ganho teórico de desempenho em aplicações paralelas. Com base nas leis de Amhdal – a fração sequencial de um programa paralelo domina o tempo de execução e limita as vantagens que arquiteturas paralelas podem obter (AMDAHL, 1967) – e de Grosch – o desempenho do computador aumenta conforme o quadrado de seu custo (GROSCH, 1953) –, e baseados em modelos de teoria de filas, Menascé e Almeida (1990) concluem que se é possível extrair mais desempenho de arquiteturas paralelas heterogêneas em aplicações paralelas quando comparadas à arquiteturas paralelas homogêneas de mesmo custo de fabricação. Quão maior for a fração de trabalho sequencial na aplicação, maior a aceleração paralela teórica da arquitetura he-

terogênea em relação à homogênea, devido aos núcleos rápidos que podem ser usados para acelerar trechos de códigos sequenciais em uma aplicação (MENASCÉ; ALMEIDA, 1990; BALAKRISHNAN et al., 2005). O Gráfico 2.1 apresenta o ganho teórico potencial de uma arquitetura heterogênea com um núcleo rápido em relação à uma arquitetura homogênea de mesmo custo, com 1106 núcleos: o ganho de desempenho aumenta juntamente com a porcentagem de trabalho sequencial.

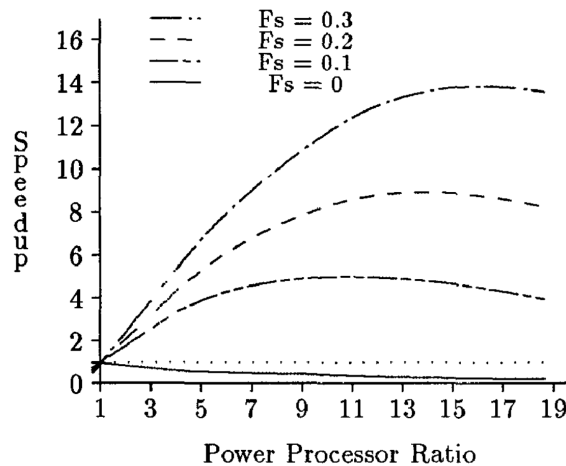


Gráfico 2.1 – Aceleração teórica paralela de uma arquitetura paralela heterogênea em relação à uma homogênea de mesmo custo. O eixo vertical corresponde à aceleração paralela e o eixo horizontal à escala que o núcleo rápido é mais rápido que os núcleos lentos. O rótulo F_s denota diferentes valores de fração de trabalho sequencial.

Fonte: Extraído de (MENASCÉ; ALMEIDA, 1990).

Entretanto tal estimativa leva em consideração um escalonamento ótimo, algo muito difícil de acontecer no mundo real, devido a inúmeras variáveis que podem afetar o desempenho das aplicações (MENASCÉ; ALMEIDA, 1990). Com a premissa que uma assimetria entre processadores pode levar a um desbalanceamento de carga em um sistema paralelo, Balakrishnan et al. (2005) conduziram um estudo que investigou o impacto da assimetria no desempenho de arquiteturas multinúcleo. Os autores avaliaram os tempos de execução de diferentes aplicações em um ambiente com assimetria induzida via alteração de frequências em conjuntos de núcleos de um SMP. Duas questões principais tiveram suas respostas buscadas no estudo: 1) Se a assimetria pode ter um impacto negativo no desempenho, e se o mesmo pode ser previsto em um AMP; 2) Quais métodos podem ajudar aplicações que tenham perdas de desempenho devido à essa assimetria.

Através de avaliações de diferentes tipos de aplicação, que variaram de aplicações comerciais a ferramentas de desenvolvimento, os autores chegaram às seguintes conclusões, dentre outras:

- A assimetria em sistemas afeta a previsibilidade de comportamento de cargas de trabalho, tornando-as menos escaláveis. O efeito aumenta proporcionalmente ao

total de concorrência no sistema;

- Um escalonador ciente da assimetria a nível de sistema operacional pode ajudar algumas aplicações a variarem menos seus tempos de execução, ao realizar migrações de cargas de trabalho para núcleos mais rápidos quando estes estão ociosos. Entretanto, isso não se aplica a aplicações que tem um controle maior do escalonamento de suas tarefas paralelas;

Como ressaltado pelo trabalho, o escalonamento de cargas de trabalho quando realizado internamente pela própria aplicação também tem impacto no desempenho. Dentre as aplicações testadas havia um conjunto de benchmarks (SPEC OMP) que utiliza OpenMP como *back-end*. A maioria desses benchmarks utilizados apresentaram desempenhos contra-intuitivos, como exibido no Gráfico 2.2(a). Os resultados mostram que não há como ter previsibilidade no desempenho, ao passo que um dos benchmarks (*ammp*) apresentou tempos menores para combinações de núcleos que possuíam um maior poder computacional quando comparado a outras combinações. O motivo para esse comportamento é a forma como os benchmarks escalonavam seus laços paralelos, fazendo divisões estáticas de tamanho igual entre os processadores.

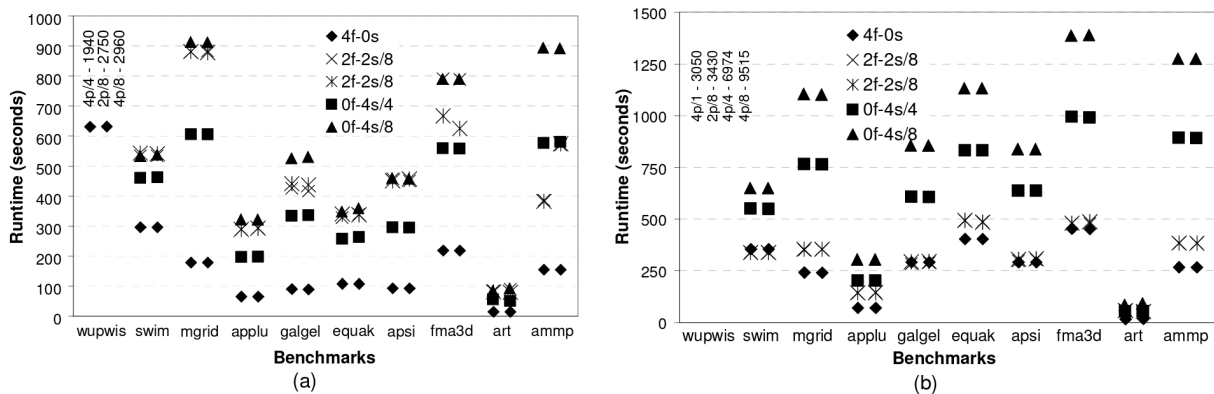


Gráfico 2.2 – Tempo de execução do conjunto de benchmarks SPEC OMP. (a) Tempo com a implementação original do SPEC; (b) Tempo com escalonamento dinâmico. Um rótulo no formato $nf - ms/escala$ denota uma configuração com n núcleos rápidos e m núcleos lentos executando a $1/escala$ da velocidade dos núcleos rápidos. O poder computacional normalizado total para uma configuração é dado por $n + m/escala$. Uma configuração $2f - 2s/8$ possui um poder computacional teórico maior que uma configuração $0f - 4s/4$, mas apresenta tempos de execução maiores para o benchmark *ammp* em (a), devido à má distribuição da carga de trabalho entre as *threads*.

Fonte: Extraído de (BALAKRISHNAN et al., 2005)

Em uma situação dessas, podem existir dois tipos de comportamento para escalonamento estático:

1. Se cada *thread* for associada à um núcleo específico e/ou o sistema operacional não possui um escalonador ciente da assimetria: núcleos lentos tendem a demorar mais

para terminarem seus trabalhos e acabam fazendo com que núcleos rápidos fiquem ociosos esperando, uma vez que os laços paralelos do OpenMP possuem barreiras implícitas no seu final, forçando uma etapa de sincronização global entre as *threads*;

2. Se o sistema operacional é ciente da assimetria e migra todo o trabalho de um núcleo lento para um núcleo rápido que estava ocioso: tal migração tende a reduzir o tempo total de execução, uma vez que o tempo de espera será menor pois o núcleo rápido tender a terminar a tarefa mais rapidamente. Entretanto, tal abordagem ainda não utiliza todo o poder computacional disponível, uma vez que ainda haverão núcleos ociosos – ainda que relativamente mais lentos – esperando pelo término do trabalho de outros núcleos.

O problema com previsibilidade de desempenho foi solucionado ao aplicar um escalonador dinâmico aos laços paralelos, conforme pode ser observado no Gráfico 2.2(b). Entretanto, apesar de eliminar os efeitos da assimetria, tal abordagem acabou aumentando o tempo total de execução. Esse aumento substancial nos tempos pode ser culpa da forma como o escalonador dinâmico do OpenMP trabalha (seção 2.4.1).

2.3 ALGORITMOS ADAPTATIVOS

Este trabalho usa a definição de algoritmos adaptativos criada por Cung et al. (2006), que definem algoritmos adaptativos em função de algoritmos híbridos. De acordo com os autores, por definição, um algoritmo é híbrido quando existe uma escolha em alto nível entre pelo menos dois algoritmos diferentes, ambos com habilidade de resolver o mesmo problema. A escolha visa um aumento de desempenho da execução, dependendo dos dados de entrada/saída e dos recursos computacionais. Um algoritmo híbrido pode ser

simples (*simple*) – $O(1)$ escolhas são realizadas independentemente de características da entrada como tamanho, por exemplo. Embora apenas um número constante de escolhas seja feito, cada escolha pode ser usada um número ilimitado de vezes durante a execução.

barroco (*baroque*) – o número de escolhas não é limitada, depende da entrada.

Embora as escolhas em um algoritmo híbrido simples possam ser definidas estaticamente antes de qualquer execução, algumas opções em algoritmos híbridos barrocos são necessariamente computadas em tempo de execução. As opções podem ser executadas com base nos parâmetros da máquina. Mas existem algoritmos eficientes que não baseiam suas escolhas em tais parâmetros. Distinguem-se as seguintes classes de algoritmos híbridos de acordo com a maneira como as escolhas são computadas:

inconsciente (*oblivious*) – O fluxo de controle do algoritmo não depende nem dos valores particulares das entradas nem das propriedades estáticas dos recursos.

ajustado (*tuned*) – Decisões estratégicas são tomadas com base em recursos estáticos, como parâmetros específicos da memória ou recursos heterogêneos dos processadores em uma computação distribuída. Um algoritmo ajustado é **projetado** (*engineered*) se uma escolha estratégica for inserida com base em uma mistura de análise e conhecimento da máquina de destino e dos padrões de entrada. Um algoritmo híbrido é **auto-ajustado** (*self-tuned*) se as opções são automaticamente computadas por um algoritmo.

adaptativo (*adaptive*) – evita qualquer parametrização específica de máquina ou memória. As decisões estratégicas são tomadas com base na disponibilidade de recursos ou nas propriedades dos dados de entrada, ambas descobertas em tempo de execução (como processadores inativos). Um algoritmo adaptativo é **introspectivo** (*introspective*) se uma decisão estratégica for tomada com base na avaliação do desempenho do algoritmo na entrada fornecida até o ponto de decisão.

A Ilustração 2.3 resume a classificação no formato de um diagrama.

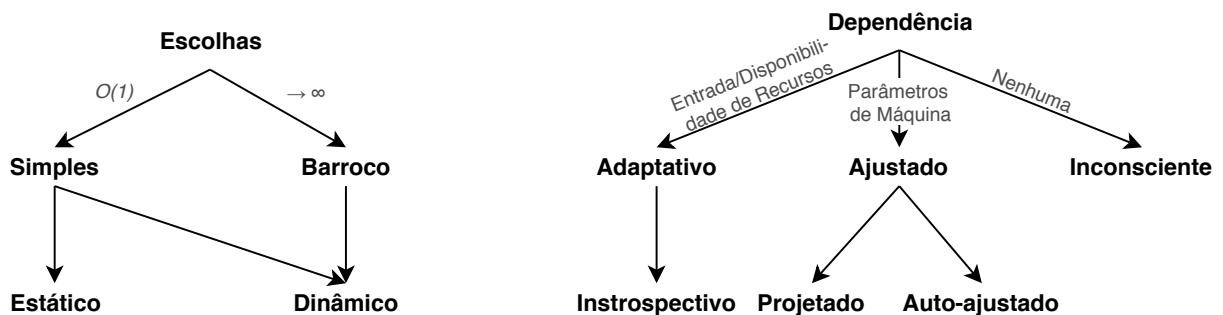


Ilustração 2.3 – Classificação de algoritmos híbridos.

Fonte: Adaptado de (CUNG et al., 2006)

2.4 ESCALONADORES DE LAÇOS PARALELOS

Muitas bibliotecas e APIs para habilitar processamento paralelo já foram implementadas, com diferenças entre si como linguagem de programação suportada, sistemas operacionais suportados, especialidades distintas, hardware específico, etc. Algumas destas bibliotecas oferecem ao desenvolvedor uma forma abstraída de dividir o trabalho de laços de repetição entre diferentes *threads*, com algumas focando em descarregar códigos e dados para PUs externas (OpenMP Architecture Review Board, 2018; EDWARDS; TROTT;

SUNDERLAND, 2014; GREGORY; MILLER, 2012; DURAN et al., 2011; OpenACC Organization, 2019) ou usando em conjunto com a CPU (DOLBEAU; BIHAN; BODIN, 2006), ou de forma distribuída (DIMAKOPOULOS; HADJIDOUKAS, 2011), ou na forma como os dados são acessados (EDWARDS; TROTT; SUNDERLAND, 2014; GREGORY; MILLER, 2012), ou em metodologias diferentes como uso de troca de mensagens (KALÉ; KRISHNAN, 1993) ou de criação de tarefas (OpenMP Architecture Review Board, 2018; Intel Corporation, 2019b; DIMAKOPOULOS; HADJIDOUKAS, 2011; DURAN et al., 2011; Intel Corporation, 2019a). Poucas dessas soluções focam na solução de problemas induzidos pela assimetria de processadores dentro da mesma CPU, com a justificativa de não ser o foco da ferramenta ou na confiança da entrega de desempenho de um escalonador dinâmico simples, como no caso do OpenMP. Este trabalho usará duas soluções existentes a fim de comparar o desempenho da solução proposta: OpenMP, que é o mais próximo de um padrão industrial devido a sua adesão por fabricantes de compiladores, a simplicidade de sua API e por suportar diferentes arquiteturas, e Intel TBB, uma biblioteca que se baseia no uso de tarefas e tem como principal objetivo a distribuição dinâmica de cargas de trabalho de forma eficiente.

2.4.1 OpenMP

OpenMP (do inglês *Open Multi-Processing*, ou Multi-processamento aberto) é uma interface de programação de aplicações (API) aberta que provê um modelo de programação compacto, mas ainda assim flexível, para Fortran, C e C++ disponível para desenvolvedores de aplicações paralelas em memória compartilhada para múltiplas plataformas e sistemas operacionais. Consiste em um conjunto de diretivas de compilador, rotinas de biblioteca e variáveis de ambiente (CHAPMAN; JOST; VAN DER PAS, 2007). Sua especificação fornece formas de paralelizar laços e diferentes escalonadores para tal. Tais escalonadores podem ser especificados através do uso da cláusula `schedule` em combinação com a diretiva de paralelização de laços `for` (Ilustração 2.4).

```

1 #pragma omp parallel for [schedule(<escalonador>[,<grao>]]
2 for (/* parametros do laço */) { /* corpo do laço */ }

```

Ilustração 2.4 – Caso exemplo de paralelização de laço com OpenMP. A cláusula `schedule` é opcional, assim como a especificação de um tamanho para grão.

Fonte: Adaptado de (OpenMP Architecture Review Board, 2018).

A diretiva `parallel` especifica a criação de uma região paralela – um bloco de código que explicitamente terá 1 ou mais *threads* executando em paralelo – e pode ser declarada separadamente ou em combinação com a diretiva `for`, como no exemplo. Em

geral, cada compilador possui uma implementação diferente da API. Mas em praticamente todas as implementações mais recentes há suporte aos três principais tipos de escalonadores: *static*, *dynamic* e *guided*.

static Escalonador estático. Quando especificado, o intervalo de iterações é dividido em t porções de tamanho aproximadamente iguais para t *threads* em uma região paralela OpenMP. Se um tamanho de grão g é especificado, o intervalo é dividido em porções de tamanho g e os mesmos são distribuídos através um algoritmo de round-robin para cada *thread* (OpenMP Architecture Review Board, 2018). Seu uso pode proporcionar bom desempenho em SMPs e laços com carga de trabalho regular, e potenciais percas de desempenho em cenários que distoem desse.

dynamic Escalonador dinâmico. Quando especificado, as iterações são distribuídas dinamicamente em pedaços entre as *threads*. Cada *thread* extrai um pedaço do intervalo, executa, e então requisita outro pedaço, até que não restem mais pedaços a serem processados (OpenMP Architecture Review Board, 2018). As iterações são disponibilizadas internamente em uma fila global centralizada, onde as *threads* conorrentemente extraem porções conforme necessário (DURAND et al., 2013). Cada pedaço contém g iterações conforme especificado pelo programador através do tamanho do grão, exceto pelo último pedaço da fila, que pode conter menos iterações. Quando nenhum tamanho de grão é especificado, o tamanho definido por padrão é 1 (OpenMP Architecture Review Board, 2018). Tal abordagem pode ajudar a alcançar melhores desempenhos em cenários onde o escalonador estático pode se apresentar ineficiente.

guided Escalonador 'guiado'. Quando especificado, as iterações são distribuídas dinamicamente em porções de tamanho variado entre as *threads*. Cada *thread* extrai um pedaço do intervalo, executa, e então requisita outro pedaço, de tamanho menor ou igual ao anterior, até que não restem mais porções, como no escalonador dinâmico (OpenMP Architecture Review Board, 2018). O tamanho do grão é definido por $\lceil r/t \rceil$, onde r é o total de iterações restantes do intervalo original e t é o total de *threads* em execução na região paralela. Esse tamanho consequentemente diminui conforme a execução avança até um tamanho mínimo g , especificado pelo usuário no tamanho do grão. Se nenhum grão for especificado, $g = 1$ por padrão (OpenMP Architecture Review Board, 2018).

Além desta diretiva com três escalonadores distintos, o OpenMP ainda oferece mais uma forma de paralelizar laços através de sua API, com a diretiva `taskloop` (Ilustração 2.5).

Diferentemente da diretiva anterior, a diretiva `taskloop` divide o trabalho do laço em um grupo de tarefas explícitas a serem computadas por cada *thread* na região paralela. Para um intervalo de iterações de tamanho s , a API prevê a criação de k tarefas

```

1  #pragma omp taskloop [num_tasks(qtdTarefas)] [grainsize(grao)]
2  for (/* parametros do laço */) { /* corpo do laço */ }
3

```

Ilustração 2.5 – Caso exemplo de paralelização de laço com a diretiva `taskloop` do OpenMP. A cláusula `num_tasks` é opcional, assim como a especificação de um tamanho para grão na cláusula `grainsize`.

Fonte: Adaptado de (OpenMP Architecture Review Board, 2018).

com tamanho mínimo de grão $g \geq 1$. A quantidade de tarefas e/ou o tamanho do grão mínimo podem ser especificadas através das cláusulas `num_tasks` e `grainsize`, respectivamente. Se nenhuma das cláusulas for especificada o valor padrão para as mesmas é dependente da especificada na implementação do compilador. As implementações providas pelos compiladores GCC e Clang definem o tamanho padrão de grão como 1. A forma como as tarefas são criadas também é dependente da implementação. O compilador GCC cria as tarefas linearmente, enquanto o Clang cria de forma recursiva enquanto o número de tarefas a serem criadas for maior que um limiar, então passando a criar de forma linear. Esse limiar pode ser estipulado pelo usuário através da variável de ambiente `KMP_TASKLOOP_MIN_TASKS`, caso contrário é definido por padrão através de um função heurística. Na versão 8 do compilador Clang essa heurística é dada por $\min(t * 10, 256)$, onde t é o número de *threads* da região paralela.

2.4.2 Intel TBB

Intel TBB (do inglês *Threading Building Blocks*, ou Blocos de Construção de *Threads*) é uma biblioteca C++ para programação paralela que oferece um conjunto de métodos e *templates* C++ para a criação de aplicações paralelas. Com foco no uso de tarefas ao invés de *threads* explícitas, o TBB possibilita alcançar bons desempenhos e escalabilidade ao dinamicamente redistribuir essas tarefas entre PUs (CONTRERAS; MARTONOSI, 2008).

Laços paralelos são suportados no TBB através dos métodos `tbb::parallel_for` e `tbb::parallel_reduce`, com o último possuindo suporte para operações de redução. A Ilustração 2.6 mostra as assinaturas de métodos que permitem a paralelização de laços.

O Intel TBB possui 4 tipos de escalonadores – que a biblioteca denomina como *partitioners* (particionadores, em inglês) – para laços paralelos (Intel Corporation, 2019b):

`tbb::auto_partitioner` Escalonador automático. Esse escalonador adapta os intervalos de iterações de acordo com os recursos disponíveis em tempo de execução. O *worker* (trabalhador) mestre cria uma tarefa com todo o intervalo de iterações. Esta

```

1 // limites do intervalo com duas variaveis
2 tbb::parallel_for(inicio , fim , corpo_laco [, escalonador]);
3
4 // limites do intervalo atraves de um tbb::blocked_range
5 tbb::parallel_for(tbb::blocked_range<tipo>(inicio , fim [, grao]), corpo_laco [,
6     escalonador]);
7
8 // reducao paralela
9 valor = tbb::parallel_reduce(tbb::blocked_range<tipo>(inicio , fim [, grao]) ,
10     identidade , corpo_laco , corpo_reducao [, escalonador]);

```

Ilustração 2.6 – Caso exemplo de paralelização de laço com biblioteca Intel TBB. *inicio* e *fim* denotam o início e o final do intervalo de iterações e possuem tipo *tipo*. *grao* é um valor para cálculo de grão. *corpo_laco* representa o bloco de código interior de um laço a ser executado em paralelo, pode ser uma função comum ou uma função *lambda*, assim como *corpo_reducao*. *escalonador* representa um objeto de escalonador que pode ser opcionalmente passado para as funções. Por fim, *identidade* representa o valor inicial de um dado a ser reduzido, *valor* o valor final, e o tipo *tbb::blocked_range* uma classe do TBB para manipulação de intervalos em uma ou mais dimensões.

Fonte: Adaptado de (Intel Corporation, 2019b).

tarefa então subdivide-se recursivamente criando novas tarefas até uma profundidade máxima D , empilhando-as na deque local de tarefas do *worker* e dividindo o intervalo pela metade a cada recursão. Quando o *worker* consome por completo uma tarefa, ele retira uma nova tarefa do final de seu deque e subdivide-a até a profundidade máxima se necessário. O comportamento é similar a um algoritmo de busca em profundidade.

Um *worker*, quando ocioso e com o deque vazio, escolhe aleatoriamente uma vítima e rouba sua tarefa do início do deque de tarefas, que geralmente contém a primeira metade do trabalho da vítima que ainda não fora dividido. O ladrão então subdivide-o até uma profundidade $D + 1$ (com exceção do primeiro roubo de cada *worker*, onde não há um acréscimo na profundidade). O acréscimo na profundidade é refletido no *worker* vitima, que também acaba aumentando sua profundidade máxima em 1.

A profundidade D é determinada pela fórmula $D = 5 + k$ com $k = \lfloor \log_2 N \rfloor - 1$ sendo $k \geq 0$ e N o número de *threads*. Caso um grão não seja especificado, o tamanho da porção de trabalho mínima é 1. Entretanto é possível especificar um valor c para o tamanho mínimo, por meio de um objeto *tbb::blocked_range*, sendo que cada porção de trabalho sequencial g será tal que $g \geq c/2$. Quando um escalonador não é especificado, este é utilizado por padrão¹;

tbb::simple_partitioner Escalonador 'simples'. A forma de divisão de trabalho entre *threads* também é recursiva, e também há roubo de trabalho, entretanto não há um

¹padrão desde a versão 2.2 da biblioteca

limite de profundidade como no escalonador automático. Apesar disso, o escalonador respeita o tamanho do grão: para um grão g , a porção de trabalho sequencial c será tal que $g/2 \leq c \leq g$. Quando um tamanho de grão não é especificado (através do uso de um objeto `tbb::blocked_range`), o tamanho do grão por padrão é determinado como 1. Sua implementação é similar à implementação do framework CilkPlus² (depreciado), também desenvolvido pela Intel, com a diferença que o CilkPlus adotava uma abordagem *work-first* (quando uma tarefa é criada, o *worker* suspende e coloca a tarefa corrente na pilha de tarefas para então executar a nova tarefa), diferente da abordagem *help-first* (coloca imediatamente a tarefa nova na pilha de tarefas) adotada pelo Intel TBB;

`tbb::static_partitioner` Escalonador estático. Praticamente igual ao escalonador estático do OpenMP. Busca dividir o intervalo o mais igualmente possível entre todas as *threads*, mas respeitando a fórmula $c \geq \max(g/3, n/t)$, onde n é o tamanho do intervalo original e t o número de *threads* TBB.

`tbb::affinity_partitioner` Escalonador com respeito à afinidade. Em suma idêntico ao escalonador automático, com a diferença que o histórico de atribuições de sub-intervalos à *threads* é salvo no objeto do particionador para ser reutilizado posteriormente, buscando um melhor aproveitamento da cache.

2.5 TRABALHOS RELACIONADOS

Alguns trabalhos foram conduzidos para desenvolver escalonadores adaptativos em diferentes contextos. Esta seção apresenta trabalhos encontrados na literatura que tomam uma direção similar ao foco deste trabalho, mas que eventualmente divergem, seja por não se dirigirem a AMPs ou sistemas heterogêneos, ou seja por não focarem na paralelização de laços.

Menon (2016) apresentou diferentes algoritmos adaptativos para balanceamento de carga inter e intra-nodo, para laços paralelos e tarefas com dependência. Inicialmente ele apresenta a GrapevineLB, uma estratégia para balanceamento de carga de laços paralelos completamente distribuída. Ela baseia o balanceamento sobre uma representação parcial do estado global do sistema. Possui dois estágios: propagação e transferência. No estágio de propagação, informações de carga de trabalho sobre PUs subcarregadas são propagadas para PUs sobrecarregadas através de um algoritmo epidêmico, selecionando aleatoriamente outras PUs a receberem e repassarem a informação a diante. A etapa

²<www.cilkplus.com>

de propagação pode ter um número limite de rodadas, ou um controle de quem já recebeu a informação. Na etapa de transferência, processadores subcarregados a receberem carga podem ser escolhidos aleatoriamente, ou aleatoriamente com uso de uma função de distribuição, onde processadores subcarregados mais próximos da média de carga do sistema tem menor probabilidade de receberem transferências de carga. Uma modificação, que os autores nomearam de Grapevine+, dá a habilidade à uma PU subcarregada de rejeitar trabalho caso perceba que ficará sobrecarregada. A forma de determinar se uma PU está sobre ou subcarregada é através de uma comparação de sua carga com a carga média. Entretanto, o autor não apresenta como esta média é obtida ou estipulada. Ambas as estratégias, implementadas em Charm++, apresentaram tempos menores que outras estratégias em ambientes homogêneos distribuídos de 4096 a 131072 unidades de processamento. O autor também propõe outro balanceador de carga para ambientes distribuídos, mas com foco na criação de tarefas locais, tirando a exclusividade da divisão em processos e comunicação por meio de troca de mensagens da proposta anterior. Nessa abordagem o intervalo de iterações é inicialmente dividido igualmente entre os nodos no sistema distribuído e, dentro de cada nodo, dividido recursivamente com a criação de tarefas paralelas, em uma abordagem muito similar à do framework CilkPlus. Para esta abordagem foram usadas diferentes implementações – dependendo da aplicação utilizada para testes – combinando bibliotecas como Charm++, OpenMP, MPI e Adaptive MPI. Com um foco maior no balanceamento de carga interno de cada nodo, essa abordagem conseguiu menores tempos de execução que as implementações originais de cada aplicação, conseguindo acelerações paralelas maiores que 2 vezes. O autor apresenta essa abordagem como uma boa alternativa para ambientes heterogêneos ou aplicações com maiores desbalanceamentos de carga. Apesar da menção à ambientes heterogêneos, todas as avaliações desta estratégia e da estratégia anterior visaram somente ambientes homogêneos distribuídos: o trabalho não apresenta resultados sobre o impacto de sua abordagem em ambientes heterogêneos ou com heterogeneidade induzida (p.ex. desativação de alguns núcleos, alteração de frequência via DVFS, entre outros). Além disso, há uma preocupação por parte do autor com o sobrecusto da criação excessiva de tarefas durante a decomposição do problema. Algumas medidas são tomadas para prevenção (retenção na criação de tarefas com base na ocupação das PUs), mas o impacto das mesmas não é avaliado no trabalho.

Meadows e Ishikawa (2017) trabalharam na otimização do CCS-QCD – uma aplicação Stencil 3D – usando tarefas OpenMP para multiplexar computações com comunicações MPI. A aplicação originalmente definia *threads*/núcleos exclusivos para gerenciar as comunicações MPI, mas essa abordagem acarretava na sub-utilização desses núcleos na maior parte do tempo relacionado à computação necessária dos dados. Eles decidiram implementar um versão alternativa do escalonador de laço de forma similar à diretiva `taskloop` do OpenMP – dividindo blocos de iterações em tarefas –, uma vez que a pró-

pria implementação da diretiva não estava otimizada na versão do compilador utilizado no trabalho (Intel C++ Compiler XE 17.0.2.174). A implementação é também equivalente à implementação do framework CilkPlus. Essa abordagem mostrou que o escalonador de tarefas padrão do OpenMP do compilador utilizado consumia em média 10% do tempo de execução somente com escalonamento. Os autores então decidiram implementar seu próprio escalonador de tarefas, chamado *untasking*, com a biblioteca Intel TBB, onde cada thread mantém sua própria fila de iterações. O laço principal para escalonamento de tarefas de um *thread* desenfileira tarefas (blocos de iterações) da fila da *thread* até a fila estar vazia, então busca nas filas das outras *threads* uma tarefa para roubar. Os autores não entram em detalhes sobre as políticas de roubo aplicadas no escalonador. A implementação final do *untasking* exibiu uma melhoria de desempenho da ordem de 1.6x quando comparado à implementação original do CCS-QCD, e 1.2x à implementação alternativa ao `taskloop`. O CPU utilizado na avaliação foi um Intel Xeon Phi 7250 com 68 núcleos. O objetivo do trabalho era muito específico – otimizar uma única aplicação – o que fez a abordagem final convergir em uma solução muito específica para a aplicação, ainda que alguns conceitos podem ser extraídos do projeto do escalonador.

Em busca de realizar um balanceamento dinâmico de carga ao mesmo tempo em que leva afinidade de memória em conta em arquiteturas NUMA (*Non-Uniform Memory Access*, Acesso Não-Uniforme à Memória), Durand et al. (2013) trabalharam em um escalonador de laços que visava prover meios de balancear a carga de laços irregulares respeitando localidade de memória, uma fonte comum de problemas de desempenho em arquiteturas NUMA. Os autores estenderam o escalonador para respeitar a localidade de memória ao definirem uma ordem de prioridade para selecionar vítimas em um roubo de trabalho: o escalonador faz os *workers* localizados em um mesmo banco de memória trabalharem em iterações contíguas entre si. Quando necessário, *workers* ociosos roubam trabalho de *workers* em núcleos que pertençam ao mesmo nó NUMA. Se não há trabalho restante para ser roubado no nó NUMA local, o *worker* ocioso emite uma requisição de roubo para uma vítima selecionada aleatoriamente no escopo inteiro da máquina. A *runtime* libGOMP foi estendida para suportar os escalonadores propostos diretamente na cláusula `schedule` do OpenMP. O desempenho do escalonador proposto foi avaliado em quatro *benchmarks* diferentes e alcançou uma aceleração paralela de 2.35 sobre o escalonador estático do OpenMP, ainda que gaste relativamente mais tempo com sobrecustos do escalonador.

Mór (2015) desenvolveu uma aproximação da primitiva originalmente adaptativa `cilk_for` da biblioteca CilkPlus para processar paralelamente estruturas de dados similares à listas. Um *worker* recebe um *iterator* referente a um elemento da lista ao invés de um índice do laço. O algoritmo é adaptativo e muito similar ao algoritmo proposto por Meadows e Ishikawa (2017). O autor mescla o algoritmo adaptativo com um algoritmo de avaliação polinomial e simula sua execução em dois tipos diferentes de estratégia de

seleção de vítima para roubo de trabalho: aleatória e *clock* mínimo (um *worker* ocioso seleciona um *worker* ativo com o menor valor de *clock*). As simulações mostraram que o número de roubos bem-sucedidos necessários é menor para a estratégia de *clock* mínimo, uma vez que um *worker* pode acabar realizando roubos mais significativos – i.e. roubar de *workers* que tenham mais trabalho restante. O objetivo do trabalho não era avaliar o desempenho do algoritmo de escalonamento em qualquer tipo de ambiente, apesar do número de roubos estar diretamente relacionado à sobrecusto paralelo, e conseqüentemente ao desempenho como um todo.

Traoré et al. (2008) apresentou um escalonador paralelo adaptativo para algoritmos STL, com roubo de trabalho. A implementação usa uma lista distribuída por *worker*, ao invés de uma *deque* (fila duplamente terminada), a fim de limitar o sobrecusto com criações de tarefas. O escalonador também possui uma etapa de mescla para algoritmos que necessitem de operações pós-computações, como reduções. O algoritmo *deque-free* é composto de um micro-laço (o curso dos ladrões de acordo com a ordem sequencial do trabalho roubado) e um nano-laço (a computação do trabalho). Roubos são feitos recursivamente, confiando a seleção de vítimas à *runtime* de tarefas. O escalonador foi implementado usando a *runtime* Kaapi e comparado com versões do Intel TBB e MCSTL, apresentando melhores tempos médios quando comparado com as outras opções na maioria dos casos.

Todos os trabalhos mencionados até aqui não exploraram e/ou relataram o desempenho e o comportamento de suas soluções em ambientes assimétricos.

Chronaki et al. (2017) propuseram duas abordagens de escalonamento de tarefas que visavam sistemas com processamento assimétrico: CPATH e HYBRID. Essas políticas de escalonamento dinâmicas reduzem o tempo total de execução tanto ao detectar o caminho maior ou crítico do grafo dinâmico de dependências de tarefas, quanto ao encontrar o executor mais recente de uma tarefa. A implementação do CPATH separa tarefas em ordem de prioridade (críticas ou não-críticas) e atribui tarefas com maior prioridade para núcleos mais rápidos. Ele também usa tempos de execução de tarefas previamente conhecidos para determinar heurísticamente caminhos críticos no grafo de dependências de tarefas. A implementação HYBRID é derivada da CPATH, mas somente leva tempos de execução em consideração se eles estiverem disponíveis, possibilitando menores sobrecustos que o CPATH. Ambos os escalonadores foram implementados usando o modelo de programação OmpSs e tiveram seus desempenhos comparados com dois escalonadores existentes (CATS e HEFT) e o algoritmo de busca em largura. A avaliação de desempenho foi executada em uma placa ODROID-XU3 (veja seção 3.7.2) e em ambientes simulados com um número total de 16 e 32 núcleos, variando o número de núcleos rápidos. Os resultados mostraram que o escalonador HYBRID alcançou um desempenho equivalente ao escalonador CATS, sendo um pouco melhor em um *benchmark* de decomposição QR.

Apesar de visar melhorar desempenho em processadores assimétricos, o trabalho foca no desenvolvimento de escalonadores de tarefas, enquanto este trabalho foca em es-

calonadores de laços paralelos. Uma tarefa, na abordagem do autor, é uma unidade de trabalho computacional que é substancialmente diferente de um laço paralelo: tarefas normalmente não podem ser sub-divididas, e podem possuir atributos como prioridades que as diferem entre si. Os escalonadores propostos pelo trabalho fazem uso de heurísticas baseadas em históricos de tempo de execução, o que não é uma abordagem vantajosa para laços paralelos, uma vez que manter históricos de tempo de execução por iteração ou blocos de iterações pode ser extremamente custoso em termos de uso de memória para a *runtime*. Além disso, a implementação não pode ser classificada como adaptativa, mas como ajustada (veja seção 2.3), desde que o algoritmo conhece características específicas dos núcleos – i.e. o algoritmo sabe se o núcleo é rápido/big ou lento/LITTLE.

2.6 SUMÁRIO

Este Capítulo apresentou uma breve introdução sobre arquiteturas paralelas e sua taxonomia, seguida da definição de processadores multinúcleo assimétricos, um exemplo existente encontrado atualmente no mercado (ARM big.LITTLE) e o impacto potencialmente negativo da assimetria induzida pelos mesmos no desempenho de aplicações. É apresentada a definição de algoritmos adaptativos e exemplos de bibliotecas referentes ao estado da arte para escalonamento de laços paralelos, descrevendo escalonadores existentes em implementações do padrão OpenMP e na biblioteca Intel TBB. Ao final, trabalhos relacionados aos temas escalonamento adaptativo e escalonamento em AMPs são apresentados, e suas diferenças em relação ao trabalho proposto são ressaltadas.

Os tópicos abordados neste Capítulo ajudam a ilustrar a necessidade de uma solução para distribuição da carga computacional de laços paralelos em AMPs, tema não abordado em sua integridade pela literatura existente. O Capítulo a seguir apresenta a proposta e detalhes da implementação de um escalonador de laços paralelos que distribua a carga dos mesmos de forma que vise se adaptar melhor a ambientes que possuam capacidade de processamento assimétrica.

3 ESCALONADOR ADAPTATIVO

Este Capítulo apresenta a proposta de um escalonador que permite adaptar a carga de trabalho de um laço paralelo em AMPs. As seções a seguir apresentam detalhes da construção interna do escalonador, bem como seus algoritmos de extração de trabalho e como sua implementação foi conduzida através do uso da linguagem de programação C++ e do paradigma de orientação à objetos.

3.1 LAÇO PRINCIPAL DO ESCALONADOR

Ao usar uma aproximação derivada de (DURAND et al., 2013), um escalonador adaptativo seguiria três etapas maiores para computar um laço paralelo:

1. Divide o intervalo original de iterações entre as *threads* em sub-intervalos;
2. Cada *thread* extrai sequencialmente uma porção de trabalho de seu sub-intervalo;
3. Quando sem trabalho, uma *thread* procura uma *thread* vítima com trabalho restante e rouba uma porção de trabalho para si, retornando à etapa 2. Se não poder roubar trabalho, encerra sua execução.

Como cada aplicação pode conter um número arbitrário de laços paralelos, essas etapas seriam repetidas a cada invocação de laço. A partir de um ponto de vista externo, esses passos podem ser interpretados como o laço principal de execução de uma ferramenta de escalonamento de laços paralelos. Usando esses passos como pilares sólidos, a construção de um escalonador pode ser proposta.

O escalonador proposto por este trabalho é composto por, além de alguns métodos para invocação de laços paralelos, um *Coordenador* responsável por instanciar, lançar e coordenar as *threads* utilizadas pelo escalonador. Durante a execução da aplicação, as *threads* criadas ficam a espera de *Trabalhadores* contendo a descrição do trabalho a ser realizado.

Um Trabalhador representa uma entidade responsável por computar uma parte do trabalho, sendo necessariamente associado à uma *thread* específica. Quando um método para criação de um laço paralelo é invocado, as informações referentes ao laço (intervalo de iteração e corpo do laço) são inseridas em Trabalhadores instanciados pelo método, e os mesmos são posteriormente transmitidos para o Coordenador, para que o mesmo possa invocar métodos de inicialização dos Trabalhadores e dar início à execução do laço.

Um Trabalhador é composto basicamente de uma trava de software e dois sub-intervalos de iterações:

- Um sub-intervalo maior, que representa a parte referente à este Trabalhador/*thread* do intervalo original do laço a ser paralelizado. Seus limites inferior e superior são delimitados pelas variáveis membro `first` e `last`, respectivamente;
- Um sub-intervalo menor, que representa um porção de trabalho, extraída do sub-intervalo maior, para ser processada de forma sequencial pela *thread* a qual o Trabalhador está vinculado. Essa extração deve, por definição, ser o mais livre de travas quanto for possível, de forma a eliminar sobrecustos paralelos com o processo de extração. Seus limites inferior e superior são delimitados pelas variáveis membro `chunk_begin` e `chunk_end`, respectivamente.

Cada *thread* possui um ciclo de vida. Quando a aplicação é lançada, as *threads* são iniciadas e ficam à espera de trabalho. Quando um trabalho é recebido através de um Trabalhador, as *threads* dão início ao laço principal de cada Trabalhador, que consomem o trabalho através de extrações sequenciais sobre o mesmo, até o momento que o trabalho local ao Trabalhador termine, fazendo com que cada Trabalhador passe a tentar roubar trabalho (extração paralela) de outros Trabalhadores. Se não há trabalho restante, a *thread* descarta o Trabalhador e espera pela *thread* mestre sinalizar se há um novo trabalho (fornecendo novos Trabalhadores, que representam um novo laço paralelo) ou se ela deve encerrar seu ciclo de vida (geralmente ao final da execução da aplicação).

A escolha entre os dois tipos de extração de trabalho (sequencial e paralela) configuram o algoritmo referente ao laço principal do escalonador como um algoritmo híbrido. Com base em um algoritmo de roubo de trabalho, o escalonador pode permitir um melhor escalonamento da carga computacional de um laço paralelo: redistribuindo adaptativamente de forma dinâmica porções do intervalo de iterações referente a um laço entre PUs com diferentes desempenhos, visando diminuir ao máximo o tempo ocioso de cada PU do sistema, e sem realizar distinções entre diferentes PUs com diferentes desempenhos, tornando o algoritmo adaptativo por definição.

A Ilustração 3.1 ilustra o ciclo de vida de uma *thread* típica, enquanto a Ilustração 3.2 apresenta um exemplo visual de como uma extração sequencial ocorre concorrentemente à uma extração paralela (roubo de trabalho) em um mesmo sub-intervalo;

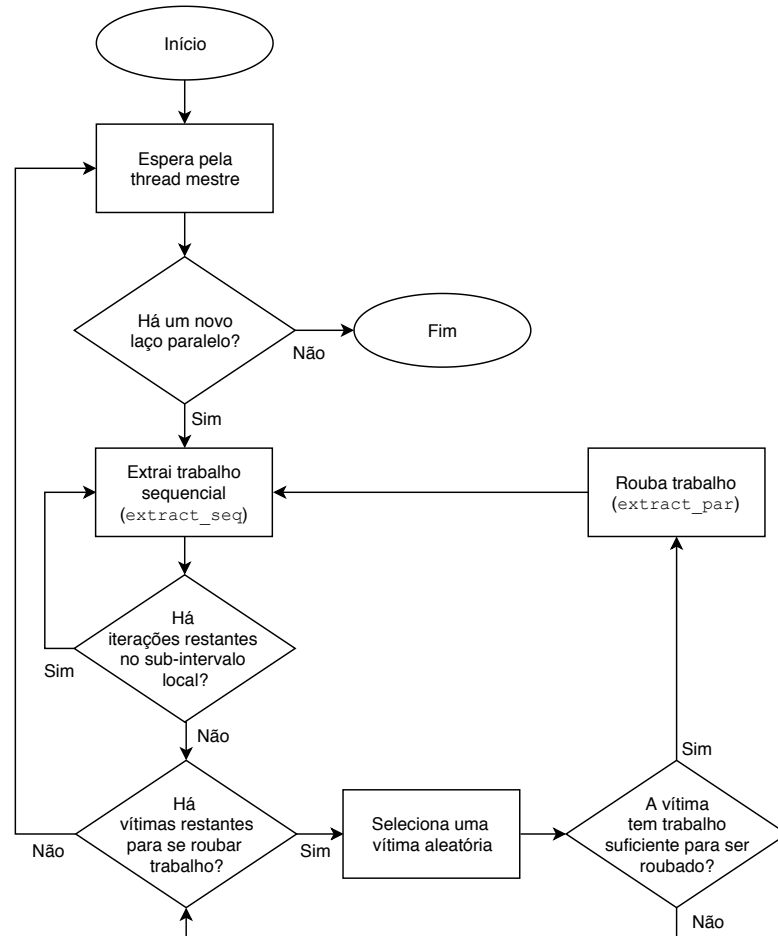


Ilustração 3.1 – Fluxograma do ciclo de vida de uma *thread*.

Fonte: O autor.



Ilustração 3.2 – Caso exemplo de extração de trabalho em um sub-intervalo de tamanho m . São ilustrados três estados subsequentes: (a) k porções de trabalho de tamanho `chunk_size` já foram extraídos pela *thread* dona do sub-intervalo; (b) Uma outra *thread* rouba uma determinada quantia do sub-intervalo (no exemplo, metade do que resta); (c) A *thread* extrai mais `chunk_size` iterações do seu sub-intervalo.

Fonte: Adaptado de (MÓR, 2015).

3.2 ALGORITMOS DE EXTRAÇÃO SEQUENCIAL E PARALELA

Como informado na seção anterior, o escalonador proposto escolhe em tempo de execução entre dois métodos de extração para extrair trabalho para cada Trabalhador: uma extração sequencial e uma extração paralela. Esses métodos consistem em uma adaptação do protocolo THE do *framework* Cilk (FRIGO; LEISERSON; RANDALL, 1998) para uso com intervalos de iterações.

O algoritmo presente na Ilustração 3.3 apresenta o processo de extração de trabalho sequencial. Através de sucessivas chamadas ao método `EXTRACT_SEQ`, um Trabalhador extrai trabalho sequencial de seu sub-intervalo maior (delimitado por `first` e `last`). A porção de trabalho extraída corresponde a um total de iterações determinado por `chunk_size` e delimitado por `chunk_begin` e `chunk_end`.

Ilustração 3.3 – Escalonador Adaptativo – Algoritmo de Extração de Trabalho Sequencial

```

1: function EXTRACT_SEQ
2:   old_first ← first
3:   chunk_begin ← MIN(first + chunk_size, last)
4:   if chunk_begin > chunk_end then
5:     first ← chunk_begin
6:     if chunk_begin < last then                                ▷ Confirma se a extração é válida
7:       chunk_end ← first
8:       chunk_begin ← old_first
9:       return TRUE
10:    end if
11:    first ← old_first                                          ▷ Conflito: desfaz e trava a si próprio
12:  end if
13:  LOCK(self)
14:  chunk_begin ← first
15:  if chunk_begin < last then
16:    first ← chunk_end ← last
17:  end if
18:  UNLOCK(self)
19:  return chunk_begin < first
20: end function

```

Fonte: O autor.

São realizados dois testes para validar a extração: (1) (linha 4) se o novo limite inferior do sub-intervalo maior (temporariamente em `chunk_begin`) é maior que o limite superior antigo da última porção de trabalho extraída (`chunk_end`), e (2) (linha 6) se o novo limite inferior do sub-intervalo maior (ainda em `chunk_begin`) é menor que o limite superior do sub-intervalo maior (`last`).

Caso um conflito seja detectado durante a extração de trabalho sequencial – isto

é, um Trabalhador concorrente rouba uma parte da porção de trabalho que seria extraída – o Trabalhador desfaz as modificações parciais (linha 11) e trava a si próprio (linha 13). Uma vez travado, o seu sub-intervalo maior não pode ser modificado concorrentemente por nenhum outro Trabalhador. A extração então é realizada e o método `EXTRACT_SEQ` retorna um valor lógico indicando se uma porção de trabalho de tamanho válido foi obtida (linha 19). Essa situação só acontecerá uma vez por sub-intervalo maior: caso um conflito seja detectado e/ou seja a última extração do sub-intervalo maior. Um sub-intervalo roubado é considerado um novo sub-intervalo maior. A ilustração 3.4 mostra um exemplo de conflito durante o processo de extração.

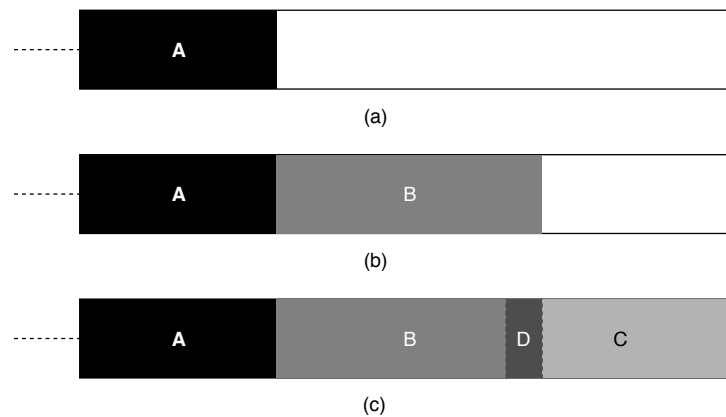


Ilustração 3.4 – Caso exemplo de um conflito durante extração de trabalho. São ilustrados três estados subsequentes: (a) O Trabalhador dono do sub-intervalo já havia extraído A iterações; (b) O Trabalhador dono do sub-intervalo se prepara para extrair outras B iterações; (c) Concorrentemente, um outro Trabalhador tenta roubar C iterações, entretanto o intervalo representado por C intersecciona com B, gerando uma região D que pode ter trabalho duplicado caso o conflito não seja tratado.

Fonte: O autor.

A extração visa ser o mais livre de travas o quanto possível e, para tal, definiu-se uma ordem de operações para extrair e, concorrentemente, roubar um bloco contíguo de iterações de um sub-intervalo. As linhas 4 e 5 do algoritmo presente na Ilustração 3.3 e as linhas 8 e 9 do algoritmo na Ilustração 3.5 são o ponto-chave para tal: a ordem das instruções assegura a corretude das operações em ambientes com consistência de memória sequencial. Entretanto, a maioria das arquiteturas paralelas modernas não suportam consistência de memória sequencial porque a adesão à seu suporte pode resultar em uma diminuição geral de desempenho, uma vez que requer a execução sequencial de instruções relacionadas à memória (GEORGOPOULOS, 2016). Desse modo, para assegurar a corretude do algoritmo, se faz necessária a inserção de barreiras de memória entre esses pares de linha. Nesta implementação, foi definido o uso de tipos atômicos `atomic` da linguagem C++ para os membros `first` e `last`, pois são projetados para não causarem condição de corrida em dados e podem ser usados para sincronizar acessos à memória entre *threads* com um custo computacional menor que travas tradicionais (ISO, 2013).

O Algoritmo 3.5 ilustra como um roubo de trabalho é efetuado. Um Trabalhador ladrão segue quatro passos para roubar trabalho de outro Trabalhador, quando necessário:

1. Ele escolhe aleatoriamente uma vítima *victim* (linha 3), testa se a mesma possui um sub-intervalo válido e tenta travá-la (linha 4) para que, caso consiga, nenhum outro Trabalhador possa roubá-la enquanto ela não for liberada;
2. O tamanho do roubo (*steal_size*) é determinado de acordo com o tamanho do sub-intervalo da vítima (método `GET_STEAL_SIZE`) (linha 5);
3. São feitos testes para verificar se o valor referente ao novo início do sub-intervalo (*new_first*, variável temporária) está dentro do sub-intervalo atual da vítima (linhas 7 e 9);
4. o roubo é efetuado (linhas 8 e 12–15).

Ilustração 3.5 – Escalonador Adaptativo – Algoritmo de Extração de Trabalho Paralelo (Roubo de Trabalho). Todas as variáveis precedidas por *victim* representam membros do Trabalhador vítima.

```

1: function EXTRACT_PAR
2:   while CAN_STEAL(workers) do
3:     victim ← SELECT_VICTIM(workers)
4:     if victim.last > victim.first AND CAN_LOCK(victim) then
5:       steal_size ← GET_STEAL_SIZE(victim)
6:       new_first ← victim.last – steal_size
7:       if victim.last > new_first AND victim.first ≤ new_first then
8:         victim.last ← new_first
9:         if victim.first > victim.last then
10:          victim.last ← new_first + steal_size           ▷ Conflito: desfaz roubo
11:        else if steal_size > 0 then
12:          first ← new_first
13:          last ← new_first + steal_size
14:          UNLOCK(victim)
15:          return TRUE                                     ▷ Roubo bem sucedido
16:        end if
17:        UNLOCK(victim)
18:      end if
19:    end if
20:  end while
21:  return FALSE                                         ▷ Não consegue roubar: encerra execução
22: end function

```

Fonte: O autor.

O ladrão não espera pela liberação da trava da vítima (linha 4) com o objetivo de não ficar ocioso. Se a trava não estiver disponível, ele então seleciona outra vítima. O

número de vezes que o método `EXTRACT_PAR` é invocado determina o sobrecusto paralelo que o escalonador introduzirá. Esse sobrecusto paralelo faz-se inexistente em execuções com somente uma *thread*, uma vez que toda concorrência é eliminada. Caso um conflito durante a extração seja detectado, a ação é desfeita (linha 10) e o ladrão desiste da vítima, liberando sua trava (linha 17). O processo se repete até que o algoritmo identifique que nenhum outro Trabalhador tenha trabalho restante para roubar (linha 2).

3.3 OTIMIZAÇÃO

Esta abordagem de escalonamento descentraliza a distribuição de carga de trabalho e reduz a concorrência paralela total quando em comparação com o escalonador dinâmico do OpenMP. Além disso, este trabalho propõe uma forma de extração sequencial das porções de trabalho o mais livre de travas quanto possível. Pode-se variar os seguintes parâmetros neste escalonador em busca de tempos ótimos:

Tamanho do sub-intervalo inicial de cada *thread*: Algumas abordagens como a do framework CilkPlus e da biblioteca Intel TBB (escalonador automático e simples) apostam em uma concentração inicial do intervalo de iterações em uma *thread*, delegando a distribuição da carga inicialmente para os algoritmos de roubo de trabalho. Experimentos preliminares apontam que essa abordagem não é muito vantajosa quando combinada com o escalonador proposto, trazendo um aumento no tempo de escalonamento: as *threads* precisam sincronizar o início de sua execução para garantir que todas tenham acesso à mesma função (leia-se corpo de laço paralelo) e intervalo de iterações, tornando redundante o custo com roubo de trabalho no início da execução. O escalonador implementado divide de forma igualitária o intervalo original entre as *threads*, de forma similar à como os escalonadores estáticos do OpenMP e TBB fazem.

Tamanho da porção de trabalho sequencial: Este trabalho propõe três abordagens diferentes para definição do tamanho das porções de trabalho a serem extraídas para processamento sequencial:

1. fixa – o número de iterações a serem processadas por vez é sempre fixo;
2. fracionada – a quantia de iterações é representada por uma fração do tamanho original do sub-intervalo de cada Trabalhador. Tal abordagem garante que cada sub-intervalo terá uma quantia máxima constante de porções a serem extraídas;
3. logarítmica – calcula-se o logaritmo na base 2 do tamanho do sub-intervalo original de cada Trabalhador. O uso de um valor logarítmico permite que o número

de porções de trabalho a serem extraídas aumentem junto com o tamanho da porção conforme o tamanho do sub-intervalo, buscando um maior dinamismo.

Limites mínimo e máximo do tamanho do roubo: Mór (2015) sugere que a quantia de iterações a ser roubada em um processo de extração paralela para algoritmos adaptativos possua um tamanho mínimo, como a raiz quadrada do tamanho original do sub-intervalo. No entanto tal abordagem apresenta problemas em AMPs: limitar o roubo de trabalho pode levar com que um núcleo mais rápido opte por não roubar trabalho de um núcleo mais lento, aumentando o tempo total de execução.

Para o limite máximo, escolheu-se metade do trabalho restante, com base no trabalho de alguns autores (MÓR, 2015; DURAND et al., 2013; QUINTIN; WAGNER, 2010). Alternativamente, abordagens como do CilkPlus e IntelTBB, que dividem recursivamente o intervalo em partes iguais e armazenam-os em um deque, permitem uma forma adaptativa de roubo de trabalho mas limitada, uma vez que pra cada intervalo de tamanho N há somente $\lceil \log_2 N \rceil$ tamanhos possíveis de porções a serem roubados (MÓR, 2015). Com a abordagem adotada, traz-se mais dinamismo ao tamanho da porção a ser roubada durante um roubo de trabalho.

Qualquer abordagem que determine que escalonador distinga entre os tipos de núcleos presentes no sistema traz a tona uma necessidade de um método extra que determine o tipo de cada processador, além de uma forma de agrupá-los conforme seu desempenho. Tais abordagens diminuem a portabilidade da solução – um mesmo código teria de ser compilado com parâmetros diferentes para AMPs diferentes, ou informações sobre os PUs deveriam ser fornecidas manualmente no início de cada execução – e encaixariam a mesma como um algoritmo *ajustado*, e não adaptativo, como proposta principal do trabalho.

A Ilustração 3.6 ilustra como a divisão inicial da carga de trabalho de um laço e roubos de trabalho são projetados para acontecer quando utilizando o escalonador proposto por este trabalho.

3.4 IMPLEMENTAÇÃO

O escalonador foi implementado na linguagem C++, com *back-end* em POSIX *threads*. A escolha da linguagem se deve ao fato de possuir determinado nível de compatibilidade com a linguagem C, na qual muitos dos benchmarks são escritos, e por aliar alto desempenho com recursos de linguagens modernas, como suporte à programação orientada à objetos e diferentes níveis de polimorfismo.

Durante a inicialização de qualquer aplicação que use o escalonador, um Coordenador representado por um objeto estático da classe `ThreadHandler` é criado e se encarrega

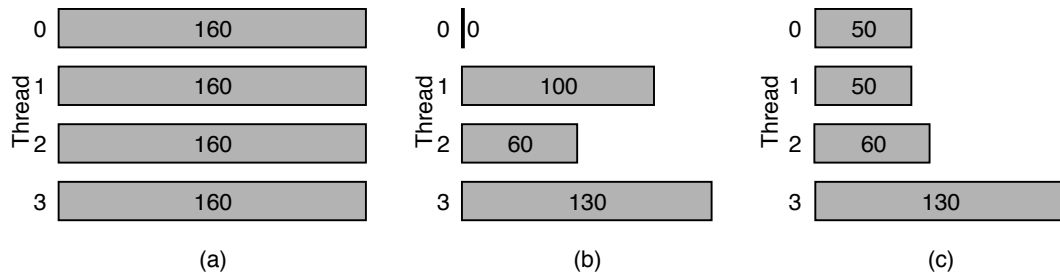


Ilustração 3.6 – Exemplos de divisão inicial e roubo de trabalho. São ilustrados três estados subsequentes: (a) Um laço paralelo de 640 iterações tem seu intervalo de iterações inicialmente dividido de forma igualitária entre todas as *threads* em um sistema com 4 *threads*; (b) Em determinado momento da execução, a *thread* 0 consome todo seu sub-intervalo de iterações, tornando o Trabalhador a ela associado um ladrão; (c) O Trabalhador ladrão escolhe aleatoriamente uma vítima (o Trabalhador da *thread* 1) e efetua um roubo, realocando metade do trabalho restante (50 de 100 iterações) para si.

Fonte: O autor.

de criar e lançar as *threads* a serem utilizadas pelo escalonador, deixando-as a espera de Trabalhadores (objetos instâncias da classe `Worker`). Cada Trabalhador acaba por tornar-se único, devido aos diferentes tipos de corpo de laço e tipos de dados usados como índice. Uma classe base abstrata `AbstractWorker` foi definida para possibilitar a manipulação dos objetos Trabalhadores de forma generalizada. Há dois tipos de Trabalhadores: `ForWorker`, para laços paralelos, e `ReductionWorker`, para laços paralelos com redução. O objeto `ThreadHandler` é destruído automaticamente pela *runtime* ao final da execução, assim como as *threads* que criou, ao passo que os objetos `Worker` são destruídos ao final de cada laço. A Ilustração 3.7 apresenta um diagrama UML das classes implementadas.

O algoritmo de redução utilizado pelo escalonador funciona em um esquema de árvore binária, conforme exibido na Ilustração 3.8. O número de esperas por resultado nesta implementação ($\log_2 N$) é menor que o número de esperas em uma implementação linear ($N - 1$) e as esperas são distribuídas entre as *threads*.

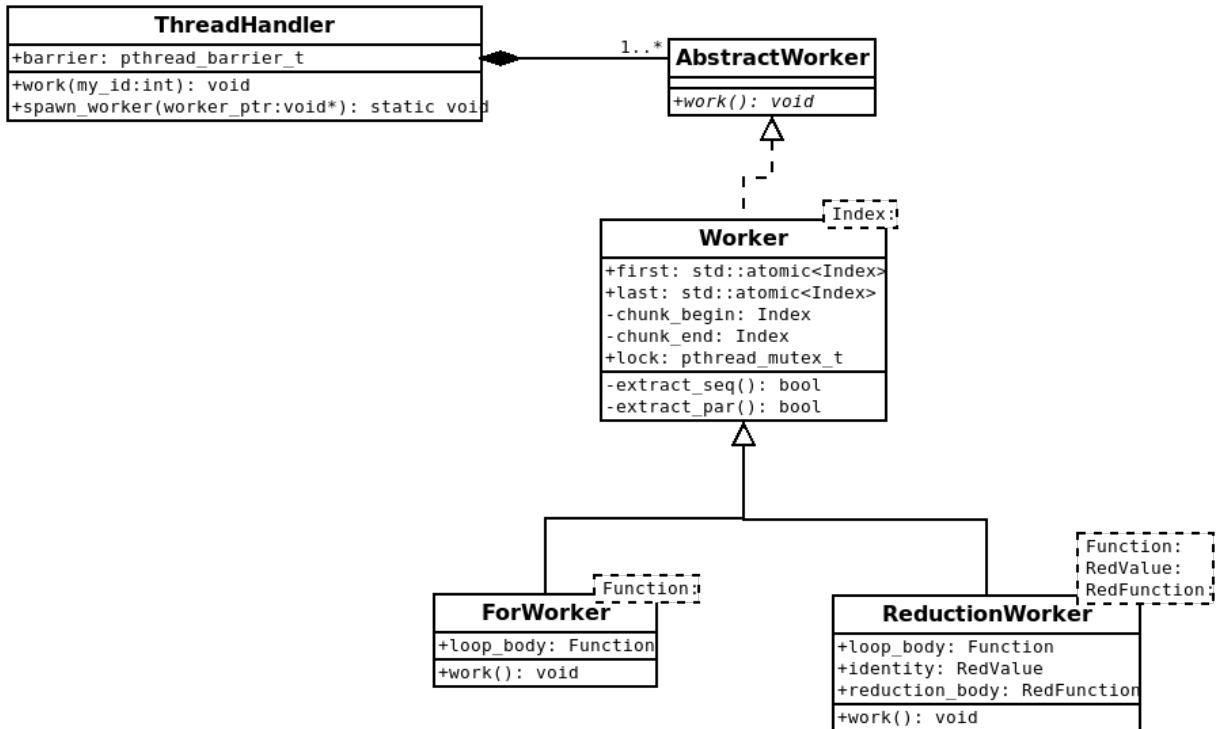


Ilustração 3.7 – Diagrama UML das classes utilizadas pelo escalonador. Há diferenciação no tipo Function de cada implementação final de Worker, uma vez que funções relativas ao corpo de laços com redução necessitam de um parâmetro a mais, e retornam um valor, diferentemente de laços paralelos comuns.

Fonte: O autor.

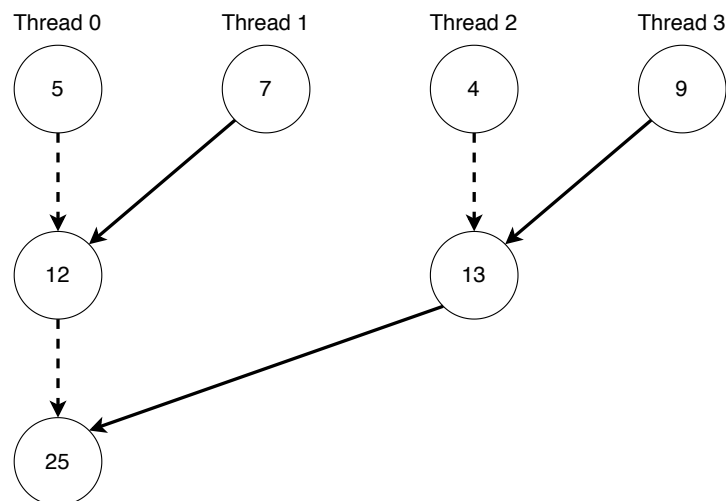


Ilustração 3.8 – Esquema de redução em árvore empregado pelo escalonador. A ilustração exhibe o emprego da redução em um laço paralelo com $N = 4$ threads e soma (+) como operação de redução. Linhas tracejadas indicam que a thread está esperando o resultado de outra thread, enquanto linhas contíguas indicam comunicação. Cada thread espera no mínimo 0 e no máximo $\lceil \log_2 N \rceil$ vezes pelo resultado de outra thread.

Fonte: O autor.

Como mencionado anteriormente, o escalonador utiliza tipos atômicos do C++

(`std::atomic`) para as variáveis que representam os limites dos sub-intervalos de iterações de cada Trabalhador. A Ilustração 3.9 justifica a sua escolha, exibindo uma comparação de desempenho de tipos atômicos contra outros tipos comuns de trava de exclusão mútua, apresentando casos onde o tipo atômico utilizado consegue realizar operações de leitura e escrita única em variáveis até 22 vezes mais rápido que uma trava `omp_lock_t` do OpenMP, por exemplo. A diretiva `atomic` do OpenMP possibilitou tempos ainda menores, entretanto a mesma não oferece suporte para uso em operações de comparação, somente de atribuição (OpenMP Architecture Review Board, 2018).

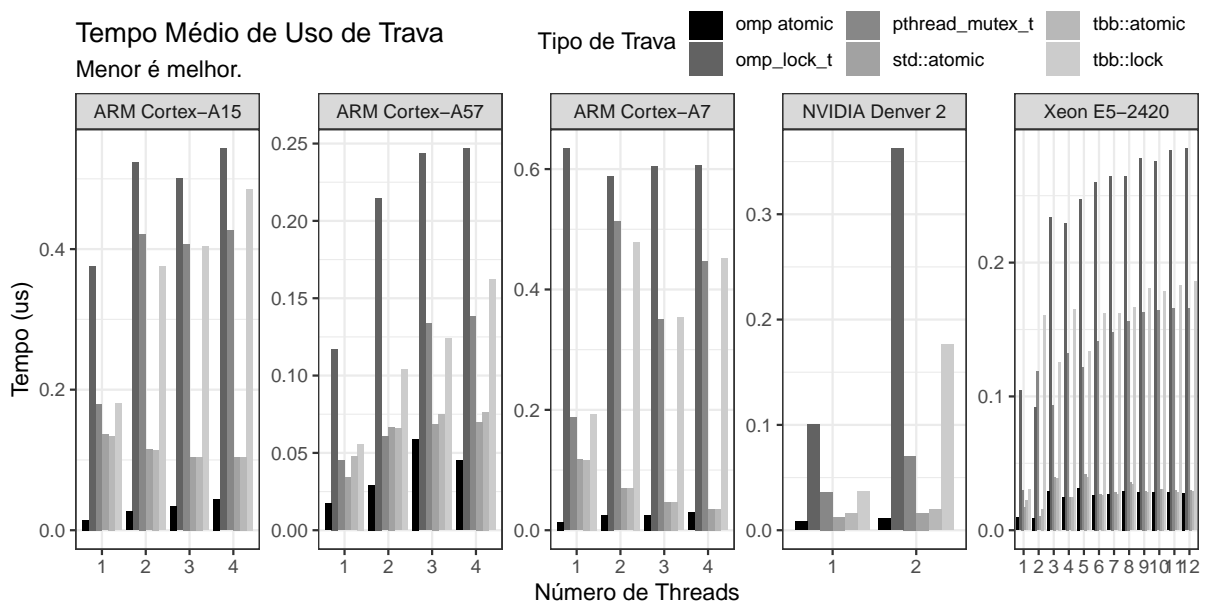


Ilustração 3.9 – Comparação de desempenho de tipos atômicos do C++ com outros tipos de trava. Cada gráfico corresponde a um tipo de CPU diferente, com o eixo horizontal representando o número de *threads* e o eixo vertical o tempo médio consumido por trava. Nesta comparação, cada *thread* incrementa concorrentemente dez milhões de vezes um contador compartilhado.

Fonte: O autor.

3.4.1 API para Laços Paralelos

Para que fosse possível portar aplicações para usar a solução proposta, foi desenvolvida uma Interface de Programação de Aplicativos (API). A API é baseada no formato da biblioteca Intel TBB, onde a criação do laço paralelo se dá através da chamada de um método. Há suporte para múltiplos tipos diferentes de funções e tipos de dado que representem os valores do intervalo de iterações graças à capacidade polimórfica da sintaxe da linguagem oferecida pelos seus *templates*. Os tipos e funções são deduzidos em tempo de compilação. A Ilustração 3.10 apresenta os dois tipos de métodos disponíveis na API: um

para laços paralelos comuns e outro para laços paralelos com redução. Um exemplo de uso pode ser encontrado na Ilustração 3.11. Como a API não prevê pré-processamento de código, mesclar laços aninhados com o laço mais externo (equivalente a cláusula `collapse` do OpenMP) não é incluído como um recurso da API: tais mesclas devem ser realizadas manualmente em código pelo desenvolvedor.

```

1 // laços paralelos
2 adapt::parallel_for(inicio , fim , corpo_laco);
3
4 // reducao paralela
5 valor = adapt::parallel_reduce(inicio , fim , identidade , corpo_laco ,
6     corpo_reducao);

```

Ilustração 3.10 – Assinaturas dos métodos de laço e redução paralelas. `inicio` e `fim` denotam o início e o final do intervalo de iterações. `corpo_laco` representa o bloco de código interior de um laço a ser executado em paralelo, podendo ser uma função típica ou uma função *lambda*, assim como `corpo_reducao`, que por sua vez define o método a ser utilizado para redução de valores. Por fim, `identidade` representa o valor inicial de um dado a ser reduzido, e `valor` o valor reduzido ao final do laço.

Fonte: O autor.

```

1 // laço paralelo
2 int iarray[10000];
3 adapt::parallel_for(0, 10000, [&iarray](const int begin, const int end) {
4     for (int i = begin; i < end; i++)
5         iarray[i] = i * 2;
6 });
7
8 // reducao paralela
9 int reducao = adapt::parallel_reduce(0, 10000, 0,
10 [&iarray](const int begin, const int end, int initial) {
11     int partial = initial;
12     for (int i = begin; i < end; i++)
13         partial += iarray[i];
14     return partial;
15 }, std::plus<int>());
16

```

Ilustração 3.11 – Caso exemplo de paralelização de laço com a API proposta. Funções *lambda* são usadas como corpos dos laços. O vetor de inteiros `iarray` é inicialmente preenchido e seus valores são reduzidos com uma soma, na sequência.

Fonte: O autor.

3.5 SUMÁRIO

Este Capítulo apresentou a proposta de um escalonador para laços paralelos. Sua implementação toma trabalhos relacionados como base (TRAORÉ et al., 2008; DURAND et al., 2013; MÓR, 2015) e especializa características dos algoritmos para ambientes com capacidade de processamento assimétrica (AMPs), valendo-se de suas características híbrida e adaptativa por, respectivamente, possuir diferentes métodos para extração de trabalho e por não discriminar tipos diferentes de processadores. As seções aqui apresentadas descrevem como o escalonador é composto estruturalmente, como seus passos são executados em seu laço principal, e como seus algoritmos de extração sequencial e paralela funcionam, bem como detalhes da API projetada para sua utilização e sua implementação em C++ através do paradigma de orientação à objetos.

O escalonador diferencia-se de outras soluções existentes não somente por sua adaptatividade, mas também por possuir características específicas em cada algoritmo de extração – como a eliminação semi-completa do uso de travas para extrair trabalho sequencial – herdados do protocolo THE (FRIGO; LEISERSON; RANDALL, 1998), que foram projetadas de forma a possibilitar vantagens pontuais em comparação a ferramentas similares do estado da arte disponíveis para uso. O capítulo a seguir aborda uma comparação de desempenho do escalonador proposto.

3.6 ANÁLISE DE DESEMPENHO

Para avaliar e comparar o escalonador proposto com demais soluções, foi definido um conjunto de ambientes e aplicações a serem usados. Este Capítulo aborda a metodologia (seção 3.7) utilizada na análise e seus respectivos resultados (seção 3.8).

3.7 METODOLOGIA

Foram coletados tempos de execução de cada conjunto de ambiente, aplicação e escalonador. Em adição ao escalonador proposto, foram utilizados os seguintes escalonadores:

OpenMP : `static`, `guided` e `dynamic` (grão 1 e grão ótimo);

Intel TBB : `auto_partitioner` e `simple_partitioner`

O escalonador adaptativo foi avaliado com os três tipos de grão propostos para extrações sequenciais (Capítulo 3 seção 3.3): fixo, fracionado e logarítmico. Os tamanhos de grão ótimo a serem listados neste Capítulo foram utilizados para determinar o tamanho da porção de extração sequencial quando no escalonador dinâmico do OpenMP e no escalonador proposto com grão de tamanho fixo. Para o grão de tamanho fracionado, foram escolhidas 256 partes (leia-se um sub-intervalo de iterações é sub-dividido em 256 partes ou menos). A quantia de partes foi escolhida para se alcançar um tamanho similar ao tamanho em uma divisão binária recursiva de até 9 níveis de profundidade, similar à quantia de divisões que escalonador automático do Intel TBB realiza por padrão nas duas plataformas utilizadas, caso não ocorram roubos de trabalho adicionais.

Cláusulas `nowait` do OpenMP foram removidas de aplicações que as possuíam, uma vez que não há solução equivalente no escalonador proposto e no Intel TBB. Os tempos de execução correspondem à média de 30 execuções da aplicação em cada conjunto de configuração, associada com um intervalo de confiança de 95%, representados nos gráficos na seção 3.8. Cada *thread* de software utilizada pelas bibliotecas foi associada à uma *thread*/núcleo físico do hardware, de modo a expor o impacto da assimetria quando delegadas as tarefas de escalonamento inteiramente às bibliotecas.

O compilador C/C++ utilizado foi a versão 8 do Clang/LLVM, bem como a *runtime* OpenMP utilizada foi a `libomp`, padrão do compilador. Os sistemas operacionais utilizados para as execuções foram versões próprias do Ubuntu para cada plataforma utilizada.

3.7.1 Métricas

Acumeração paralela é uma forma de avaliar o quão rápida uma aplicação paralela é quando comparada à sua melhor versão equivalente executada de forma sequencial. Para calcular a aceleração paralela A_P , divide-se o tempo de execução da melhor versão sequencial T_S pelo tempo de execução da versão paralela T_P .

$$A_P = \frac{T_S}{T_P} \quad (3.1)$$

Uma vez que AMPs possuem mais de um tipo de núcleo, o menor tempo de execução da versão sequencial entre todos os núcleos é escolhido.

Outra métrica comumente utilizada para estimar desempenho de implementações paralelas é a eficiência paralela. Considerando um SMP com N núcleos e uma aplicação com tempos de execução T_S da versão sequencial e T_P da versão paralela, a eficiência paralela E_P é calculada de acordo com a Equação 3.2 (FOX et al., 1988). Idealmente a eficiência deve ser 1 – isto é, a carga de trabalho é igualmente dividida entre os núcleos e cada um leva exatamente tempo T_S/N para finalizar seu trabalho.

$$E_P = \frac{T_S}{N * T_P} \quad (3.2)$$

Entretanto essa métrica não é aplicável para AMPs, uma vez que núcleos de tipos diferentes podem não ter a mesma capacidade de processamento e, conseqüentemente, mesmos tempos de execução. Kalinov (2006) definiu uma equação genérica que pode determinar a eficiência paralela desse tipo de ambiente, e que abrange tanto SMPs como AMPs. Considerando um conjunto $R = r_i, i \in [1, N]$ de desempenhos de processadores representados por números reais positivos r_i , estima-se o desempenho médio r_{avg} :

$$r_{avg} = \frac{\sum_{i=1}^N r_i}{N} \quad (3.3)$$

e considerando r_{seq} como o desempenho do processador usado para executar a versão sequencial da aplicação, com tempo de execução $T_S^{r_{seq}}$, e T_P^R como o tempo de execução da versão paralela em todos os processadores do conjunto R . Então para avaliar a eficiência paralela basta multiplicar a forma equivalente da Equação 3.2 pela razão entre r_{seq} e r_{avg} :

$$E_P = \frac{r_{seq}}{r_{avg}} * \frac{T_S^{r_{seq}}}{N * T_P^R} \quad (3.4)$$

Como neste trabalho se é usado somente dois tipos de núcleo, uma versão simplificada da equação pode ser escrita. Primeiro, determina-se a razão entre desempenhos de núcleos big e LITTLE. Como 'desempenho' é um conceito abstrato, optou-se por utilizar tempos de execução das versões sequenciais dos programas em ambos os tipos de

núcleo, T_S^{big} e T_S^{LITTLE} . A razão de desempenho σ é então calculada como:

$$\sigma = \frac{1/T_S^{big}}{1/T_S^{LITTLE}} = \frac{T_S^{LITTLE}}{T_S^{big}} \quad (3.5)$$

Por fim descreve-se a eficiência paralela para um AMP big.LITTLE usando o tempo sequencial em um núcleo LITTLE como base. As variáveis N^{big} e N^{LITTLE} representam o total de núcleos presente nos clusters big e LITTLE, respectivamente.

$$E_P = \frac{T_S^{LITTLE}}{(\sigma * N^{big} + N^{LITTLE}) * T_P^R} \quad (3.6)$$

Como o desempenho de um processador varia de acordo com o tipo de aplicação sendo executada, faz-se importante recalcular o valor de σ para aplicações diferentes.

3.7.2 Ambientes

A avaliação foi conduzida em dois dispositivos computacionais embarcados diferentes conforme descritos na Tabela 3.1.

Tabela 3.1 – Dispositivos Embarcados Utilizados na Avaliação

Recurso	Hardkernel ODROID-XU3 Lite	NVIDIA Jetson TX2
Arquitetura	ARM 32-bit	ARM 64-bit ^a
Modelo	ARM Cortex-A15	NVIDIA Denver 2
Núcleos	4	2
Frequência	1.8 GHz	2 GHz
CPU big	Cache L1D	64 KB
	Cache L1I	128 KB
	Cache L2	2 MB ^b
Execução	Fora-de-Ordem	Em-Ordem
Modelo	ARM Cortex-A7	ARM Cortex-A57
Núcleos	4	4
Frequência	1.3 GHz	2 GHz
CPU LITTLE	Cache L1D	32 KB
	Cache L1I	48 KB
	Cache L2	2 MB ^b
Execução	Em-Ordem	Fora-de-Ordem
Capacidade	2 GB	8 GB
Memória RAM	Tipo	LPDDR3
	Frequência	1866 MHz
	Largura Máxima de Banda	59.7 GB/s

^aA CPU NVIDIA Denver 2 executa internamente seu próprio ISA e converte instruções ARMv8 para tal durante a execução (WASSON, 2014).

^bMemória compartilhada.

Fonte: (CHRONAKI et al., 2017; ODROID Wiki, 2017; NVIDIA Developer, 2019; ARM Limited, 2013a)

A Tabela 3.2 apresenta o número de núcleos e frequências usados por cada configuração de cluster em cada plataforma. Para fins de objetividade, as plataformas serão, a partir deste ponto, referenciadas como TX2 e XU3, para NVIDIA Jetson TX2 e Hardkernel Odroid XU3-Lite, respectivamente.

Tabela 3.2 – Núcleos de CPU por Configuração de Cluster

Tipo de Cluster	ODROID-XU3 Lite	Jetson TX2
big	4 (1.8 GHz)	2 (2 GHz)
LITTLE	4 (1.3 GHz)	4 (2 GHz)
big.LITTLE	8 (1.8 e 1.3 GHz)	6 (2 GHz)

Fonte: O autor.

3.7.3 Benchmarks

Foram escolhidas 4 aplicações científicas e um conjunto de benchmarks para avaliar o desempenho dos escalonadores de laço. Todos os benchmarks utilizavam originalmente construções OpenMP e a maioria implementados na linguagem C. Com a finalidade de possibilitar a compatibilidade com nosso escalonador e o Intel TBB, os códigos foram portados para a linguagem C++.

O NAS *Parallel Benchmark* (BAILEY et al., 1994) é um conjunto de benchmarks que foi desenvolvido para avaliar o desempenho de supercomputadores altamente paralelos. É distribuído com conjuntos de benchmarks de kernels paralelos e aplicações de simulação. Como a implementação original é escrita em Fortran, o trabalho baseou-se em uma versão¹ portada para a linguagem C. A Tabela 3.3 lista as aplicações e kernels usados na avaliação, juntamente com informações úteis sobre suas implementações paralelas.

Tabela 3.3 – Aplicações e Kernels do NAS Parallel Benchmark Usadas na Avaliação

Nome	Sigla	# Passos	# Laços Par.	Grão Ótimo ^a	
				TX2	XU3 Lite
<i>Block Tridiagonal Solver</i>	BT	200	54	1	1
<i>Conjugate Gradient</i>	CG	75	20	256	1000
<i>Embarassingly Parallel</i>	EP	1	2	1	1
<i>3D Fast Fourier Transform</i>	FT	20	6	1	1
<i>LU Solver</i>	LU	250	29	8	8
<i>Multigrid</i>	MG	20	12	1	1
<i>Pentadiagonal Solver</i>	SP	400	70	12	8

^aTamanho de grão que permite ao escalonador dinâmico do OpenMP extrair mais desempenho.

Fonte: O autor.

Os tamanhos de entrada são determinados em tempo de compilação. São divididos em classes de problemas, denotadas por uma letra (BAILEY et al., 1994) (S, W, A, B, e C – tamanho crescente nesta ordem). O tamanho B foi escolhido como padrão para este trabalho devido à limitações de memória do ambiente ODROID-XU3 Lite, que não suportava alguns problemas da classe C.

Além do conjunto de *benchmarks*, foram selecionadas 4 aplicações científicas para compor o conjunto de aplicações para avaliação. As aplicações selecionadas foram escolhidas com o intuito de representarem cargas de trabalho mais próximas de cargas de trabalho de aplicações reais. Em suma elas representam aplicações de dinâmica molecular e de fluídos computacional, métodos de simulação numérica utilizados pela academia e pela indústria com o intuito de simular comportamentos de partículas ou materiais do mundo real em determinados cenários.

¹<https://github.com/benchmark-subsetting/NPB3.0-omp-C>

CoMD – *Co-design Molecular Dynamics*² é uma aplicação *proxy* para dinâmica molecular – uma aplicação que encapsula a carga de trabalho de um aplicação científica real – empregada pelo *Exascale Co-Design Center for Materials in Extreme Environments*, e materializa um esforço para desenvolver uma estrutura na qual as interações entre software e hardware possam ser exploradas em favor da pesquisa com dinâmica molecular (MOHD-YUSOF; SWAMINARAYAN; GERMANN, 2013). Sua estrutura de código pode ser paralelizada com a ajuda de algumas bibliotecas multi-threading, como MPI, pthreads e OpenMP. Possui dois kernels de força: *Leonard-Jones* (LJ) e o Modelo de Átomo Incorporado (EAM), ambos dominando no mínimo 80% do tempo de execução na grande maioria dos casos (CICOTTI; MNISZEWSKI; CARRINGTON, 2014). Para este trabalho o kernel LJ foi selecionado, seguindo o padrão da aplicação, onde a força de cada molécula é calculada em função das moléculas vizinhas e, como cada molécula pode ter um número diferente de vizinhos, apresenta um padrão irregular nos tempos pra cada iteração do kernel.

LBM – *Lattice Boltzmann Method* é diferente de outros métodos numéricos de mecânica contínua e derivados da dinâmica molecular (SCHEPKE; MAILLARD, 2007). O LBM foi desenvolvido como uma ferramenta numérica viável para a dinâmica de fluidos computacional. Possui um algoritmo simples, de fácil manuseio das condições de contorno e possui fácil adequação ao paralelismo. A estrutura de seu algoritmo é composta por um laço principal com quatro etapas por iteração, cada uma com um padrão diferente de acesso à memória. Várias versões já foram implementadas (pthreads, OpenMP, Intel Cilk, CUDA, OpenCL, OpenACC, HOMP e OpenMP com *offloading* para GPU) (TRINDADE; LIMA, 2017), mas nenhuma visando processadores assimétricos.

LULESH – *Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics* é um problema completo para testar várias técnicas de ajuste e diferentes modelos de programação em hidrodinâmica (KARLIN, 2012). LULESH possui duas versões e a mais recente é explorada por este trabalho³. Também possui versões para diferentes APIs programação paralela e arquiteturas, incluindo OpenMP (com MPI), Cuda, OpenACC, OpenMP com *offloading*, OpenCL, entre outros.

Ondes3D – Ondes3D⁴ é uma aplicação científica que visa resolver equações elastodinâmicas e simular a propagação de ondas sísmicas usando o método das diferenças finitas (DUPROS; DO; AOCHI, 2013). Possui versões que usam OpenMP (com MPI), CUDA (com MPI) e uma versão heterogênea com StarPU, implementada com paralelismo de tarefas (MARTÍNEZ et al., 2015). Observa-se que a aplicação possui um

²<<https://github.com/ECP-copa/CoMD>>

³<<https://github.com/LLNL/LULESH>>

⁴<<https://bitbucket.org/fdupros/ondes3d>>

padrão de acesso à memória irregular, e seus laços paralelos tendem a mudar de comportamento em relação à tempo de execução por iteração ao longo da simulação.

As quatro aplicações científicas possuem em comum o uso de uma malha/grid tridimensional que representa ou contém os dados a serem processados. A Tabela 3.4 apresenta os tamanhos de malha utilizados e outras informações a respeito de sua estrutura em código.

Tabela 3.4 – Aplicações Científicas Usadas na Avaliação

Nome	# Passos	# Laços Paralelos	Tamanho da Malha	Grão Ótimo ^a	
				TX2	XU3 Lite
CoMD	100	11 ^b	80 x 80 x 80	224	48
LBM	400	4	1000 x 1000 x 9	88	88
LULESH	100	39	120 x 120 x 120	4516	512
Ondes3D	100	3	197 x 200 x 101	1	1

^aTamanho de grão que permite ao escalonador dinâmico do OpenMP extrair mais desempenho.

^bCom kernel de força Lennard-Jones.

Fonte: O autor.

3.8 RESULTADOS EXPERIMENTAIS

Esta seção apresenta os resultados da análise de desempenho, destacando tempos de execução sequenciais e paralelos para cada aplicação/*benchmark* em cada ambiente de processamento assimétrico utilizado, conforme descrito na seção anterior.

3.8.1 Desempenho Sequencial

A Tabela 3.5 exibe os tempos de execução dos *benchmarks* em cada tipo de núcleo de cada plataforma, além de exibir a razão de diferença de desempenho de um núcleo rápido para um núcleo lento (símbolo σ), e um valor de aceleração paralela máxima estimada (APME), considerando 1 núcleo lento como $1/\sigma$ núcleos. Essa estimativa se baseia nos tempos de execução total fornecidos pelos *benchmarks* e não discrimina eventuais trechos de código sequenciais em meio aos códigos paralelos.

A plataforma XU3-Lite exibe diferenças de desempenho entre seus núcleos maiores em comparação às diferenças obtidas com a plataforma TX2. Apesar de possui 8 núcleos ao todo, as acelerações máximas estimadas não ultrapassam 5.32 vezes o tempo de exe-

Tabela 3.5 – Tempos^a de Execução Sequencial dos Benchmarks, em Segundos

Benchmark	TX2				XU3-Lite			
	big	LITTLE	σ	APME^b	big	LITTLE	σ	APME^b
BT	692.63	896.41	1.29	5.09	1823.75	5525.39	3.03	5.32
CG	75.74	255.66	3.37	3.18	321.95	2483.71	7.71	4.52
EP	191.18	257.65	1.35	4.97	281.08	539.68	1.92	6.08
FT	94.87	118.70	1.25	5.20	173.37	583.07	3.35	5.19
LU	463.56	763.79	1.65	4.43	1178.53	3329.29	4.94	4.81
MG	13.11	21.00	1.60	4.49	35.37	174.70	2.82	5.42
SP	386.29	646.59	1.67	4.39	996.71	4242.35	4.32	4.92
CoMD	1349.35	1390.84	1.03	5.88	1828.30	4157.93	2.27	5.76
LBM	68.82	84.40	1.23	5.26	148.90	427.62	2.87	5.39
LULESH	560.91	621.28	1.11	5.61	1227.73	3902.80	3.18	5.26
Ondes3D	189.64	397.00	2.09	3.91	430.50	1338.32	3.11	5.29

^a Médias de 3 execuções, valores aproximados.

^b Aceleração Paralela Máxima Estimada.

Fonte: O autor.

cução em um núcleo rápido, trazendo uma impressão prévia que o uso dos núcleos lentos podem não compensar tanto em questão de desempenho.

3.8.2 NAS Parallel Benchmark

Os Gráficos 3.1, 3.2 e 3.3 a seguir apresentam, respectivamente, tempo médio, eficiência paralela e aceleração paralela para cada aplicação/kernel do NAS *Parallel Benchmark* em cada ambiente utilizado. Todos os gráficos seguem o mesmo padrão de formatação: cada linha de gráficos representam um ambiente, enquanto cada coluna representa uma aplicação. As barras são categorizadas em cores e grupos de barras, com a cor base diferente para cada biblioteca/*framework* de programação paralela. Gráficos onde a barra referente ao escalonador OMP Dynamic (melhor) não está presente indicam que, para este cenário em particular, o grão ótimo para uso com o escalonador dinâmico do OpenMP possui tamanho 1, como já ilustrado na barra referente ao escalonador OMP Dynamic (1). Cada barra referente ao escalonador adaptativo representam, respectivamente, os desempenhos com grão de tamanho fixo (tamanho de grão idêntico ao usado pelo escalonador dinâmico do OpenMP), fracionado (256 partes) e logarítmico.

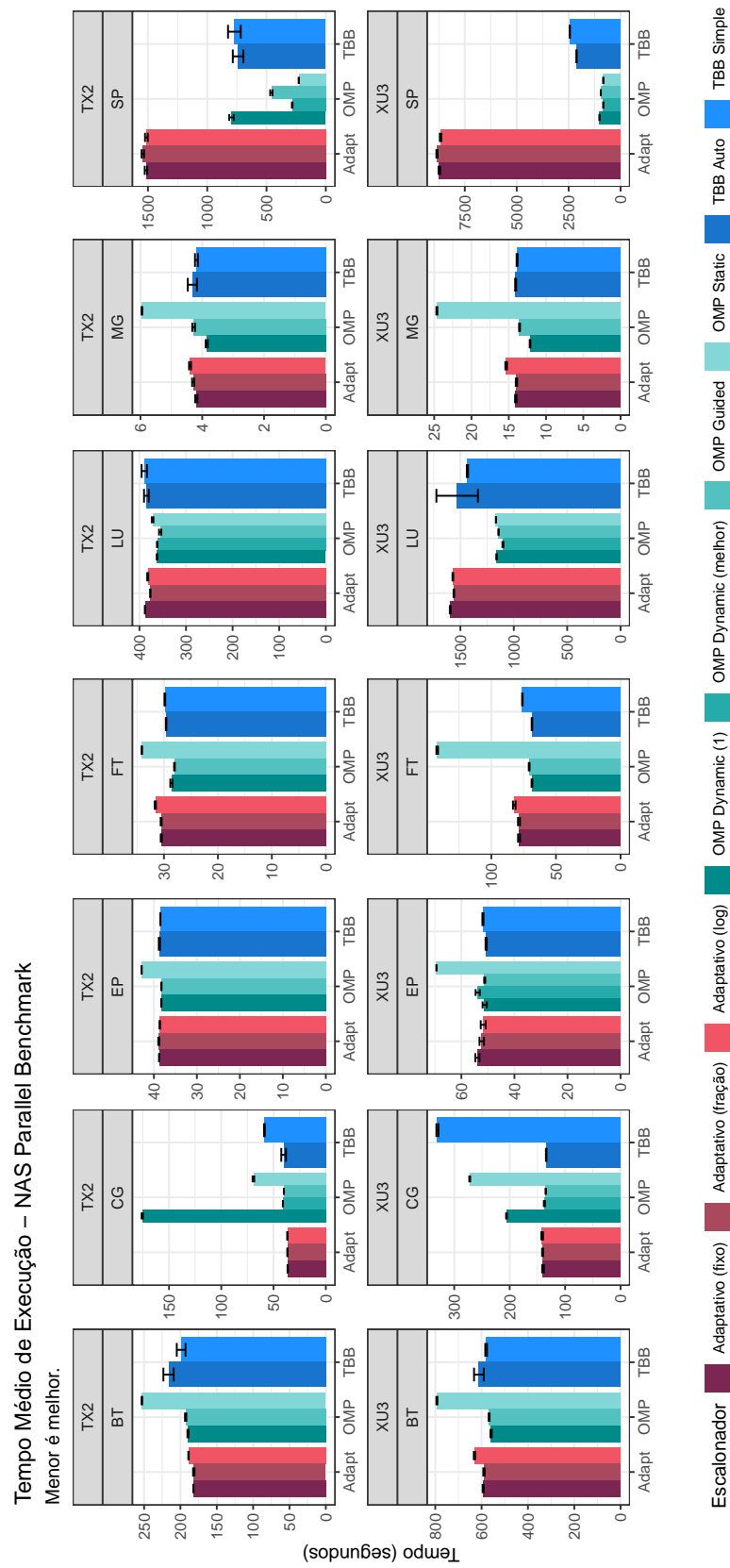


Gráfico 3.1 – Tempo médio de execução com o NAS *Parallel Benchmark*. As barras verticais representam o intervalo de confiança de 95% em relação às amostras de tempo coletadas.

Fonte: O autor.

Eficiência Paralela – NAS Parallel Benchmark
 Maior é melhor.

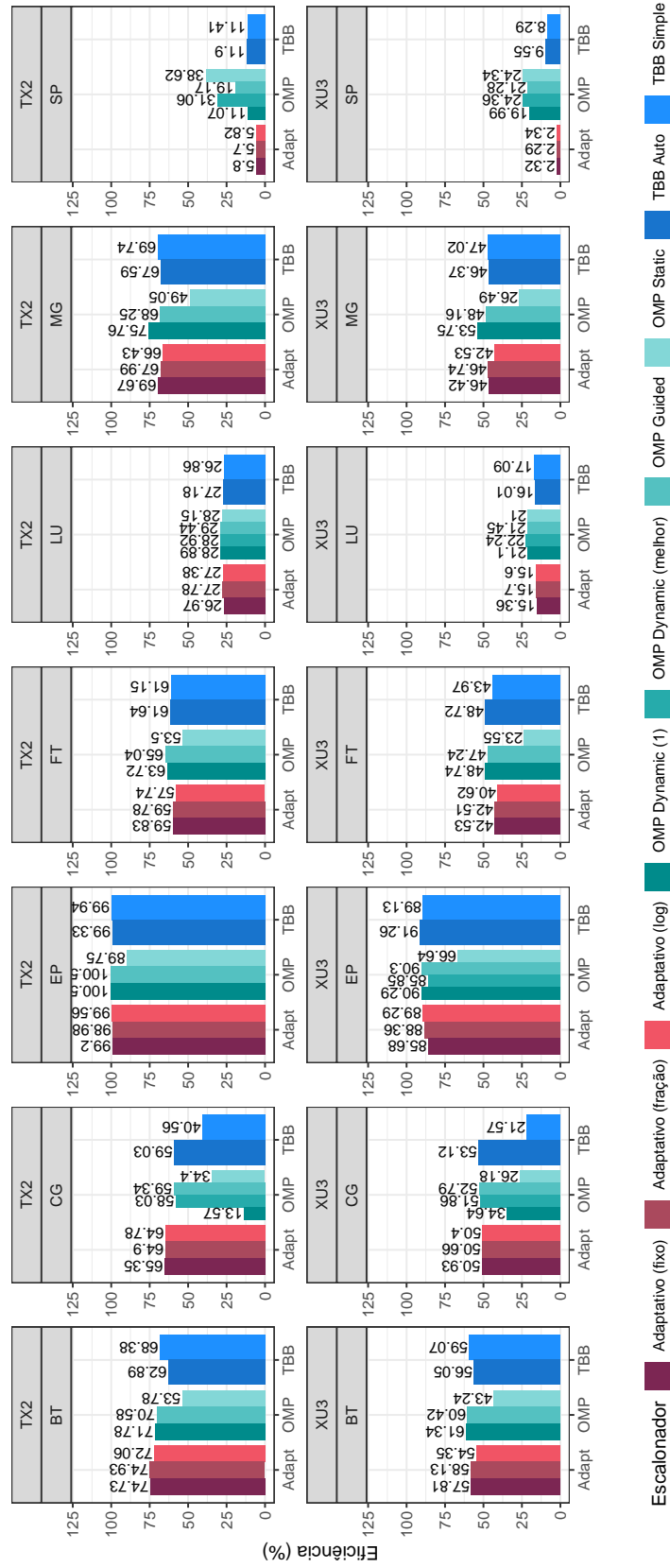


Gráfico 3.2 – Eficiência paralela com o NAS *Parallel Benchmark*.

Fonte: O autor.

Aceleração Paralela – NAS Parallel Benchmark
 Maior é melhor.

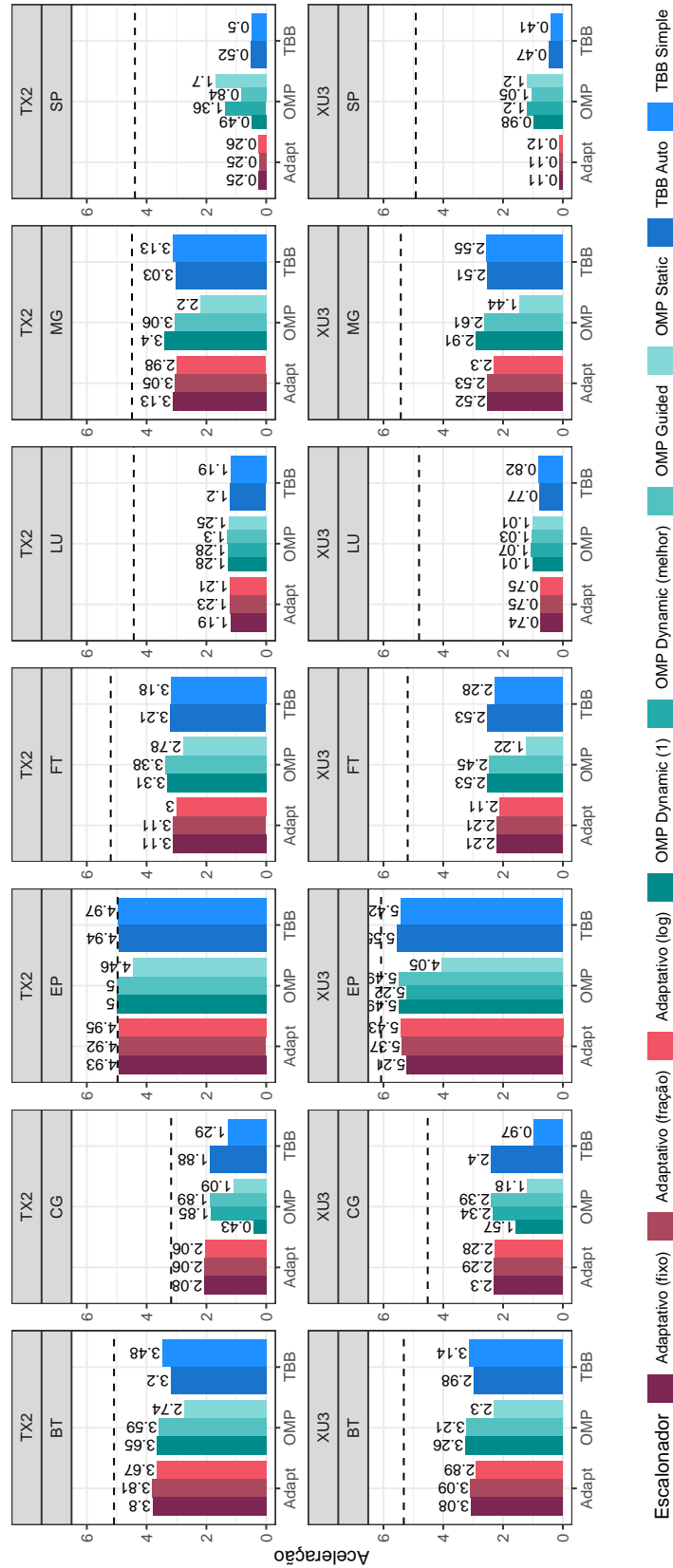


Gráfico 3.3 – Aceleração paralela com o NAS *Parallel Benchmark*. Linhas horizontais indicam a aceleração paralela máxima estimada a partir dos tempos de execução sequenciais, conforme descrito na Tabela 3.5.

Fonte: O autor.

O escalonador adaptativo conseguiu menores tempos de execução, ainda que com diferença pouco significativa, em 2 dos 14 cenários (BT e CG no ambiente TX2), de acordo com o intervalo de confiança de 95% aplicado (Ilustração 3.1). A maioria dos outros cenários apresentam menores tempos para soluções com OpenMP (11 de 14), deixando poucos cenários onde a biblioteca Intel TBB consegue vencer os concorrentes (3 de 14). Em 2 cenários (CG e FT em XU3) as bibliotecas OpenMP e TBB atingiram juntamente o melhor resultado, de acordo com o intervalo de confiança aplicado. O *benchmark* CG, dentre todas as aplicações avaliadas, é o que mais possui discrepâncias nos desempenhos entre núcleos rápidos e lentos nas duas plataformas. Tal característica serve para ilustrar que uma assimetria relativamente grande não prejudica o escalonamento adaptativo, podendo ser contornada com efetividade.

Apesar do número reduzido de cenários em que superou as demais soluções, o escalonador adaptativo atingiu valores de eficiência paralela razoáveis: em 10 dos 12 cenários onde não obteve o melhor resultado, a diferença em termos de eficiência paralela para a melhor solução não foi maior que 7%, tendo em metade destes cenários uma diferença menor que 3% (Ilustração 3.2).

Para os *benchmarks* LU e SP, o escalonador adaptativo apresentou desempenhos muito próximos ao das versões sequenciais, ficando inclusive mais lento em alguns casos. Tal característica também pode ser observada com a biblioteca Intel TBB e, apesar de ter extraído melhores desempenhos, os escalonadores do OpenMP não conseguiram acelerações paralelas maiores que 1.7 vezes (Ilustração 3.3). A explicação é a estrutura dos próprios *benchmarks*: apesar do *benchmark* SP ser composto por 70 laços paralelos distintos, apenas 6 deles são responsáveis por 97.5% de todas as 24.5 milhões de invocações a laços paralelos. Esses 6 laços combinados são responsáveis por apenas 14.4% do tempo de execução do *benchmark* em sua versão sequencial, entretanto, como suas cargas de trabalho são muito pequenas devido à classe de entrada escolhida – com tempos de execução sequenciais variando entre 0.6 e 5.14 microssegundos em um núcleo *big* do ambiente TX2 – os mesmos acabam por não se beneficiar do paralelismo, tendo seus tempos aumentados de 10 até 81 vezes devido ao sobrecustos com escalonamento e sincronização que a paralelização traz consigo, e saltando dos 14.4% para 80.7% do tempo total de execução.

De forma similar, o *benchmark* LU possui 6 laços de pequena carga computacional – 0.6 a 4.2 milissegundos – que são responsáveis por 98.6% de todas as invocações de laços paralelos e 78.4% do tempo de execução da versão sequencial. Entretanto, como esses laços possuem relativamente mais carga que os destacados do *benchmark* SP, se é possível extrair desempenho através do uso de paralelismo, ainda que limitado, explicando o porquê de a aceleração paralela no ambiente TX2 se manter baixa ainda que mais com um tempo menor que a versão sequencial, fato que não se repetiu no ambiente XU3 onde o sobrecusto de gerenciar 4 núcleos relativamente bem mais lentos acabou refletindo na

aceleração obtida, abaixo de 1. Em ambos os cenários as versões com OpenMP se beneficiam do uso de regiões paralelas em comum entre alguns laços, reduzindo o sobrecusto com a criação das mesmas, recurso que não possui equivalência tanto no escalonador proposto quanto nos escalonadores do TBB.

Além disso, uma análise do perfil de execução das aplicações, coletados com a ferramenta Score-P e analisados com a ferramenta Vampir, mostra que nos cenários descritos acima, boa parte dos tempos de execução são responsabilidade das barreiras colocadas no início e final de cada laço paralelo, de forma a garantir a sincronia das *threads* durante a execução e resultado corretos. O *benchmark* SP, em especial, passa maior parte do tempo com suas *threads* ociosas devido à forma como as barreiras da biblioteca Pthread, utilizada como *backend* no escalonador, foram implementadas. As barreiras `pthread_barrier_t` forçam cada *thread* participante a esperar pelas demais *threads* de forma ociosa, liberando temporariamente o processador a qual estão atreladas.

A fim de avaliar como seria um resultado sem esse comportamento, implementou-se uma classe de barreira alternativa à da biblioteca Pthread, com variáveis atômicas e espera ocupada. Em testes experimentais com o *benchmark* SP e a nova classe de barreira, os tempos de execução foram reduzidos em até 50%, com ganhos de desempenhos menores ou imperceptíveis para as demais aplicações, com melhores efeitos podendo ser observados no ambiente XU3, que dispõe de mais *threads*/processadores a serem sincronizadas através das barreiras. Os ganhos de desempenho não são significativos o suficiente para tornar o escalonador como a melhor opção nos cenários em que inicialmente não foi, mas aproxima ainda mais a eficiência paralela máxima atingida em relação à das melhores soluções. Para trabalhos posteriores, investigar-se-á com mais detalhes o tamanho do impacto desse tipo de primitiva de sincronização, além do efeito do emprego de uma também nova classe de travas de software com espera ocupada, implementada também com variáveis atômicas.

3.8.3 Aplicações Científicas

Os Gráficos 3.4, 3.5 e 3.6 apresentam, respectivamente, tempo médio, eficiência paralela e aceleração paralela para cada aplicação científica em cada ambiente utilizado. Assim como na seção anterior, todos os gráficos seguem o mesmo padrão de formatação: cada linha de gráficos representam um ambiente, enquanto cada coluna representa uma aplicação. As barras são categorizadas em cores e grupos de barras, com a cor base diferente para cada biblioteca/*framework* de programação paralela. Gráficos onde a barra referente ao escalonador OMP Dynamic (melhor) não está presente indicam que, para este cenário em particular, o grão ótimo para uso com o escalonador dinâmico do OpenMP possui tamanho 1, como já ilustrado na barra referente ao escalonador OMP Dynamic (1).

Cada barra referente ao escalonador adaptativo representam, respectivamente, os desempenhos com grão de tamanho fixo (tamanho de grão idêntico ao usado pelo escalonador dinâmico do OpenMP), fracionado (256 partes) e logarítmico.

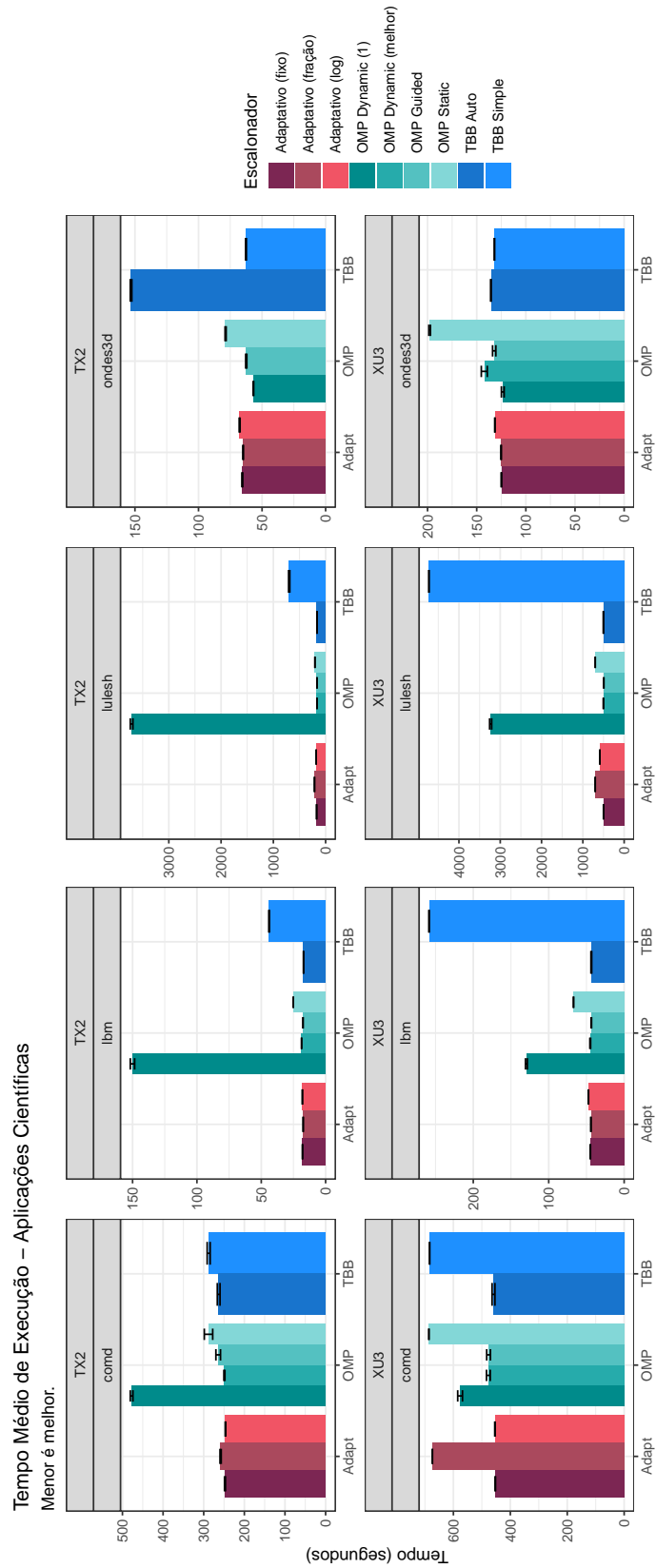


Gráfico 3.4 – Tempo médio de execução com as aplicações científicas selecionadas. As barras verticais representam o intervalo de confiança de 95% em relação às amostras de tempo coletadas.

Fonte: O autor.

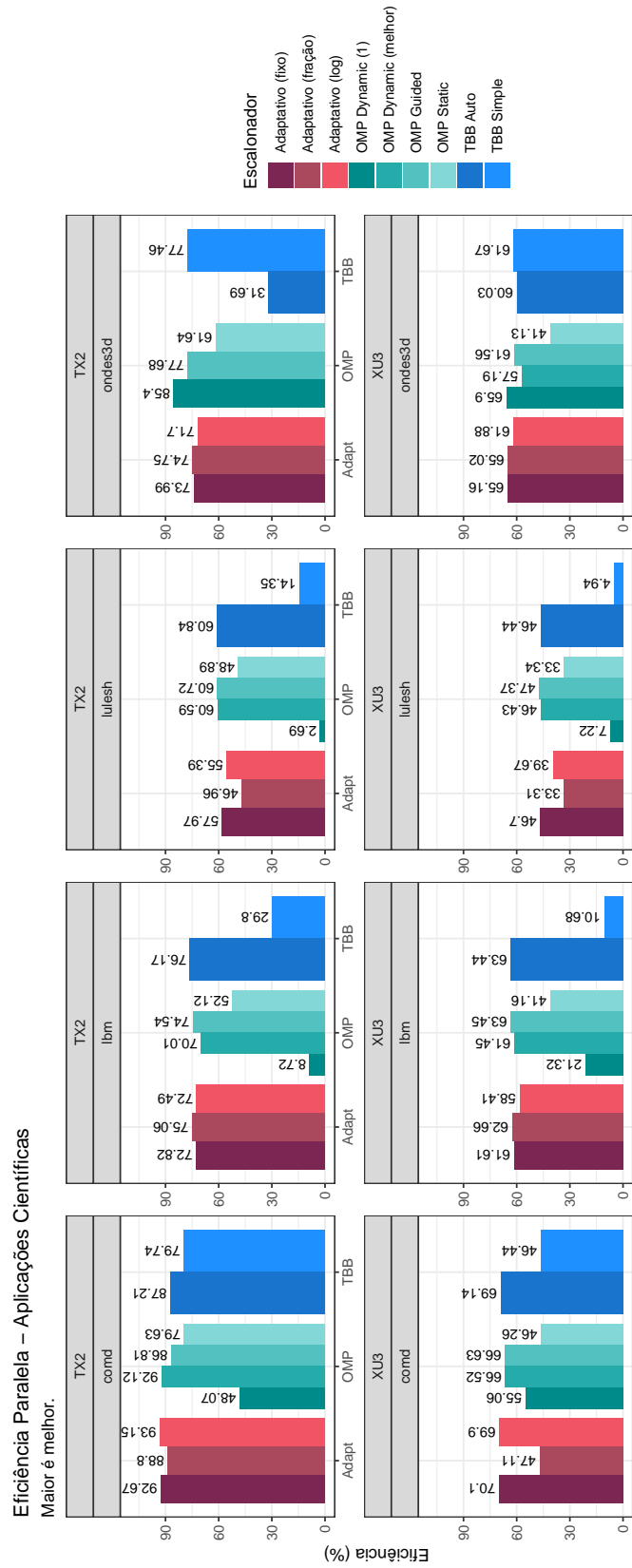


Gráfico 3.5 – Eficiência paralela com as aplicações científicas selecionadas

Fonte: O autor.

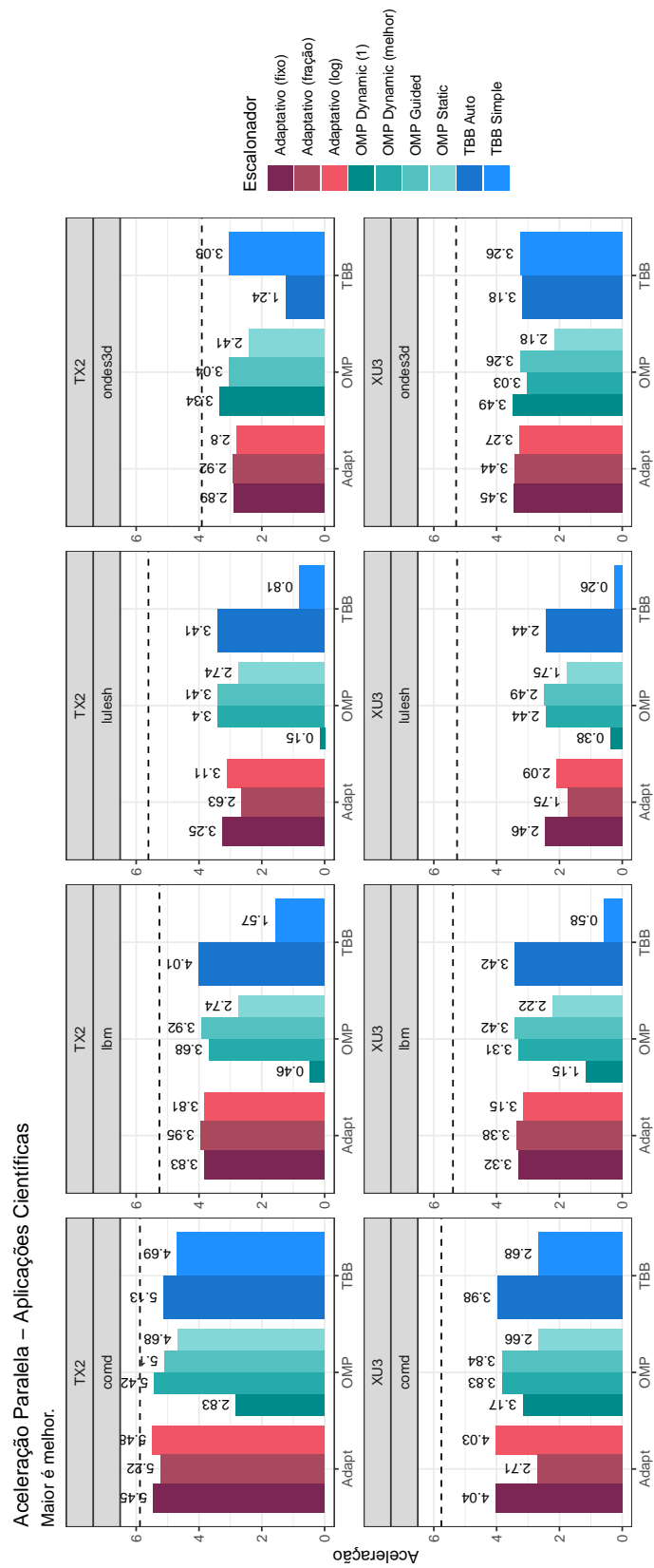


Gráfico 3.6 – Aceleração paralela com as aplicações científicas selecionadas. Linhas horizontais indicam a aceleração paralela máxima estimada a partir dos tempos de execução sequenciais, conforme descrito na Tabela 3.5.

Fonte: O autor.

O escalonador adaptativo também atingiu resultados interessantes dentre as aplicações científicas. Dos 8 cenários ilustrados, o escalonador conseguiu atingir menores tempos de forma isolada em 2 cenários (Gráfico 3.4), ambos da aplicação CoMD. Nos demais cenários, OpenMP possibilitou 5 melhores resultados, ao passo que o TBB possibilitou dois. Em dois destes cenários (LBM em XU3 e LULESH em TX2) OpenMP e TBB novamente compartilham os melhores resultados, quando levado em consideração o intervalo de confiança aplicado. Em contraste à situação analisada nos *benchmarks* NAS, neste cenário o escalonador adaptativo obteve melhores tempos na aplicação que apresenta algumas das menores assimetrias em tempo de execução entre os núcleos. Tal característica pode ser considerada como prova empírica que o nível de assimetria no sistema/aplicação não prejudica o escalonamento adaptativo.

Assim como nos cenários envolvendo os *benchmarks* NAS, o escalonador adaptativo manteve seu desempenho muito próximo das melhores opções. Nos 6 cenários onde não obteve os menores tempo, alcançou uma eficiência paralela não menor que 2.88% a menos que o melhor concorrente, com exceção da execução da aplicação Ondes3D no ambiente TX2 (Gráfico 3.5).

Uma característica comum entre três das aplicações utilizadas (CoMD, LBM e LULESH) é o desempenho relativamente inferior em casos com o grão muito fino. O impacto negativo no desempenho decorrente do uso de escalonadores com grão fino (OpenMP Dynamic (1) e TBB Simples) refletiu em acelerações paralelas muito próximas de 1 ou abaixo disso (Gráfico 3.6). Tal comportamento é observável em aplicações onde o custo médio para se computar uma iteração é baixo, mas a quantia de iterações por laço é grande o suficiente para se extrair desempenho com paralelismo. Na aplicação LBM, por exemplo, dois laços são responsáveis por 99.55% do tempo de execução sequencial e cada invocação necessita em média de 0.15 e 0.21 segundos no ambiente XU3. Os tempos médios por iteração nesses laços são 50.82 e 1.58 nanossegundos, inviabilizando um escalonamento dinâmico com o grão muito fino. Esse cenário serve pra ilustrar o impacto do uso de uma fila centralizada no escalonamento quando a concorrência por acesso é grande: 8 *threads* constantemente requisitam porções de iterações de tamanho diminuto e acabam ficando muito tempo ociosas devido à espera imposta pela exclusão mútua no acesso à fila.

A fim de avaliar o desempenho do escalonado adaptativo em cenários como esses, foram realizados experimentos adicionais no ambiente XU3 onde definiu-se um grão de tamanho fixo, o menor possível (1 iteração) para as aplicações LBM e LULESH. Os resultados mostram que a distribuição inicial das iterações entre as *threads*, assim como proposto pelo trabalho, associado pelo uso reduzido de travas de software durante a extração de trabalho, representaram ganhos de desempenho aproximados de 10% e 72% para as aplicações LBM e LULESH, respectivamente. Os ganhos são maiores em aplicações com maior custo computacional, e aumentam conforme a carga de trabalho aumenta, indicando uma maior escalabilidade por parte do escalonador adaptativo.

Dentre os 4 cenários em que o escalonador adaptativo proposto se mostrou a melhor opção, em 3 deles podemos encontrar um padrão de comportamento dos laços paralelos: tanto a aplicação CoMD quanto a CG possuem heterogeneidade nos tempos de cada iteração de seus laços com maior carga de trabalho. O laço principal da aplicação CoMD, responsável por calcular seu kernel Lennard-Jones para cálculo de energia potencial inter-atômica total, onde cada iteração corresponde à uma molécula calculando sua energia potencial a partir da interação com outras moléculas vizinhas em uma distância pré-estipulada. Como algumas moléculas podem ter mais moléculas vizinhas que outras, algumas iterações acabam sendo mais custosas computacionalmente que outras. Já para o *benchmark* CG, o laço com maior carga (94.4% do tempo sequencial) percorre uma matriz esparsa com cada iteração processando uma linha da matriz, com cada linha tendo quantidades diferentes de colunas. A linha com mais células tem, para o tamanho de entrada utilizado na avaliação, mais de 10 vezes o tamanho da linha com menos células. Tal característica em comum indica que o escalonador adaptativo pode se mostrar uma melhor opção para esse tipo de situação, como suposto durante os capítulos anteriores.

3.9 SUMÁRIO

Este Capítulo apresentou uma avaliação de desempenho do escalonador adaptativo proposto por este trabalho. Foram definidas métricas para avaliação (tempo de execução, aceleração paralela e eficiência paralela), plataformas embarcadas reais com AMPs (NVIDIA Jetson TX2), aplicações-alvo (conjunto de *benchmarks* NAS e as aplicações científicas CoMD, LBM, LULESH e Ondes3D) e ferramentas alternativas de escalonamento de laços paralelos referentes ao estado da arte para comparação com o escalonador proposto (OpenMP e IntelTBB).

Os resultados indicam uma vantagem pequena do escalonador adaptativo em cenários específicos previstos, e uma desvantagem relativamente pequena na maioria dos demais casos. Em poucos cenários onde o sobrecusto relacionado à criação de laços paralelos é posto a prova, o escalonador adaptativo mostra-se uma má solução quando em comparação as demais, ainda que nenhuma das soluções obtivessem um desempenho próximo do esperado ou sequer satisfatório. Esses cenários destacaram como a etapa interna de sincronização do escalonador não era eficiente o suficiente para o tornar competitivo, instigando possíveis soluções que contornem o problema ainda que parcialmente, como a substituição de barreiras e travas de exclusão mútua por soluções livre de travas.

Por fim, hipóteses foram traçadas sobre o desempenho do escalonador, como um potencial para escalabilidade em configurações com mais núcleos, onde o sobrecusto com concorrência na extração de trabalho se torne maior. Um estudo mais aprofundado sobre o impacto do tamanho do grão também pode ser conduzido em trabalhos futuros.

4 CONCLUSÃO

Esta dissertação apresentou a proposta e o desenvolvimento de um escalonador adaptativo de laços paralelos com foco em possibilitar uma maior extração de desempenho através do uso de roubo de trabalho em procesadores multinúcleo assimétricos. O escalonador permite a paralelização de laços comuns e laços com operações de redução implementados nas linguagens C e C++. O Capítulo 2 apresentou a fundamentação teórica na qual este trabalho se embasa, e os trabalhos relacionados que foram utilizados como inspiração. O Capítulo 3 apresentou o escalonador adaptativo proposto, ressaltando a forma como foi planejado e detalhes de sua implementação em C++. Por fim, o Capítulo 3.6 relatou uma análise de desempenho conduzida para atestar a eficiência do escalonador contra outras soluções bem consolidadas em ambientes embarcados assimétricos reais, apresentando metodologia, resultados e discussão dos mesmos.

É necessário salientar a importância do estudo comparativo com as soluções completas OpenMP e Intel TBB: ambas representam o estado da arte em computação de alto desempenho, sendo bibliotecas de uso extremamente difundido e de forte aceitação. Desenvolver uma solução que se aproxime em desempenho das duas já é por definição uma tarefa árdua, uma vez que ambas são desenvolvidas e mantidas por dezenas de pessoas capacitadas na área de HPC ao redor do mundo. Este trabalho, em adição ao escalonador, traz como contribuição adicional o desenvolvimento de uma biblioteca compacta que possibilita o seu uso de forma abstraída.

A análise de desempenho possibilitou delinear as seguintes considerações a respeito do escalonador desenvolvido:

- O uso do escalonador possibilita um melhor balanceamento, apesar de possuir uma vantagem pequena em comparação às soluções concorrentes, em laços com cargas de trabalho heterogêneas em custo computacional – isto é, iterações com tempo de execução diferentes – para os processadores utilizados na avaliação.
- Níveis opostos de assimetria estão presentes nas aplicações onde o escalonador adaptativo obteve melhores resultados: diferenças em desempenho entre os tipos de núcleo muito baixas (aplicação CoMD) e muito altas (*benchmark* CG). Tal característica pode ser interpretada como uma prova empírica que o nível de assimetria não tem impacto negativo no desempenho do escalonador devido a sua adaptabilidade, podendo o mesmo ser utilizado em ambientes simétricos ou assimétricos sem consequências no desempenho como um todo;
- O escalonador reduz o tempo gasto com exclusão mútua em extração de trabalho quando comparado à escalonadores que fazem uso de uma fila global centralizada de trabalho. O uso de variáveis atômicas para controle dos intervalos de iteração

pertencentes a cada *thread* permite a extração de trabalho sequencial o mais livre do uso de travas de software quanto é possível. Tal característica pode o tornar mais escalável que as outras soluções avaliadas em alguns cenários;

- O uso de barreiras com espera ociosa pode representar um gargalo no desempenho do escalonador em cenários que os laços não demandam carga computacional o suficiente pra tirar proveito da paralelização. Uma proposta de solução de implementação de barreiras com espera ocupada contorna parcialmente esse problema, reduzindo o tempo de execução em até 50% em alguns dos cenários avaliados. O uso deste recurso pode significar uma vantagem em sistemas com um número maior de núcleos que demandem mais tempo para sincronizar suas *threads* através de barreiras. A implementação de travas com espera ocupada também pode ser levada em consideração;
- Mesmo em casos no qual não se apresenta como a melhor solução, o escalonador permite extrair desempenhos muito similares às melhores soluções, com diferenças em eficiência paralela variando entre 1 a 7% em média.

Com base na análise realizada, concluímos que o escalonador desenvolvido cumpre o que lhe é proposto e tem potencial para ser mais escalável que as demais soluções avaliadas. O escalonador encontra-se disponível em um repositório Git¹ hospedado no Núcleo de Ciência da Computação da Universidade Federal de Santa Maria, bem como as versões adaptadas dos *benchmarks* NAS² e das aplicações científicas³, para o escalonador e para a biblioteca Intel TBB, considerados pelo autor como uma contribuição adicional do trabalho.

Estudos futuros podem incluir novas funcionalidades como laços com diferentes formas de progressão – atualmente a implementação só aceita o incremento de 1 no índice por iteração –; um estudo mais aprofundado do impacto de barreiras com espera ocupada no desempenho; o uso de travas de exclusão mútua com espera ocupada; avaliação em sistemas com um número ainda maior de núcleos, reais ou simulados; avaliação de *benchmarks* e bibliotecas diferentes; estudo de impacto do uso de diferentes tamanhos de grão para trabalhos sequenciais, dinâmicos ou não.

¹<<https://github.com/648trindade/adaptive>>

²<<https://gitlab.inf.ufsm.br/rtrindade/NAS-OMP-3.0>>

³<<https://gitlab.inf.ufsm.br/rtrindade/adaptive-experiments>>

AGRADECIMENTOS

Esta dissertação foi financiada em sua integralidade pelo CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) sob o edital número 132723/2018-2. Gostaríamos de agradecer à Nvidia Corporation por ceder a plataforma de desenvolvimento embarcada NVIDIA Jetson TX2 utilizada neste trabalho, através do programa *NVIDIA GPU Hardware Grant*.

REFERÊNCIAS BIBLIOGRÁFICAS

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: **Proceedings of the April 18-20, 1967, Spring Joint Computer Conference**. New York, NY, USA: ACM, 1967. (AFIPS '67 (Spring)), p. 483–485. Disponível em: <<http://doi.acm.org/10.1145/1465482.1465560>>.

ARM Limited. Online, **ARM Cortex-A57 Technical Reference Manuals**. 2013. Acesso em 7 Julho 2019. Disponível em: <<http://infocenter.arm.com/help/topic/com.arm.doc.subset.cortexa.a57/index.html>>.

_____. White Paper, **big.LITTLE Technology: The Future of Mobile**. 2013. Acesso em 6 novembro 2019. Disponível em: <https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf>.

BACKUS, J. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. **Communications of ACM**, ACM, New York, NY, USA, v. 21, n. 8, p. 613–641, ago. 1978. ISSN 0001-0782. Disponível em: <<http://doi.acm.org/10.1145/359576.359579>>.

BAILEY, D. et al. **The NAS Parallel Benchmarks**. Washington, DC, 1994. Disponível em: <<https://www.nas.nasa.gov/assets/pdf/techreports/1994/rnr-94-007.pdf>>.

BALAKRISHNAN, S. et al. The impact of performance asymmetry in emerging multicore architectures. In: **32nd International Symposium on Computer Architecture (ISCA'05)**. [s.n.], 2005. p. 506–517. Disponível em: <<http://doi.org/10.1109/ISCA.2005.51>>.

BRENT, R. P.; KUNG, H. T. Systolic VLSI arrays for polynomial GCD computation. **IEEE Transactions on Computers**, C-33, n. 8, p. 731–736, Aug 1984.

CASTILLO, E. et al. CATA: Criticality aware task acceleration for multicore processors. In: **2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [s.n.], 2016. p. 413–422. ISSN 1530-2075. Disponível em: <<http://doi.org/10.1109/IPDPS.2016.49>>.

CHAPMAN, B.; JOST, G.; VAN DER PAS, R. Introduction. In: _____. **Using OpenMP: Portable Shared Memory Parallel Programming**. MITP, 2007. cap. 1. Disponível em: <<https://ieeexplore.ieee.org/document/6280926>>.

CHRONAKI, K. et al. Task scheduling techniques for asymmetric multi-core systems. **IEEE Transactions on Parallel and Distributed Systems**, v. 28, n. 7, p. 2074–2087, July 2017. ISSN 1045-9219. Disponível em: <<https://doi.org/10.1109/TPDS.2016.2633347>>.

CICOTTI, P.; MNISZEWSKI, S. M.; CARRINGTON, L. An evaluation of threaded models for a classical md proxy application. In: **2014 Hardware-Software Co-Design for High Performance Computing**. [S.l.: s.n.], 2014. p. 41–48. ISSN null.

CONTRERAS, G.; MARTONOSI, M. Characterizing and improving the performance of Intel Threading Building Blocks. In: **2008 IEEE International Symposium on Workload Characterization**. [s.n.], 2008. p. 57–66. Disponível em: <<http://doi.org/10.1109/IISWC.2008.4636091>>.

COSTERO, L. et al. Refactoring conventional task schedulers to exploit asymmetric ARM big.LITTLE architectures in dense linear algebra. In: **2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. [s.n.], 2016. p. 692–701. Disponível em: <<http://doi.org/10.1109/IPDPSW.2016.104>>.

CUNG, V.-D. et al. Adaptive and Hybrid Algorithms: classification and illustration on triangular system solving. In: DUMAS, J.-G. (Ed.). **Transgressive Computing 2006**. Grenade, Spain: Copias Coca, Madrid, 2006. p. 131–148. Disponível em: <<https://hal.archives-ouvertes.fr/hal-00318540>>.

DIETZ, H. G.; COHEN, W. E. **A Massively Parallel MIMD Implemented by SIMD Hardware?** [S.l.], 1992.

DIETZ, H. G.; YOUNG, B. D. Mimd interpretation on a gpu. In: GAO, G. R. et al. (Ed.). **Languages and Compilers for Parallel Computing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 65–79. ISBN 978-3-642-13374-9.

DIMAKOPOULOS, V. V.; HADJIDOUKAS, P. E. Hompi: A hybrid programming framework for expressing and deploying task-based parallelism. In: JEANNOT, E.; NAMYST, R.; ROMAN, J. (Ed.). **Euro-Par 2011 Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 14–26. ISBN 978-3-642-23397-5.

DOLBEAU, R.; BIHAN, S.; BODIN, F. White Paper, **HMPP: A Hybrid Multi-core Parallel Programming Environment**. Caps Entreprise, 2006. Disponível em: <<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.357.3627>>.

DUPROS, F.; DO, H.-T.; AOCHI, H. On scalability issues of the elastodynamics equations on multicore platforms. **Procedia Computer Science**, v. 18, p. 1226–1234, 2013. ISSN 1877-0509. 2013 International Conference on Computational Science. Disponível em: <<https://doi.org/10.1016/j.procs.2013.05.289>>.

DURAN, A. et al. OmpSs: A proposal for programming heterogeneous multi-core architectures. **Parallel Processing Letters**, World Scientific, v. 21, n. 02, p. 173–93, 2011. Disponível em: <<https://doi.org/10.1142/S0129626411000151>>.

DURAND, M. et al. An efficient openmp loop scheduler for irregular applications on large-scale numa machines. In: RENDELL, A. P.; CHAPMAN, B. M.; MÜLLER, M. S. (Ed.). **OpenMP in the Era of Low Power Devices and Accelerators**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 141–155. ISBN 978-3-642-40698-0. Disponível em: <https://doi.org/10.1007/978-3-642-40698-0_11>.

EDWARDS, H. C.; TROTT, C. R.; SUNDERLAND, D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. **Journal of Parallel and Distributed Computing**, v. 74, n. 12, p. 3202 – 3216, 2014. ISSN 0743-7315. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0743731514001257>>.

FEDOROVA, A. et al. Maximizing power efficiency with asymmetric multicore systems. **Communications of ACM**, ACM, New York, NY, USA, v. 52, n. 12, p. 48–57, dez. 2009. ISSN 0001-0782. Disponível em: <<http://doi.acm.org/10.1145/1610252.1610270>>.

FLYNN, M. J.; RUDD, K. W. Parallel architectures. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 28, n. 1, p. 67–70, mar. 1996. ISSN 0360-0300. Disponível em: <<http://doi.acm.org/10.1145/234313.234345>>.

FOX, G. C. et al. **Solving Problems on Concurrent Processors. Vol. 1: General Techniques and Regular Problems**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN 0-13-823022-6.

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The implementation of the cilk-5 multithreaded language. In: **Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation**. New York, NY, USA: Association for Computing Machinery, 1998. (PLDI '98), p. 212–223. ISBN 0897919874. Disponível em: <<https://doi.org/10.1145/277650.277725>>.

GEORGOPOULOS, G. **Memory Consistency Models of Modern CPUs**. 2016. Acesso em 24 Junho 2019. Disponível em: <<https://es.cs.uni-kl.de/publications/datarsg/Geor16.pdf>>.

GREGORY, K.; MILLER, A. **C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++**. [S.l.]: Microsoft Press, 2012. ISBN 978-0735664739.

GROSCH, H. R. J. High speed arithmetic: The digital computer as a research tool. **Journal of the Optical Society of America**, OSA, v. 43, n. 4, p. 306–310, Apr 1953. Disponível em: <<http://doi.org/10.1364/JOSA.43.000306>>.

Intel Corporation. Online, **Cilk Plus**. 2019. Acesso em 11 novembro 2019. Disponível em: <<https://www.cilkplus.org/>>.

_____. Online, **Intel Threading Building Blocks Documentation**. 2019. Acesso em 11 novembro 2019. Disponível em: <<https://www.threadingbuildingblocks.org/docs/help/index.htm>>.

ISO. **ISO/IEC JTC1 SC22 WG21 N3690. Programming Languages – C++**. [s.n.], 2013. 1139–1141 p. Disponível em: <<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf>>.

KALÉ, L.; KRISHNAN, S. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In: PAEPCKE, A. (Ed.). **Proceedings of OOPSLA'93**. [S.l.]: ACM Press, 1993. p. 91–108. ISBN 0-89791-587-9.

KALINOV, A. Measuring the scalability of heterogeneous parallel systems. In: WYRZYKOWSKI, R. et al. (Ed.). **Parallel Processing and Applied Mathematics**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 1066–1073. ISBN 978-3-540-34142-0. Disponível em: <https://doi.org/10.1007/11752578_129>.

KARLIN, I. **LULESH Programming Model and Performance Ports Overview**. United States, 2012. Disponível em: <<https://doi.org/10.2172/1059462>>.

KUMAR, R. et al. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In: **Proceedings. 31st Annual International Symposium on Computer Architecture, 2004**. [S.l.: s.n.], 2004. p. 64–75.

MARTÍNEZ, V. et al. Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system. In: **2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. [s.n.], 2015. p. 1–8. ISSN 1550-6533. Disponível em: <<https://doi.org/10.1109/SBAC-PAD.2015.33>>.

MEADOWS, L.; ISHIKAWA, K.-i. OpenMP tasking and MPI in a lattice QCD benchmark. In: SUPINSKI, B. R. de et al. (Ed.). **Scaling OpenMP for Exascale Performance and Portability**. Cham: Springer International Publishing, 2017. p. 77–91. ISBN 978-3-319-65578-9. Disponível em: <https://doi.org/10.1007/978-3-319-65578-9_6>.

MENASCÉ, D.; ALMEIDA, V. Cost-performance analysis of heterogeneity in supercomputer architectures. In: **Proceedings of the 1990 ACM/IEEE Conference on Supercomputing**. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990. (Supercomputing '90), p. 169–177. ISBN 0-89791-412-0. Disponível em: <<http://dl.acm.org/citation.cfm?id=110382.110430>>.

MENON, H. M. G. **Adaptive load balancing for HPC applications**. Out 2016. Tese (Doutorado) — University of Illinois at Urbana-Champaign, Out 2016. Disponível em: <<http://hdl.handle.net/2142/95292>>.

MITTAL, S. A survey of techniques for architecting and managing asymmetric multicore processors. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 48, n. 3, p. 45:1–45:38, fev. 2016. ISSN 0360-0300. Disponível em: <<http://doi.acm.org/10.1145/2856125>>.

MOHD-YUSOF, J.; SWAMINARAYAN, S.; GERMANN, T. Co-Design for Molecular Dynamics: An Exascale Proxy Application. In: **LA-UR 13-20839**. [s.n.], 2013. p. 88–89. Disponível em: <http://www.lanl.gov/orgs/adts/publications/science_highlights_2013/docs/Pg88_89.pdf>.

MÓR, S. D. K. **Analysis of Synchronizations in Greedy-Scheduled Executions and Applications to Efficient Generation of Pseudorandom Numbers in Parallel**. nov 2015. Tese (Thesis) — Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brazil, nov 2015. Disponível em: <<http://hdl.handle.net/10183/130529>>.

NVIDIA Developer. Online, **Hardware For Every Situation**. 2019. Acesso em 19 Junho 2019. Disponível em: <<https://developer.nvidia.com/embedded/develop/hardware>>.

ODROID Wiki. **old product: odroid-xu3**. 2017. Wiki. Acesso em 19 Junho 2019. Disponível em: <https://wiki.odroid.com/old_product/odroid-xu3/odroid-xu3>.

OpenACC Organization. Online, **OpenACC**. 2019. Acesso em 19 Nov 2019. Disponível em: <<https://www.openacc.org/>>.

OpenMP Architecture Review Board. Online, **OpenMP Application Program Interface Version 5**. 2018. Disponível em: <<https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>>.

QUINTIN, J.-N.; WAGNER, F. Hierarchical work-stealing. In: D'AMBRA, P.; GUARRACINO, M.; TALIA, D. (Ed.). **Euro-Par 2010 - Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 217–229. ISBN 978-3-642-15277-1.

SCHEPKE, C.; MAILLARD, N. Performance improvement of the parallel lattice boltzmann method through blocked data distributions. In: **19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07)**. [s.n.], 2007. p. 71–78. ISSN 1550-6533. Disponível em: <<https://doi.org/10.1109/SBAC-PAD.2007.12>>.

TRAORÉ, D. et al. Deque-free work-optimal parallel STL algorithms. In: LUQUE, E.; MARGALEF, T.; BENÍTEZ, D. (Ed.). **Euro-Par 2008 – Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 887–897. ISBN 978-3-540-85451-7. Disponível em: <https://doi.org/10.1007/978-3-540-85451-7_95>.

TRINDADE, R. G.; LIMA, J. ao V. F. Estudo de Implementação com Laços Paralelos em OpenMP 4 com Offloading para GPU no Método de Lattice Boltzmann. In: **Workshop de Iniciação Científica do WSCAD 2017 (XVIII Simpósio em Sistemas Computacionais de Alto Desempenho) WSCAD-WIC 2017**. Sociedade Brasileira de Computação (SBC), 2017. p. 27–32. ISSN 2358-6613. Disponível em: <<http://www.lbd.dcc.ufmg.br/bdbcomp/servlet/Trabalho?id=25743>>.

WASSON, S. Online, **Nvidia claims Haswell-class performance for Denver CPU core**. 2014. Acesso em 4 novembro 2019. Disponível em: <<https://techreport.com/news/26906/nvidia-claims-haswell-class-performance-for-denver-cpu-core/>>.