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

Gabriel Freytag

UMA IMPLEMENTAÇÃO BASEADA EM TAREFAS E FLUXO DE DADOS DO MÉTODO DE LATTICE-BOLTZMANN

Santa Maria, RS
2018

Gabriel Freytag

UMA IMPLEMENTAÇÃO BASEADA EM TAREFAS E FLUXO DE DADOS DO MÉTODO DE LATTICE-BOLTZMANN

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação, da Universidade Federal de Santa Maria (UFSM), como requisito parcial para obtenção do título de **Mestre em Ciência da Computação**.

Orientador: Dr. João Vicente Ferreira Lima

Santa Maria, RS
2018

Freytag, Gabriel
Uma Implementação Baseada em Tarefas e Fluxo de Dados do Método de Lattice-Boltzmann / Gabriel Freytag.- 2018.
105 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, 2018

1. Lattice-Boltzmann 2. Paralelismo de Tarefas 3. Fluxo de Dados 4. OpenMP 5. NUMA I. Lima, João Vicente Ferreira II. Título.

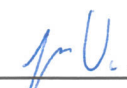
Sistema de geração automática de ficha catalográfica da UFSM. Dados fornecidos pelo autor(a). Sob supervisão da Direção da Divisão de Processos Técnicos da Biblioteca Central. Bibliotecária responsável Paula Schoenfeldt Patta CRB 10/1728.

Gabriel Freytag

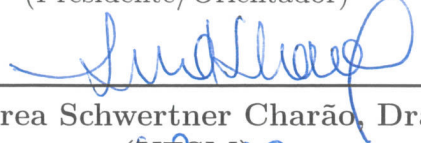
UMA IMPLEMENTAÇÃO BASEADA EM TAREFAS E FLUXO DE DADOS DO MÉTODO DE LATTICE-BOLTZMANN

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação, da Universidade Federal de Santa Maria (UFSM), como requisito parcial para obtenção do título de **Mestre em Ciência da Computação**.

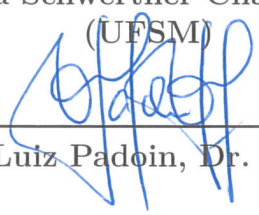
Aprovado em 22 de fevereiro de 2018:



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



Andrea Schwertner Charão, Dra.
(UFSM)



Edson Luiz Padoin, Dr. (UNIJUI)

Santa Maria, RS, Brasil

2018

Este trabalho é dedicado aos meus pais e à minha namorada.
Aos meus pais Silvério Freytag e Rosane M. Freytag, fontes da minha inspiração.
À minha namorada Andréia Luisa Friske, aconchego do meu coração.

Agradecimentos

Aos meus pais **Silvério Freytag** e **Rosane Maria Freytag** e aos meus sogros **Ivanir Elaine Hermann** e **Arteno Ilson Friske** pelo apoio, suporte, paciência e confiança. À minha namorada **Andréia Luisa Friske** pelo carinho, paciência, apoio, confiança e companheirismo.

Ao meu orientador, **professor Dr. João Vicente Ferreira Lima**, por aceitar orientar-me no Mestrado, pela paciência que teve ao longo dos últimos dois anos e por compartilhar seu vasto conhecimento e conselhos comigo.

À **todos os professores** que, ao longo do Mestrado, tiveram a humildade de compartilhar seu conhecimento, além da amizade e da atenção dedicada.

À **Deus**, por permitir-me estar entre todas essas pessoas especiais.

Sem vocês, nada disso seria possível.

Muito obrigado a todos!

"Aprenda como se você fosse viver para sempre.
Viva como se você fosse morrer amanhã.
(Mohandas Karamchand Gandhi)

Resumo

Lattice-Boltzmann é um método numérico iterativo para a modelagem mesoscópica e simulação da dinâmica de fluxos de fluidos. Esse método simula as propriedades discretas de sistemas físicos e, para que as simulações sejam realizadas em um tempo computacionalmente aceitável, exige um grande poder computacional. Diversos estudos na literatura dedicam-se à paralelização e avaliação do desempenho do método de Lattice-Boltzmann aplicado a uma variedade de problemas utilizando uma ampla gama de arquiteturas de computação de alto desempenho, desde arquiteturas de memória compartilhada, distribuída e híbrida até aceleradores GPU, Xeon Phi, entre outros. No cenário atual em que fabricantes de processadores praticamente duplicam a quantidade de transistores em um mesmo chip a cada nova geração dedicando-os à replicação de núcleos, a exploração eficiente do crescente paralelismo oferecido especialmente por arquiteturas de computação de alto desempenho com memória compartilhada depende da adoção de técnicas de paralelismo que otimizem a execução de aplicações nessas arquiteturas. Uma das técnicas de paralelismo mais populares nesse tipo de arquitetura é o paralelismo de iterações de laços de repetição utilizando a API OpenMP. Apesar de proporcionar ganhos significativos de desempenho, o paralelismo oferecido por esse tipo de arquitetura nem sempre é explorado em sua totalidade e técnicas como a de paralelismo de tarefas podem ser utilizadas a fim de otimizar a exploração do paralelismo por aplicações. Embora o paralelismo de tarefas seja suportado desde a versão 3.0 da API OpenMP, o conceito de dependência de dados entre tarefas foi introduzido somente na versão 4.0. Por meio da especificação das dependências de dados é possível adicionar restrições ao escalonamento de tarefas de modo que a ordem de execução seja determinada conforme as operações de leitura e escrita realizadas por cada tarefa em endereços na memória evitando, assim, inconsistências na execução paralela de tarefas. Por se tratar de um método iterativo, a paralelização do método de Lattice-Boltzmann utilizando tarefas exige que a cada nova iteração as diferentes tarefas sejam sincronizadas, que pode ser realizada a partir da determinação das dependências de cada tarefa. Portanto, o objetivo deste trabalho é apresentar e avaliar o desempenho de uma implementação baseada em tarefas e fluxo de dados do método de Lattice-Boltzmann utilizando tarefas OpenMP com dependências. Os resultados experimentais realizados em uma arquitetura de memória compartilhada NUMA composta por 48 núcleos de processamento mostram que o desempenho da implementação baseada em tarefas com dependências foi até 22,49% melhor se comparado ao desempenho obtido por uma implementação baseada no paralelismo de iterações dos laços de repetição.

Palavras-chave: Lattice-Boltzmann. Paralelismo de Tarefas. Fluxo de Dados. OpenMP. NUMA.

Abstract

Lattice-Boltzmann is an iterative numerical method for mesoscopic modeling and simulation of fluid flow dynamics. This method simulates the discrete properties of physical systems and, for the simulations to be performed in a computationally acceptable time, requires a great computational power. Several studies in the literature are dedicated to parallelizing and evaluating the performance of the Lattice-Boltzmann method applied to a variety of problems using a wide range of high-performance computing architectures ranging from shared, distributed and hybrid memory architectures to GPU accelerators, Xeon Phi, among others. In today's scenario where processor manufacturers practically double the number of transistors on a single chip with each new generation dedicating them to core replication, the efficient exploitation of the increasing parallelism offered especially by shared memory high performance computing architectures depends on the adoption of parallelism techniques that optimize the execution of applications in these architectures. One of the most popular parallelism techniques in this type of architecture is the parallelism of loop iterations using the OpenMP API. In spite of providing significant performance gains, the parallelism offered by this type of architecture is not always exploited in its entirety and techniques such as task parallelism can be used to optimize the exploitation of parallelism by applications. Although task parallelism has been supported since version 3.0 of the OpenMP API, the concept of data dependency between tasks has only been introduced in version 4.0. By specifying the data dependencies it is possible to add constraints to the scheduling of tasks so that the execution order is determined according to the read and write operations performed by each task in memory addresses, thus avoiding inconsistencies in the parallel execution of tasks. Because it is an iterative method, the parallelization of the Lattice-Boltzmann method using tasks requires that at each new iteration the different tasks be synchronized, which can be performed from the determination of the dependencies of each task. Therefore, the objective of this work is to present and evaluate the performance of a task-based and data flow implementation of the Lattice-Boltzmann method using OpenMP tasks with dependencies. The results of experiments performed on a NUMA shared memory architecture composed of 48 processing cores show that the performance of the task-based implementation with dependencies was up to 22.49% better when compared to the performance achieved by an implementation based on loop-level parallelism.

Keywords: Lattice-Boltzmann. Task Parallelism. Data Flow. OpenMP. NUMA.

Lista de ilustrações

Figura 1 – Representação das regras de colisão do modelo FHP em um reticulado triangular na qual setas denotam partículas microscópicas e suas direções de movimentação. 22
Figura 2 – Sub-reticulados com os respectivos vetores de velocidade q_0, q_1, q_2 e q_3 representados por setas. 22
Figura 3 – A geometria do reticulado D3Q19. 23
Figura 4 – Funcionamento do mecanismo de *bounce-back*. 26
Figura 5 – Taxonomia de Flynn. 30
Figura 6 – Representação de sistemas de memória compartilhada, distribuída e híbrida. 33
Figura 7 – Modelo de programação fork-join utilizado pela API OpenMP. 34
Figura 8 – Representação do particionamento do domínio. 41
Figura 9 – Divisão e vizinhança ortogonal. 43
Figura 10 – Divisão e vizinhança diagonal. 45
Figura 11 – Gráfico de dependências da implementação OpenMP Tarefas. 61
Figura 12 – Gráfico de dependências da implementação OpenMP Tarefas com Buffers. 70
Figura 13 – Arquitetura do sistema SGI UV2000. 76
Figura 14 – Sobrecarga paralela T_1/T_s das implementações paralelas OpenMP. . . . 78
Figura 15 – Sobrecarga paralela T_1/T_s da implementação baseada em tarefas OpenMP (sem `buffers`) utilizando apenas inversão da ordem dos laços de repetição (Inv), apenas matrizes 1D (SoA) e sem inversão de laços e com matrizes 3D (sem SoA Inv). O tamanho do reticulado utilizado nos experimentos foi de 128. 80
Figura 16 – Tempo de execução das implementações OpenMP nos particionamentos que obtiveram os melhores tempos de execução em cada tamanho de reticulado em 12, 24, 36 e 48 threads. 81
Figura 17 – Melhores tempos de execução e respectivos MFLUP/s obtidos dentre as variações de threads e particionamentos de domínio utilizados nas implementações OpenMP for, baseada em tarefas e baseada em tarefas com buffers. 82
Figura 18 – Quantidade de memória DRAM local e remota transferida por ambas as implementações OpenMP nos particionamentos que obtiveram os melhores tempos de execução em 12, 24, 36 e 48 threads. 83

Figura 19 – Quantidade de memória DRAM local e remota transferida por ambas as implementações OpenMP nos experimentos que obtiveram os melhores tempos de execução dentre as variações de threads e particionamento dos reticulados. 84
Figura 20 – Quantidade total de memória DRAM transferida pelas implementações OpenMP nos particionamentos que obtiveram os melhores tempos de execução em 12, 24, 36 e 48 threads. 85
Figura 21 – Quantidade total de memória DRAM transferida por ambas as implementações OpenMP nos experimentos que obtiveram os melhores tempos de execução dentre as variações de threads e particionamento dos reticulados. 86
Figura 22 – Ciclos Por Instrução (CPI) utilizados pelas implementações OpenMP nos particionamentos que obtiveram os melhores tempos de execução em 12, 24, 36 e 48 threads. 87
Figura 23 – Ciclos Por Instrução (CPI) utilizados pelas implementações OpenMP nos experimentos que obtiveram os melhores tempos de execução dentre as variações de threads e particionamento dos reticulados. 87
Figura 24 – Tempo de execução com o melhor particionamento para cada reticulado e com alocação de memória intercalada (On) e padrão (Off) nas implementações OpenMP for e baseada em tarefas com buffers em 12, 24, 36 e 48 threads. 89
Figura 25 – Tempo de execução das implementações OpenMP for e baseada em tarefas com buffers compiladas utilizando GCC e Clang em 12, 24, 36 e 48 threads. 90
Figura 26 – Ciclos Por Instrução (CPI) nas implementações OpenMP for e baseada em tarefas com buffers compiladas utilizando GCC e Clang em 12, 24, 36 e 48 threads. 90
Figura 27 – Melhores tempos de execução obtidos com os melhores particionamentos para cada reticulado nas implementações utilizando somente SoA, utilizando somente inversão dos laços de repetição (Inv), não utilizando nem SoA nem a inversão dos laços, utilizando SoA juntamente com a inversão dos laços e utilizando SoA juntamente com a inversão dos laços e buffers em 12, 24, 36 e 48 threads. 92
Figura 28 – Tempo de execução em acelerador Xeon Phi KNL. 103
Figura 29 – Tempo de execução. 105
Figura 30 – Proporção do tempo de execução. 105

Lista de tabelas

Tabela 1 – Abordagens de simulação de acordo com o nível da simulação. 20
Tabela 2 – Velocidade relativa das partículas para cada uma das direções dos sub-reticulados. 23
Tabela 3 – Peso dos fatores w_i para os vetores de velocidades discretas e_i dos modelos de reticulado 3D mais populares. 25
Tabela 4 – Configurações nas quais foram obtidos os melhores tempos de execução em cada implementação OpenMP em alguns tamanhos de reticulado. . 93

Lista de abreviaturas e siglas

API	Application Programming Interface
BE	Boltzmann Equation
BGK	Bhatnagar-Gross-Krook
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DDR3	Double Data Rate type Three
DNS	Direct Numerical Simulations
DRAM	Dynamic Random Access Memory
DSM	Distributed Shared Memory
ELB	Entropic Lattice-Boltzmann
FHP	Frisch-Hasslacher-Pomeau
GB	Giga Bytes
GHz	Giga Hertz
GPU	Graphic Processor Units
HPP	Hardy-Pazzis-Pomeau
I/O	Input/Output
LBE	Lattice-Boltzmann Equation
LBM	Lattice-Boltzmann Method
LES	Large Eddy Simulations
LGA	Lattice Gas Automata
MFLUP/s	Million Fluid Lattice Updates Per Second
MIMD	Multiple Instruction stream and Multiple Data stream

MISD	Multiple Instruction stream and Single Data stream
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
NS	Navier-Stokes
OpenMP	Open Multi-Processing
PU	Processing Unit
RAM	Random Access Memory
SIMD	Single Instruction stream and Multiple Data stream
SISD	Single Instruction stream and Single Data stream
SMP	Symmetric Multiprocessor
SoA	Structure of Array
UMA	Uniform Memory Access

Sumário

Introdução 16

1 MÉTODO DE LATTICE-BOLTZMANN 19
1.1 Descrição do Método de Lattice-Boltzmann 19
1.2 Modelos de Reticulados 21
1.3 Equação de Lattice-Boltzmann 23
1.4 Condições de Contorno 25
1.5 O Algoritmo 26
1.6 Áreas de Aplicação 27
1.7 Considerações do Capítulo 27

2 PARALELISMO EM APLICAÇÕES E ARQUITETURAS 28
2.1 Classes de Paralelismo 28
2.2 Arquiteturas de Memória Compartilhada 30
2.3 Arquiteturas de Memória Distribuída 32
2.4 Interface de Programação Paralela OpenMP 33
2.5 Trabalhos Relacionados 36
2.6 Considerações do Capítulo 39

3 IMPLEMENTAÇÕES 40
3.1 Particionamento de Domínio 40
3.1.1 Vizinhos Ortogonais 42
3.1.2 Vizinhos Diagonais 44
3.2 MPI 46
3.3 OpenMP Tarefas 49
3.3.1 Dependências entre Tarefas 52
3.3.1.1 Tarefa de Inicialização 52
3.3.1.2 Tarefa de Redistribuição 54
3.3.1.3 Tarefa de Propagação 55
3.3.1.4 Tarefa de Cópia de Bordas 56
3.3.1.5 Tarefa de Bounceback 58
3.3.1.6 Tarefa de Relaxação 59
3.3.2 Escalonamento de Tarefas 59
3.4 OpenMP Tarefas com Buffers 64
3.4.1 Dependências das Tarefas 65
3.4.1.1 Tarefa de Cópia das Bordas para os Buffers 65

3.4.1.2 Tarefa de Cópia das Bordas a Partir dos Buffers 68
3.4.2 Escalonamento das Tarefas 68
3.5 OpenMP For 72
3.6 Considerações do Capítulo 73

4 RESULTADOS EXPERIMENTAIS 75
4.1 Ambiente de Teste e Plataformas de Experimentação 75
4.2 Sobrecarga Paralela 78
4.3 Avaliação de Desempenho 79
4.4 Análise e Discussão dos Resultados 88
4.5 Considerações do Capítulo 94

Conclusão 95

REFERÊNCIAS 97

ANEXOS 102

ANEXO A – IMPLEMENTAÇÃO OPENMP TAREFAS 103

ANEXO B – IMPLEMENTAÇÃO MPI+OPENMP TAREFAS . . . 104

Introdução

Desde a introdução do primeiro microprocessador o desempenho de aplicações estava relacionado diretamente ao aumento do desempenho provido por parte dos fabricantes do único núcleo de processamento que compôs os microprocessadores até o início do século XXI. Ao atingir os limites práticos de dissipação de calor com o aumento da frequência do *clock* de microprocessadores monolíticos, as fabricantes de microprocessadores começaram a dedicar o crescente número de transistores, adicionados geração após geração, à replicação de núcleos a fim de continuar aumentando o desempenho de microprocessadores. Essa mudança de paradigma fez com que o desempenho de aplicações não dependesse mais do aumento do desempenho de um único núcleo e, sim, da exploração do paralelismo oferecido por múltiplos núcleos de processamento (*multicore*) em um mesmo microprocessadores.

Atualmente, tanto a complexidade quanto o paralelismo em microprocessadores aumentam consideravelmente a cada nova geração. Isso é resultado da duplicação do número de transistores aproximadamente a cada dois anos, conforme previsão de Moore (1965), que, de acordo com Larus (2009), são utilizados para aumentar o número de núcleos independentes disponíveis em microprocessadores. Um exemplo disso é o microprocessador IBM Power9 que contém 8 bilhões de transistores e 24 núcleos individuais de processamento (SADASIVAM et al., 2017), praticamente o dobro da geração anterior IBM Power8 que possui 4,2 bilhões de transistores e 12 núcleos (ZYUBAN et al., 2015).

O crescente paralelismo presente em arquiteturas *multicore* e *manycore*[1] é amplamente explorado em sistemas de Processamento de Alto Desempenho (*High Performance Computing* – HPC), onde essas são aglomeradas às centenas ou aos milhares resultando na disponibilidade de um número elevado de núcleos de processamento e, consequentemente, de um grande poder computacional. No entanto, para que aplicações se beneficiem de todo esse poder computational, é necessária a utilização de linguagens de programação paralela como OpenMP, TBB, Cilk, FastFlow, MPI, CUDA, OpenCL, OpenACC, entre outras, que possibilitam a execução simultânea de diferentes partes ou estágios de uma aplicação nos diversos núcleos e, assim, aceleram sua execução.

Algumas das aplicações que mais se beneficiam do nível elevado de paralelismo presente em sistemas HPC são aplicações de computação intensiva e de dados intensivos. Essas aplicações normalmente demandam um grande poder computacional para que sejam executadas em um tempo computacionalmente aceitável. Aplicações de previsão meteorológica (ELLIOTT et al., 2014), stencil (ENDO; TAKASAKI; MATSUOKA, 2015) e várias aplicações científicas como de álgebra linear (LIMA et al., 2017), bem como aplicações

[1] Processadores *multicore* especialistas com um alto nível de paralelismo proveniente de um número elevado de núcleos de processamento, normalmente centenas e até milhares de núcleos.

de simulação astronômica (BUNTEMEYER et al., 2016), termodinâmica (BAUER et al., 2015), fluidodinâmica (BLOCKEN, 2015) e de propagação de ondas sísmicas (MARTÍNEZ et al., 2015) são apenas alguns exemplos de aplicações de computação intensiva.

Lattice-Boltzmann é um método numérico iterativo para modelagem mesoscópica e simulação da dinâmica dos fluxos de fluidos (SCHEPKE; DIVERIO, 2007). Este método intensivo em computação simula sistemas físicos e demanda um poder computacional elevado para ser simulado em um tempo computacionalmente aceitável. Várias implementações paralelas do método de Lattice-Boltzmann (*Lattice-Boltzmann Method* – LBM) desenvolvidas em linguagens de programação como MPI (SCHEPKE; MAILLARD; NAVAUX, 2009), OpenMP (BAŞAĞAOĞLU et al., 2017b), híbrido MPI+OpenMP (CLAUSEN; REASOR; AIDUN, 2010), MPI+CUDA (KRAUS et al., 2013), híbrido MPI+OpenMP+CUDA (FEICHTINGER et al., 2015) podem ser encontradas na literatura explorando o paralelismo oferecido por uma diversidade de sistemas HPC. Embora existam algumas implementações paralelas baseadas em tarefas OpenMP (DONATH et al., 2008), há uma predominância de implementações OpenMP baseadas no paralelismo de iterações de laços de repetição. Além disso, a maioria dos estudos que paralelizam o LBM utilizam modelos de reticulado bidimensionais ao invés de modelos tridimensionais, uma vez que esses possuem uma complexidade significativamente menor devido ao número reduzido de dependências envolvidas na simulação em sistemas paralelos em comparação aos modelos tridimensionais.

Portanto, neste trabalho é apresentada uma implementação baseada em tarefas e fluxo de dados do método de Lattice-Boltzmann utilizando um modelo de reticulado tridimensional no qual as forças de um fluido se propagam em 19 direções (denominado D3Q19) e utilizando tarefas OpenMP com dependências. Além disso, é apresentada uma segunda implementação na qual, com o auxílio de estruturas de dados temporárias (`buffers`), são eliminas dependências presentes na implementação proposta inicialmente com o intuito de otimizar sua execução paralela. A avaliação do desempenho de ambas as implementações baseadas em tarefas com dependências é realizada em comparação ao desempenho de uma terceira implementação, baseada no paralelismo de iterações de laços de repetição, a partir de experimentos conduzidos em uma arquitetura *multicore* de memória compartilhada composta por 48 núcleos de processamento. Apresenta-se ainda uma avaliação do desempenho de ambas as implementações compiladas a partir do compilador Clang 4.0, em conjunto com a biblioteca libKOMP, e o compilador GCC 7.2. Ambas as implementações do método de Lattice-Boltzmann apresentadas neste trabalho foram desenvolvidas a partir do trabalho apresentado por Schepke e Diverio (2007).

Este trabalho está organizado da seguinte forma. No Capítulo 1 descreve-se o método de Lattice-Boltzmann, o modelo de reticulado D3Q19, a Equação de Lattice-Boltzmann que fundamenta o método, as condições de contorno do reticulado, o algoritmo e as áreas de aplicação do método. Em seguida, no Capítulo 2, são descritas as classes

de paralelismo existentes, as arquiteturas de memória compartilhada e distribuída, uma metodologia popularmente utilizada no projeto de aplicações paralelas, a interfaces de programação paralela OpenMP e os trabalhos relacionados à paralelização do LBM. No Capítulo 3, por sua vez, descreve-se o particionamento e a vizinhança do domínio bem como as implementações MPI na qual este trabalho está baseado, a implementação baseada em tarefas OpenMP com dependências, a implementação baseada em tarefas OpenMP com dependências otimizada com o auxílio de `buffers` e a implementação baseada no paralelismo de iterações de laços de repetição. Por fim, são apresentados os resultados obtidos por meio de experimentos realizados em uma arquitetura paralela de ambas as implementações OpenMP no Capítulo 4, bem como as conclusões às quais foi possível chegar a partir da análise dos resultados dos experimentos e as sugestões de possíveis trabalhos futuros no capítulo de Conclusão.

1 Método de Lattice-Boltzmann

Neste capítulo descrever-se-á o método de Lattice-Boltzmann apresentando as principais características que o destacam de outros métodos convencionais, especialmente as características que o tornam um método simples e altamente escalável em sistemas de processamento paralelo. Além disso, será apresentado ainda o modelo de reticulado utilizado nas simulações, a equação de Lattice-Boltzmann que constitui o método, as condições de contorno que definem o comportamento das partículas em casos de colisão com os contornos do reticulado e o algoritmo do método utilizado neste trabalho. Por fim, apresentar-se-ão algumas aplicações do método na simulação computacional da dinâmica de fluidos para demonstrar as capacidades do método de Lattice-Boltzmann.

1.1 Descrição do Método de Lattice-Boltzmann

O Método de Lattice-Boltzmann é um processo numérico para a simulação da dinâmica de fluidos e modelagem de propriedades físicas de fluidos que se originou a partir do Autômato de Lattice Gas (*Lattice Gas Automata* – LGA), uma cinética discreta de partículas que utiliza tanto uma estrutura discreta quanto um tempo discreto (CHEN; DOOLEN, 1998). No entanto, apesar de ser originado a partir do LGA, que utiliza variáveis booleanas, no LBM o valor médio e a evolução do tempo das populações são definidas por meio de variáveis reais (MCNAMARA; ZANETTI, 1988). A vantagem da utilização de variáveis reais ao invés de variáveis booleanas é a eliminação do ruído estatístico que geralmente afeta as simulações dos Autômatos de Lattice Gas (BENZI; SUCCI; VERGASSOLA, 1992).

A principal característica do LBM é a simplicidade na representação de microssistemas. Esse método computacional foi desenvolvido para o estudo numérico da equação de Navier-Stokes (NS) com base na simulação de um sistema microscópico relativamente simples ao invés da integração direta de equações diferenciais parciais (MCNAMARA; ZANETTI, 1988). O desenvolvimento do LBM foi baseado na ideia de que, devido à correspondência não singular existente entre um sistema fluido e um microssistema, é possível construir microssistemas artificiais que são simples o suficiente para serem simulados em um computador mas que, mesmo assim, ainda contêm toda a física requerida de um sistema fluido realista (SHAN; CHEN, 1993).

O método de Lattice-Boltzmann é baseado em modificações microscópicas e equações cinéticas mesoscópicas, ao contrário de métodos numéricos convencionais tais como o método dos Elementos Finitos ou o método dos Volumes Finitos que são baseados em discretizações de equações de continuidade macroscópicas (CHEN; DOOLEN, 1998).

Como o método possui raízes na teoria cinética e no conceito de autômato celular, a Equação de Lattice-Boltzmann (*Lattice-Boltzmann Equation* – LBE) pode ser usada para obter quantidades de fluxo contínuo a partir de regras de atualização simples e locais baseadas nas interações de partículas (AIDUN; CLAUSEN, 2010). De acordo com Chen e Doolen (1998), a ideia fundamental do LBM é construir modelos cinéticos simplificados que incorporem a física essencial dos processos microscópicos ou mesoscópicos de modo que as propriedades macroscópicas médias obedeçam às equações macroscópicas desejadas. Na Tabela 1 são apresentadas as abordagens de simulação utilizadas de acordo com os níveis de simulação macroscópica, mesoscópica e microscópica.

Tabela 1 – Abordagens de simulação de acordo com o nível da simulação.

Nível de simulação	**Abordagem para simulação**
Macroscópica	Meios contínuos (Elementos Finitos)
Mesoscópica	Mecânica estatística (Lattice-Boltzmann)
Microscópica	Dinâmica molecular

Fonte: O autor.

Conforme Chen e Doolen (1998), a natureza cinética do LBM apresenta três características importantes que a distinguem de outros métodos numéricos: o operador de convecção (deslocamento das partículas) no espaço de fase (ou espaço de velocidade) é linear ao contrário dos termos de convecção não-linear em outras abordagens com representação macroscópica; as equações de Navier-Stokes incompressíveis podem ser obtidas no limite quase incompressível do LBM no qual a pressão é calculada usando uma equação de estado; e utiliza um conjunto mínimo de velocidades no espaço de fase em comparação com a teoria cinética tradicional com a distribuição de equilíbrio Maxwell-Boltzmann na qual o espaço de fase é um espaço funcional completo.

O desenvolvimento de uma versão simplificada da equação cinética, por um lado, evita a resolução de equações cinéticas complexas, como a equação completa de Boltzmann, bem como seguir cada partícula como em simulações de dinâmica molecular e, por outro lado, fornece algumas vantagens da dinâmica molecular como imagens físicas claras, fácil implementação de condições de contorno e algoritmos totalmente paralelos (CHEN; DOOLEN, 1998). Segundo Aidun e Clausen (2010), a simplicidade de formulação e aplicação na resolução de problemas em comparação com a resolução das equações de Navier-Stokes (NS), bem como o alto nível de escalabilidade em sistemas de processamento paralelo contribuíram para a popularização do método. Portanto, devido ao paralelismo, à simplicidade da programação e à capacidade de incorporar as interações do modelo, o método de Lattice-Boltzmann se tornou um poderoso método computacional no estudo de vários sistemas complexos (HE; LUO, 1997).

1.2 Modelos de Reticulados

O fato do LBM ser uma derivação da classe de autômato celular LGA faz com que os seus modelos de reticulados sejam similares aos modelos do LGA. O primeiro modelo LGA, chamado HPP, foi definido por Hardy, De Pazzis e Pomeau (1976) com base em um reticulado quadrado bidimensional no qual as partículas microscópicas de massa e velocidade unitárias se movem ao longo de suas quatro direções. Nesse modelo existe apenas uma única partícula por direção em um determinado momento no tempo em um nó e, portanto, quando duas partículas microscópicas chegam a um nó por meio de direções opostas essas imediatamente deixam o nó mediante outras duas direções anteriormente desocupadas. Apesar dessas regras conservarem a massa (quantidade de partículas) e o momento, o principal problema desse modelo é o fato do comportamento gasoso modelado ser anisotrópico (WEI et al., 2004).

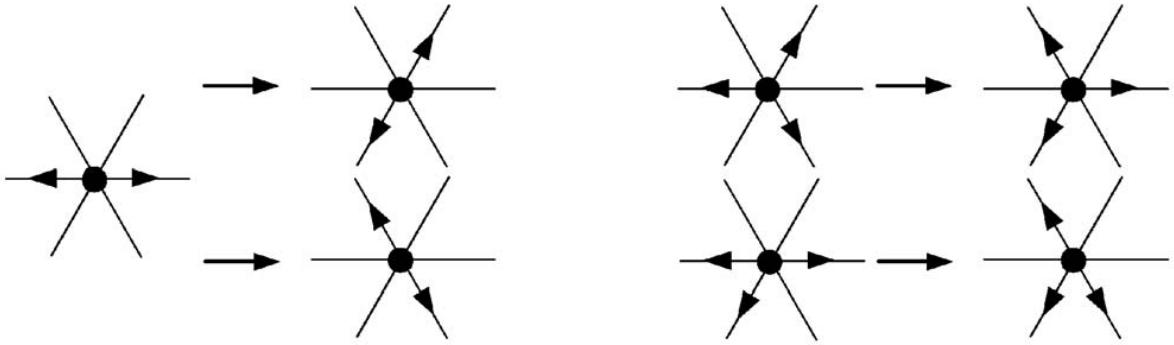
Posteriormente, Frisch, Hasslacher e Pomeau (1986) apresentaram um modelo de reticulado hexagonal bidimensional denominado FHP que, ao contrário do modelo HPP, garante a isotropia macroscópica. Neste modelo, cada célula possui seis vizinhos mais próximos e, consequentemente, seis direções de velocidade por meio das quais cada célula pode se deslocar. A atualização da grade envolve dois tipos de regras: a propagação, que consiste na movimentação das partículas microscópicas para o vizinho mais próximo ao longo de sua direção de velocidade; e a colisão, que é a parte mais importante da atualização na qual as partículas podem ser forçadas a mudar de direção, decisão essa tomada pelo operador de colisão (WEI et al., 2004). Esse operador de colisão, no entanto, deve ser definido de modo que a massa e o momento sejam preservados em qualquer decisão.

Na Figura 1 são apresentadas as diferentes regras de colisão do modelo FHP. Quando duas partículas microscópicas entram no mesmo nó com velocidades opostas ambas são desviadas 60 graus de modo que o momento após a colisão seja igual a zero e, quando múltiplos estados são possíveis, seja adotada uma seleção aleatória (WEI et al., 2004). Essas regras são projetadas para conservar o número de partículas e o momento em cada vértice (FRISCH; HASSLACHER; POMEAU, 1986).

Como neste trabalho serão realizadas apenas simulações tridimensionais, nos concentraremos somente nos modelos de reticulados 3D. Para uma grade 3D, a geometria do modelo deve ser simétrica para satisfazer o requisito isotrópico das propriedades do fluido bem como para recuperar corretamente as equações NS (WEI et al., 2004). Uma forma de facilitar a compreensão da concepção do modelo tridimensional do LBM é definir quatro sub-reticulados simétricos em uma grade 3D a partir dos vinte e seis vizinhos de uma célula individual. Esses sub-reticulados são exibidos na Figura 2 e as respectivas velocidades das partículas em cada sub-reticulado são apresentadas na Tabela 2.

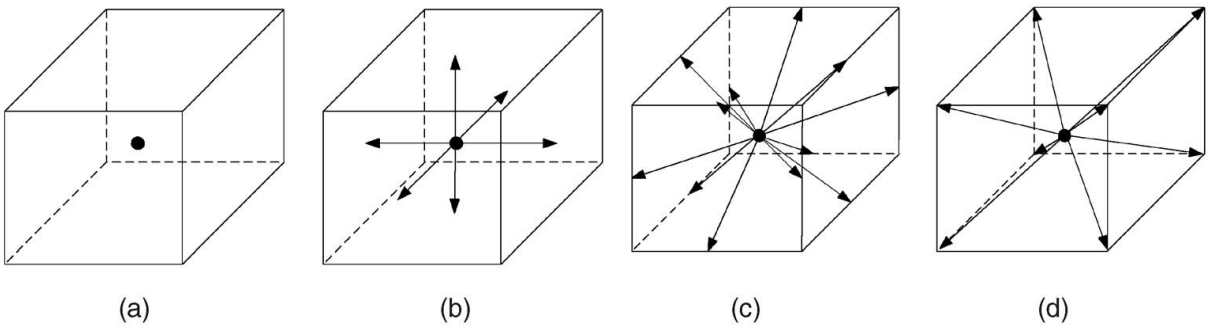
A combinação desses quatro sub-reticulados de diferentes maneiras gera modelos

Figura 1 – Representação das regras de colisão do modelo FHP em um reticulado triangular na qual setas denotam partículas microscópicas e suas direções de movimentação.



Fonte: Wei et al. (2004).

Figura 2 – Sub-reticulados com os respectivos vetores de velocidade q_0, q_1, q_2 e q_3 representados por setas.



Fonte: Wei et al. (2004).

distintos identificados por meio do número de dimensões (D) e do número de direções de deslocamento possíveis (Q). Na prática, existem três geometrias LBM tridimensionais amplamente utilizadas: D3Q15, a combinação dos sub-reticulados Figura 2a, Figura 2b e Figura 2d e possui o menor número de valores de distribuição de pacotes, é o menos isotrópico e mais propenso à instabilidade numérica; D3Q27, composto por todos os quatro sub-reticulados e, devido a essa complexidade, requer 27 cálculos de distribuição de pacotes em cada nó fluido resultando em uma alta taxa de utilização de tempo de CPU e armazenamento; e D3Q19, que consiste na combinação dos sub-reticulados Figura 2a, Figura 2b e Figura 2c representando um bom compromisso em termos de eficiência e confiabilidade computacional pois são necessárias somente 19 distribuições de pacotes em cada nó fluido (WEI et al., 2004). Dessa forma, neste trabalho utilizaremos apenas o modelo D3Q19, apresentado na Figura 3, na qual as direções de velocidade das 18 movimentações de distribuição são representadas por setas, bem como a distribuição

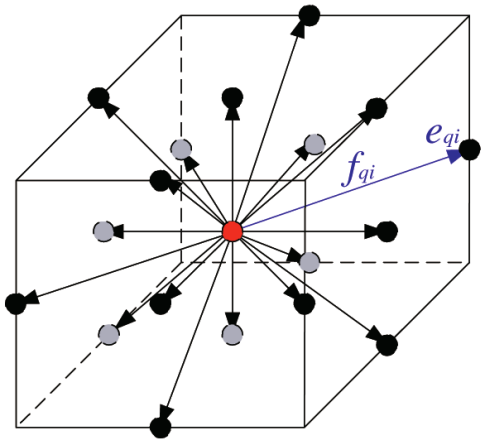
Tabela 2 – Velocidade relativa das partículas para cada uma das direções dos sub-reticulados.

Sub-reticulado	**Velocidade**
q_0	0
q_1	1
q_2	$\sqrt{2}$
q_3	$\sqrt{3}$

Fonte: Schepke e Diverio (2007).

estática representada por uma velocidade igual a zero no centro da geometria.

Figura 3 – A geometria do reticulado D3Q19.



Fonte: Wei et al. (2004).

1.3 Equação de Lattice-Boltzmann

A Equação de Lattice-Boltzmann descreve a evolução da função de distribuição de velocidade em um reticulado de forma que o comportamento dinâmico fluido macroscópico seja recuperado (STERLING; CHEN, 1996). De acordo com Chen e Doolen (1998), existem várias maneiras de obter a LBE a partir de modelos de velocidade discreta e da equação cinética de Boltzmann e, além disso, também existem várias formas de derivar as equações macroscópicas de Navier-Stokes a partir da LBE. A obtenção da Equação de Lattice-Boltzmann a partir da Equação de Boltzmann (*Boltzmann Equation* – BE) é realizada de acordo com Schepke e Diverio (2007). Por se tratar de uma extensão do trabalho apresentado por esses autores, as equações apresentadas a seguir, bem como maiores detalhes sobre as equações do LBM, podem ser encontradas no mesmo.

A função de propagação $f(x, e, t)$, negligenciando forças externas, pode ser expressa pela Equação de Boltzmann como

$$\frac{\partial f}{\partial t} + e\frac{\partial f}{\partial x} = Q(f), \tag{1.1}$$

na qual o termo de relaxação $Q(f)$ é quadrático em f. Uma simplificação apropriada da integral de colisão para o estado de quase-equilíbrio em hidrodinâmicas com baixo número de *Mach* é a aproximação via tempo de relaxação simples, conhecida como modelo BGK (Bhatnagar-Gross-Krook) (BHATNAGAR; GROSS; KROOK, 1954)

$$Q(f) = -\frac{1}{\lambda}(f - \overline{f}), \tag{1.2}$$

onde $\overline{f}$ é a função de distribuição de equilíbrio de Maxwell-Boltzmann e λ é um tempo de relaxação que controla a taxa de aproximação do equilíbrio.

Para determinar f numericamente, inicialmente ela é discretizada em relação ao campo da velocidade, utilizando um conjunto finito de vetores de velocidade e_i, onde $(i = 0, ..., d)$, gerando a equação de velocidade discreta de Boltzmann

$$\frac{\partial f_i}{\partial t} + e_i\frac{\partial f_i}{\partial x} = -\frac{1}{\lambda}(f_i - \overline{f_i}), i = 0, ..., d, \tag{1.3}$$

na qual $f_i(x, t)$ é equivalente a $f(x, e_i, t)$ e $\overline{f_i}$ é a função de distribuição de equilíbrio discreto.

Neste trabalho é utilizado o modelo de reticulado D3Q19 do LGA para as simulações do LBM. Apesar da discretização definir um pequeno número de direções de deslocamento para esse modelo, elas são suficientes para descrever um fluido próximo do estado de equilíbrio para hidrodinâmicas com baixos valores de *Mach*. Para esse modelo a função $\overline{f_i}$ apropriada para a distribuição de equilíbrio possui a forma

$$\overline{f_i} = pw_i\left[1 + \frac{3}{c^2}e_i \cdot u + \frac{9}{2c^4}(e_i \cdot u)^2 - \frac{3}{2c^2}u \cdot u\right], \tag{1.4}$$

onde $c = \Delta x/\Delta t$ e os vetores de velocidade de partícula discretos e_i. O peso do fator w_i depende somente do modelo do reticulado que é apresentado na Tabela 3 para o modelo D3Q19. A função de distribuição de equilíbrio discreta $\overline{f_i}$ é derivada da função de distribuição $\overline{f}$ de equilíbrio de Maxwell-Boltzmann de modo que os momentos de velocidade até a quarta ordem são iguais à $\overline{f}$.

Os valores macroscópicos da densidade p, momento pu e de tensor de fluxo de momento $\Pi_{\alpha\beta}$, sendo N o número de partículas e $\sum_j = \sum_i^d$, podem ser desenvolvidos como

$$p = \sum_{j=0}^{N} f_j \simeq \sum_{j=0}^{N} \overline{f_j}, \tag{1.5}$$

$$pu = \sum_{j=0}^{N} e_j f_j \simeq \sum_{j=0}^{N} e_j \overline{f_j}, \tag{1.6}$$

Tabela 3 – Peso dos fatores w_i para os vetores de velocidades discretas e_i dos modelos de reticulado 3D mais populares.

Modelo	$\|e_i\|^2 = 0$	$\|e_i\|^2 = 1$	$\|e_i\|^2 = 2$	$\|e_i\|^2 = 3$
D3Q15	$w_i = 2/9$	$w_i = 1/9$	$w_i = 0$	$1/72$
D3Q19	$w_i = 1/3$	$w_i = 1/18$	$w_i = 1/36$	0
D3Q27	$w_i = 8/27$	$w_i = 2/27$	$w_i = 1/54$	$1/216$

Fonte: Adaptado de Aidun e Clausen (2010).

$$\Pi_{\alpha\beta} = \sum_{j=0}^{N} e_{j\alpha} e_{j\beta} f_j. \tag{1.7}$$

A velocidade do som c_s desse modelo, por sua vez, é de

$$c_s = c/\sqrt{3}. \tag{1.8}$$

Já a pressão p é definida pela equação de estado de um gás ideal

$$p = \rho c_s^2. \tag{1.9}$$

A discretização do espaço e do tempo é realizada por meio de uma aproximação de diferenças finitas cuja forma explícita das equações discretizadas é dada por

$$f_i(x + e_i\Delta t, t + \Delta t) - f_i(x,t) = -\frac{1}{\tau}(f_i(x,t) - \overline{f_i}(x,t)), \tag{1.10}$$

na qual $\tau = \lambda/\Delta t$ é o tempo de relaxação adimensional.

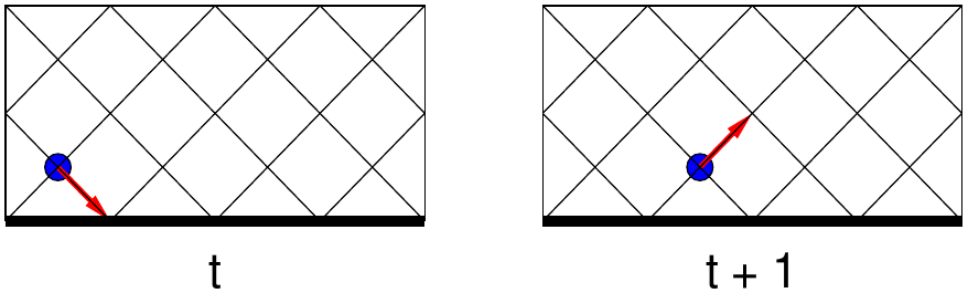
1.4 Condições de Contorno

Neste trabalho a simulação das bordas rígidas dos reticulados é realizada por meio de um mecanismo denominado *bounce-back* (barreiras reflexivas). Esse mecanismo consiste na inversão da direção da velocidade das partículas localizadas nas bordas do reticulado de modo que:

$$f_i^{out}(x,t) = f_i^{in}(x,t), \tag{1.11}$$

onde $e_i = -e_i$, ou seja, é realizada a rotação de e_i para cada direção de velocidade da borda impedindo que o fluxo ultrapasse as bordas, retornando-o para o fluido na próxima iteração do laço com momento contrário (SCHEPKE; DIVERIO, 2007). Na Figura 4 é apresentado o funcionamento desse mecanismo ante um ponto em duas iterações consecutivas.

Figura 4 – Funcionamento do mecanismo de *bounce-back*.



Fonte: Schepke e Diverio (2007).

1.5 O Algoritmo

O algoritmo utilizado neste trabalho para realizar as simulações do método de Lattice-Boltzmann foi apresentado por Schepke e Diverio (2007) e está dividido nas sete etapas apresentadas a seguir. Essas etapas estão organizadas em cinco funções da linguagem C tanto na implementação MPI apresentada por esses autores (descrita na seção 3.2 na qual as cinco funções, mais a função de sincronização, são apresentadas no Código 3.1) quanto nas implementações OpenMP apresentadas neste trabalho.

1. Determinar as condições iniciais para todos os pontos da grade, escolher adequadamente a densidade e a velocidade das células externas e selecionar a escala de tempo de relaxação τ;

2. Calcular a densidade e a velocidade das variáveis macroscópicas para cada célula utilizando a Equação 1.5 e a Equação 1.6;

3. Calcular os valores da função de equilíbrio utilizando a Equação 1.4 e, a partir disso, obter os valores da função de relaxação por meio da Equação 1.2;

4. Utilizar o valor de equilíbrio para calcular a função de distribuição para cada ponto de acordo com a Equação 1.10;

5. Propagar a distribuição das partículas para todas as células vizinhas;

6. Modificar a distribuição local dos pontos para satisfazer as condições de contorno por meio da Equação 1.11;

7. Retornar ao passo 2 enquanto o número de iterações for menor que o máximo estabelecido (`t_max`).

1.6 Áreas de Aplicação

Alguns dos desenvolvimentos mais ativos incluem o método de Lattice-Boltzmann Entrópico (*Entropic Lattice-Boltzmann* – ELB) e a aplicação do LBM ao fluxo turbulento, ao fluxo multifásico e às suspensões de partículas e fibras deformáveis (AIDUN; CLAUSEN, 2010). De acordo com Chen e Doolen (1998), a condição de limite de *bounce-back* sem deslizamento é uma característica atraente do LBM pelo fato do custo em relação ao tempo computacional ser relativamente baixo, o que o torna o método útil para a simulação de fluxos em geometrias complexas, como por exemplo fluxos que atravessam meios porosos, onde os limites das paredes são extremamente complexos e é essencial um esquema eficiente para manipular as interação parede-fluido. Alguns exemplos que demonstram a capacidade do método de Lattice-Boltzmann são Simulações Numéricas Diretas (*Direct Numerical Simulations* – DNS) de turbulência isotrópica homogênea de decomposição incompressível, Simulações de Grandes Vórtices (*Large Eddy Simulations* – LES) do fluxo passado uma esfera lisa, suspensões de partículas em fluidos, uma gotícula deslizando sobre uma superfície inclinada por gravidade e fluxos de múltiplos componentes por meio de meios porosos (LUO; KRAFCZYK; SHYY, 2010).

1.7 Considerações do Capítulo

Como pôde ser observado nesse capítulo, o método de Lattice-Boltzmann é uma evolução do autômato de Lattice Gas no qual os ruídos estatísticos foram eliminados, o que tornou as simulações mais precisas. Além disso, observou-se que o método, em relação aos métodos convencionais, destaca-se pela simplicidade uma vez que, ao invés de simular microssistemas em nível macroscópico, simula microssistemas em nível microscópico e mesoscópico permitindo a implementação facilitada de condições de contorno e o desenvolvimento de algoritmos totalmente paralelos. Essas características são apenas alguns dos motivos da popularização do método na simulação de dinâmica de fluidos computacionais.

2 Paralelismo em Aplicações e Arquiteturas

Neste capítulo abordar-se-á alguns conceitos inerentes a este trabalho no que diz respeito à arquitetura paralela utilizada à exploração do paralelismo nessa arquitetura. Inicialmente serão descritas algumas classes de paralelismo existentes de acordo com uma das mais populares taxonomias e, em seguida, serão descritas duas arquiteturas de computação paralela: de memória compartilhada e distribuída. Por fim, descrever-se-á a interface de programação paralela OpenMP utilizada neste trabalho para o desenvolvimento tanto da implementação baseada em tarefas quanto da implementação baseada no paralelismo de iterações do método, bem como alguns trabalhos relacionados a este.

2.1 Classes de Paralelismo

Existem dois tipos de paralelismo que podem ser explorados na tentativa de otimizar o desempenho de aplicações: o paralelismo da própria aplicação (*software*); e o paralelismo da arquitetura (*hardware*). Segundo Hennessy e Patterson (2011), o paralelismo em uma aplicação basicamente pode ser explorado em nível de dados e em nível de tarefas. O paralelismo em nível de dados existe quando vários elementos em um conjunto de dados em uma aplicação podem ser processados simultaneamente. Já o paralelismo em nível de tarefas ocorre quando várias funções ou operações em uma aplicação podem ser executadas de forma independente.

Esses dois tipos de paralelismo em aplicações podem ser explorados pelos computadores em nível de instrução, por meio de arquiteturas vetoriais e Unidades de Processamento Gráfico (*Graphic Processor Units* – GPUs), em nível de *thread* e em nível de requisição (HENNESSY; PATTERSON, 2011). O paralelismo em nível de instrução explora o paralelismo de dados por meio da execução de instruções paralelamente. Arquiteturas vetoriais e GPUs também exploram o paralelismo de dados, porém, aplicando a mesma instrução em um conjunto de dados de forma paralela. Já o paralelismo em nível de *thread* explora tanto o paralelismo de dados quanto de tarefas aplicando diferentes operações em dados distintos. Por fim, o paralelismo em nível de requisição explora o paralelismo de tarefas executando tarefas independentes em paralelo.

Foi baseado no paralelismo de sequências de instruções e de dados que Flynn (1972) estabeleceu uma das mais populares taxonomia para a categorização de arquiteturas de computadores existente atualmente. As sequências de instruções são relacionadas ao fluxo de instruções que partem da memória principal em direção à Unidade Central de Processamento (*Central Processing Unit* – CPU) em um ciclo completo de execução das instruções. Já as sequências de dados são relacionadas ao fluxo de dados bidirecional

existente entre o processador e a memória principal de um computador. As quatro classificações são categorizadas de acordo com a magnitude (tanto no espaço quanto no tempo multiplexado) das interações de instruções e fluxos de dados (FLYNN, 1972):

1. *Single Instruction stream and Single Data stream* (SISD) – Nessa arquitetura, apesar da execução das instruções ser sequencial, existe a possibilidade da exploração do paralelismo de instruções por meio de técnicas como a execução superescalar e especulativa (HENNESSY; PATTERSON, 2011) na qual a execução das instruções é sobreposta nos estágios de execução (*pipelining*) (HWANG; BRIGGS, 1984).

2. *Single Instruction stream and Multiple Data stream* (SIMD) – Esse tipo de arquitetura caracteriza-se pela exploração do paralelismo em nível de dados. As operações de uma mesma instrução são aplicadas sobre diferentes conjuntos de dados oriundos de distintas sequências de dados (HWANG; BRIGGS, 1984). Apesar de cada processador possuir sua própria memória de dados, existe uma única memória de instruções e um único processador de controle que busca e despacha as instruções em arquiteturas como as vetoriais, em extensões multimídia para conjuntos de instruções padrão e em GPUs (HENNESSY; PATTERSON, 2011).

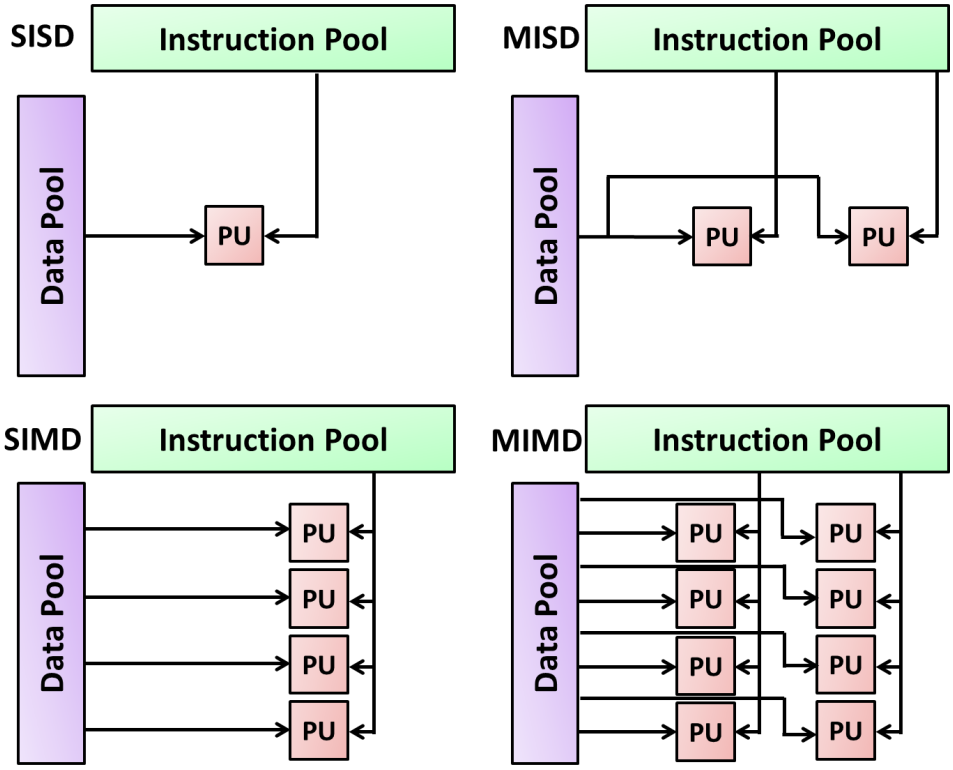
3. *Multiple Instruction stream and Single Data stream* (MISD) – Essa é uma arquitetura conceitual na qual são aplicadas múltiplas instruções sobre as mesmas sequências de dados e suas derivações (HWANG; BRIGGS, 1984). No entanto, até o momento não foram comercializados computadores desse tipo (HENNESSY; PATTERSON, 2011).

4. *Multiple Instruction stream and Multiple Data stream* (MIMD) – Arquiteturas desse tipo caracterizam-se pela exploração do paralelismo tanto em nível de dados quanto em nível de tarefas. Cada processador busca suas próprias instruções e opera sobre seus próprios dados tornando-a uma arquitetura mais flexível que a arquitetura SIMD, porém, mais custosa (HENNESSY; PATTERSON, 2011). A maioria dos sistemas multiprocessadores e multicomputadores podem ser classificados nessa categoria (HWANG; BRIGGS, 1984) e, além disso, podem ser subdivididos em arquiteturas fortemente acopladas, que exploram o paralelismo em nível de *threads*, e fracamente acopladas, que exploram o paralelismo em nível de requisições (HENNESSY; PATTERSON, 2011).

Na Figura 5 é apresentada uma representação da taxonomia de Flynn. Apesar dessa taxonomia classificar as arquiteturas em quatro categorias de acordo com os fluxos de instruções e de dados, segundo Hennessy e Patterson (2011), a maioria dos sistemas multiprocessadores e multicomputadores atualmente são híbridos das categorias SISD, SIMD e MIMD. Esses sistemas, conforme visto na descrição da categoria MIMD, podem ser constituídos de processadores forte acoplados ou fracamente acoplados, respectivamente.

De acordo com Aki (1989), Rauber e Rünger (2010), Hwang e Jotwani (2011), Pacheco (2011), sistemas multiprocessadores constituídos de processadores fortemente acoplados podem ser classificados como arquiteturas de memória compartilhada e sistemas multicomputadores constituídos de processadores fracamente acoplados, por sua vez, podem ser classificados como arquiteturas de memória distribuída. Essas são as duas principais classes de computadores paralelos que se distinguem tanto na forma como a memória é compartilhada entre os processos quanto na forma como os processos se intercomunicam (HWANG; JOTWANI, 2011). Como o foco deste trabalho é o paralelismo em nível de dados e de tarefas em arquiteturas multiprocessadores e multicomputadores, deter-nos-emos daqui por diante somente na exploração do paralelismo oferecido por essas duas arquiteturas.

Figura 5 – Taxonomia de Flynn.



Fonte: Chiesi (2014).

2.2 Arquiteturas de Memória Compartilhada

Arquiteturas de memória compartilhada consistem em um aglomerado de processadores ou núcleos de processamento (*cores*), uma memória física compartilhada entre os processadores (memória global) e uma rede de interconexões que conecta todos processadores com a memória (RAUBER; RÜNGER, 2010). Nesse tipo de sistema os processadores tipicamente são coordenados e utilizados por um único sistema operacional que compartilha

a memória por meio de um espaço de endereços compartilhado (HENNESSY; PATTERSON, 2011). Em uma arquitetura de memória compartilhada todos os processadores podem acessar cada local na memória e usualmente se comunicam implicitamente por meio do acesso à estruturas de dados compartilhadas (PACHECO, 2011). É devido ao alto grau de compartilhamento de recursos existente entre os processadores em sistemas multiprocessadores que esses também são chamados de sistemas altamente acoplados (HWANG; JOTWANI, 2011).

De acordo com Hennessy e Patterson (2011), sistemas de memória compartilhada exploram o paralelismo em nível de *thread* por meio de dois modelos de *software* distintos: a execução de um conjunto de *threads* altamente acopladas que colaboram em uma única tarefa, tipicamente denominado processamento paralelo; e a execução de múltiplos processos relativamente independentes como uma forma de exploração do paralelismo em nível de requisição tanto por uma única aplicação em execução em múltiplos processos quanto por múltiplas aplicações independentes em execução, geralmente denominado multiprogramação. As *threads* ainda podem ser utilizadas para a exploração do paralelismo de dados em arquiteturas MIMD. Apesar da sobrecarga em relação às arquiteturas SIMDs ser relativamente maior, o paralelismo de dados pode ser explorado eficientemente com uma granularidade suficientemente grande (HENNESSY; PATTERSON, 2011).

Arquiteturas multiprocessadores de memória compartilhada podem ser classificadas de acordo com o tipo de acesso à memória, que pode ser uniforme (*Uniform Memory Access* – UMA) e não uniforme (*Nonuniform Memory Access* – NUMA) (HENNESSY; PATTERSON, 2011; HWANG; JOTWANI, 2011; PACHECO, 2011). Isso se deve ao fato da memória física ser compartilhada uniformemente entre todos os processadores em arquiteturas UMA que, dessa forma, possuem uma latência uniforme de memória enquanto que em arquiteturas NUMA, onde as memórias compartilhas estão distribuídas fisicamente entre todos os processadores, o tempo de acesso depende do local em que o dado está armazenado na memória (HWANG; JOTWANI, 2011). As principais desvantagens da arquitetura NUMA são que a comunicação de dados entre processadores se torna complexa e requer um esforço elevado para que o aumento da largura de banda da memória oferecido pelas memórias distribuídas seja utilizado eficientemente (HENNESSY; PATTERSON, 2011). Segundo Pacheco (2011), por um lado sistemas UMA são mais fáceis de programar pois o programador não necessita levar em consideração os diferentes tempos de acesso aos locais na memória e, por outro lado, sistemas NUMA oferecem um acesso relativamente mais rápido à memória diretamente conectada e têm um potencial de utilização de quantidades de memória maiores que sistemas UMA (PACHECO, 2011).

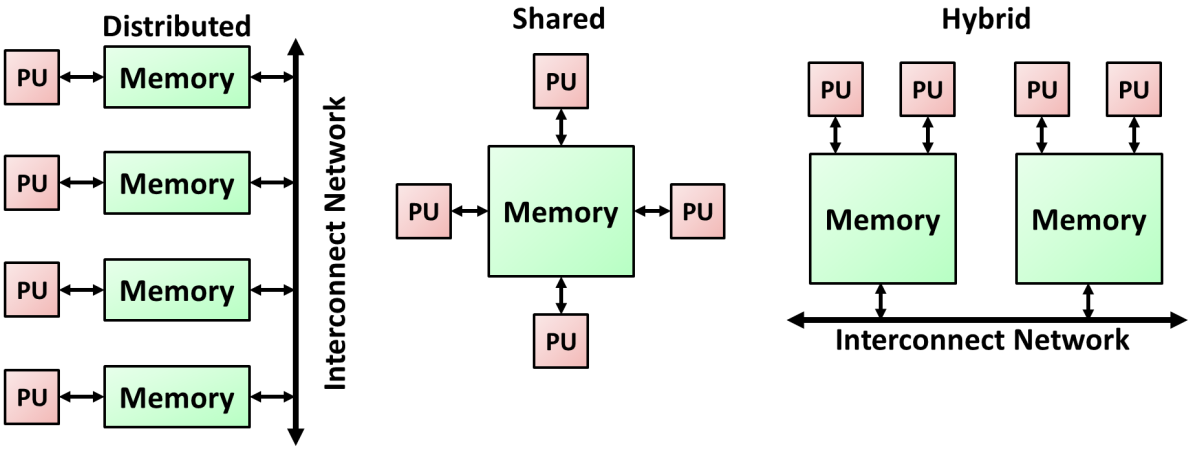
2.3 Arquiteturas de Memória Distribuída

De acordo com Hwang e Jotwani (2011), a principal diferença entre multiprocessadores e multicomputadores reside na forma como a memória é compartilhada e nos mecanismos utilizados para a comunicação entre os processadores. Uma arquitetura de memória distribuída consiste em um conjunto de elementos de processamento, denominados nós, e uma rede de interconexão que conecta e dá suporte à transferência de dados entre os nós (RAUBER; RÜNGER, 2010). Um nó é uma unidade independente composta de processadores, memória local, componentes de I/O (*input/output*) e algumas vezes de elementos periféricos (RAUBER; RÜNGER, 2010; HWANG; JOTWANI, 2011). Pelo fato da memória local de cada nó em um sistema multicomputador não ser compartilhada com os demais a comunicação entre os nós é realizada por meio de troca de mensagens (HWANG; JOTWANI, 2011) ou mediante uso de funções especiais que proveem acesso à memória de outros nós (PACHECO, 2011).

Sistemas de memória distribuídas são fortemente conectados com o modelo de programação de troca de mensagens que se baseia na comunicação entre processos sequenciais cooperantes (RAUBER; RÜNGER, 2010) e, segundo Hwang e Briggs (1984), exploram o paralelismo em nível de requisições. Uma vez que a memória local de um nó é privada e somente os processadores locais podem acessá-la diretamente, quando um processador necessita de informações armazenadas em outro nó para realizar suas operações é necessário que os processadores nesses nós se comuniquem de modo que essas informações possam ser trocadas. Essa troca de informações é realizada por meio de operações de envio e recebimento de mensagens que são executadas pelos processadores pertencentes aos nós correspondentes. Quando um processador P_B pertencente ao nó B necessita de dados armazenados na memória local do nó A o processador P_A, que pertence ao nó A, realiza uma operação de envio contendo esses dados para o processador de destino P_B que, por sua vez, realiza uma operação de recebimento especificando uma estrutura temporária (`buffer`) para armazenar os dados recebidos do processador origem P_A do qual os dados são esperados (RAUBER; RÜNGER, 2010).

Segundo Pacheco (2011), nós de sistemas de memória distribuída usualmente são sistemas de memória compartilhada com um ou mais processadores *multicore* unidos por uma rede de interconexões e, dessa forma, esses sistemas são denominados híbridos. Na Figura 6 é apresentada uma representação da organização dos sistemas de memória compartilhada, distribuída e híbrida. De acordo com Rauber e Rünger (2010), para uma coleção de computadores de memória distribuída interligados por uma rede dedicada dá-se o nome de *cluster* e para um conjunto de *clusters* conectados entre si dá-se o nome de *grid*.

Figura 6 – Representação de sistemas de memória compartilhada, distribuída e híbrida.



Fonte: Chiesi (2014).

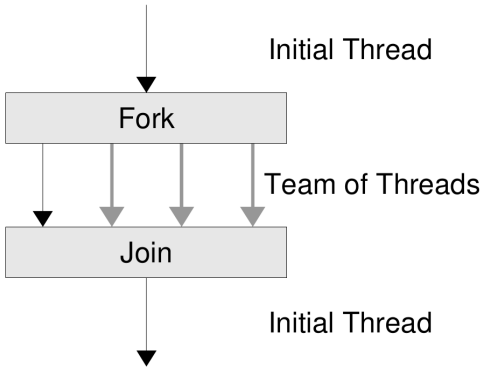
2.4 Interface de Programação Paralela OpenMP

OpenMP (*Open Multi-Processing*) é uma Interface de Programação de Aplicações (API) desenvolvida com o intuito de proporcionar uma programação paralela portável para arquiteturas de memória compartilhada (CHAPMAN; JOST; PAS, 2008). A API proporciona uma abordagem de paralelização incremental de um código existente de modo que porções de uma aplicação sejam paralelizados em passos sucessivos, ao contrário de outros paradigmas de programação que exigem a conversão completa de aplicações em um único passo (CHAPMAN; JOST; PAS, 2008). Esses trechos de código são paralelizados por meio da especificação de diretivas de compilação, que nada mais são do que comentários no código, denominadas *pragmas* na linguagem de programação C/C++ e Fortran. É a partir dessas diretivas de alto nível de programação que a implementação do OpenMP resolve os detalhes de baixo nível relacionados à criação de *threads* independentes para a execução dos trechos de código e atribuição de trabalho para as mesmas (CHAPMAN; JOST; PAS, 2008).

A API OpenMP é baseada em um modelo de programação paralela denominado *fork-join* ilustrado na Figura 7 (OpenMP Architecture Review Board, 2015). Nesse modelo, a execução de uma aplicação é iniciada de forma sequencial por uma única *thread*, denominada *thread* inicial. Quando a *thread* inicial encontra uma construção paralela OpenMP durante a execução de uma aplicação, é criado um time de *threads* (*fork*) na qual a *thread* inicial se torna a *thread* mestre/principal (*master*) e colabora com as demais *threads* do time na execução dinâmica do código envolto pela construção paralela (CHAPMAN; JOST; PAS, 2008). Ao final da construção as demais *threads* são terminadas e somente a *thread* inicial continua a execução da aplicação (*join*). A criação e terminação de *threads* ocorre

sempre que a *thread* inicial encontra uma construção paralela e o código envolto pela construção (denominado região paralela) é, então, executado paralelamente pelo time de *threads*.

Figura 7 – Modelo de programação fork-join utilizado pela API OpenMP.



Fonte: Chapman, Jost e Pas (2008).

Por meio da API OpenMP é possível criar times de *threads* para execução paralela, especificar como o trabalho deve ser compartilhado entre os membros do time, declarar tanto variáveis compartilhadas quanto privadas e sincronizar *threads* bem como permitir que as *threads* executem exclusivamente certas operações, sem interferência de outras *threads* (CHAPMAN; JOST; PAS, 2008). A criação de times de *threads* é realizada por meio da especificação de regiões paralelas a partir da inserção da diretiva `parallel` anteriormente ao código a ser executado em paralelo para marcar o início da região paralela. Quando uma *thread* encontra a diretiva `parallel`, a *thread* cria um time de *threads* para as quais é gerado um conjunto de tarefas implícitas, uma por *thread*, constituídas pelo código contido no interior da construção `parallel` e executadas somente pela *thread* para a qual cada tarefa foi atribuída inicialmente (OpenMP Architecture Review Board, 2015). Ao final da região paralela existe uma barreira de sincronização implícita que impede o prosseguimento das *threads* antes da finalização da execução de todas as *threads* do time e, após a finalização, somente a *thread* principal do time continua a execução da aplicação.

O compartilhamento do trabalho em uma região paralela, por sua vez, deve ser especificado a fim de determinar como a computação nessa região será distribuída entre as *threads* e, dessa forma, evitar a execução redundante do código na região por todas as *threads* do time. Uma das abordagens mais populares de compartilhamento de trabalho entre as *threads* de um time é a distribuição das iterações de laços de repetição do tipo `for` da linguagem de programação C/C++. Por meio da inserção da diretiva `for` anteriormente a cada laço de repetição `for` em uma região paralela as iterações dos laços são distribuídas entre as *threads* do time de modo que ambas as iterações sejam executadas paralelamente. Todas as estratégias OpenMP de compartilhamento de trabalho em laços atribuem um ou

mais conjuntos separados de iterações a cada *thread* e podem ser determinados por meio de cláusulas adicionais (CHAPMAN; JOST; PAS, 2008).

Outra abordagem de compartilhamento de trabalho entre as *threads* de um time é a criação de tarefas a partir do código contido em regiões paralelas. O suporte a tarefas foi introduzido ao OpenMP no ano de 2008 na versão 3.0 com o intuito de possibilitar a paralelização de algoritmos que possuem um fluxo de execução irregular, o que até então não era possível em muitos casos. De acordo com Pas, Stotzer e Terboven (2017), uma tarefa OpenMP pode ser definida como um bloco de código contido em uma região paralela que pode ser executada simultaneamente com outras tarefas em uma mesma região.

Quando uma *thread* encontra uma construção do tipo `task` ela gera uma nova tarefa explícita e a atribui a uma *thread* do time correspondente à região paralela na qual é gerada, atribuição essa sujeita à disponibilidade de *threads* no time (OpenMP Architecture Review Board, 2015). Após a geração, a tarefa pode ou não ser executada imediatamente por uma *thread* do time. Na API OpenMP as *threads* podem suspender a execução de uma tarefa para executar outras tarefas e, caso essa tarefa tenha sido gerada exclusivamente para determinada *thread*, somente a *thread* atribuída inicialmente pode retomar a execução da tarefa, caso contrário, qualquer tarefa no time pode retomar a execução da tarefa suspensa. Tarefas na API OpenMP por padrão são geradas exclusivamente para uma *thread* e para que qualquer *thread* no time possa retomar a execução de uma tarefa é necessária a especificação da cláusula `untied` na diretiva de criação de uma tarefa. A finalização da execução de todas as tarefas explícitas vinculadas a uma região paralela é garantida antes da *thread* principal deixar a barreira implícita ao final da região paralela, no entanto, para a finalização de um subconjunto dessas *threads* é necessária a especificação de construções de sincronização das tarefas (OpenMP Architecture Review Board, 2015).

Como as tarefas são geradas em uma região paralela cada *thread* do time gera e executa todas as tarefas. No entanto, normalmente o que se deseja é a geração de todas as tarefas de modo que essas sejam executadas arbitrariamente pelas *threads* do time. Para garantir que cada tarefa seja gerada apenas uma vez as tarefas podem ser envoltas em uma construção `single` ou `master` (PAS; STOTZER; TERBOVEN, 2017). Dessa forma, somente uma única *thread* do time gera as tarefas e as demais apenas aguardam a disponibilidade de tarefas para iniciar a execução.

Apesar da geração das tarefas seguir a ordem de especificação das diretivas no código, o escalonamento e execução das tarefas é não determinístico. Ou seja, a execução das tarefas pelas *threads* no time de uma região paralela independe da posição no código e da ordem de geração. Enquanto que em determinadas aplicações a ordem de execução das tarefas não interfere no resultado, em outras a ordem é essencial para alcançar os resultados esperados. Nesses casos é possível utilizar uma cláusula OpenMP denominada `depend` em conjunto com a diretiva `task` a fim de determinar a ordem de execução das tarefas

baseado na ordem de geração e nas dependências de dados de cada tarefa.

Por meio da cláusula `depend` é possível especificar o tipo de dependência existente entre uma tarefa e as estruturas de dados armazenadas em endereços na memória utilizadas durante a sua execução. As restrições especificadas por meio da cláusula `depend` estabelecem dependências apenas entre tarefas irmãs, ou seja, apenas entre tarefas geradas em uma mesma região paralela. O tipo de uma dependência pode ser `in`, `out` ou `inout` (OpenMP Architecture Review Board, 2015). Uma dependência do tipo `in` indica que os dados armazenados em um determinado endereço na memória são dados de entrada para a tarefa, ou seja, a tarefa somente realiza a leitura desses dados. A dependência do tipo `out`, por sua vez, indica que a tarefa somente realiza a escrita de dados na estrutura armazenada em determinado endereço na memória. Por fim, a dependência do tipo `inout` especifica que a tarefa realiza tanto a leitura das informações armazenadas em determinado endereço na memória quanto a escrita de dados, ou seja, a modificação dos dados da estrutura.

A partir desses três tipos de dependência é possível especificar listas de dependências que introduzem restrições adicionais no escalonamento de tarefas. Quando o endereço na memória de ao menos um elemento na lista de dependências de duas tarefas irmãs for o mesmo e o tipo de dependência da primeira tarefa for `in` enquanto o da segunda tarefa for `out` ou `inout`, a segunda tarefa será dependente da primeira uma vez que ambos os tipos de dependência da segunda tarefa impedem que a tarefa seja executada antes da finalização da execução da primeira tarefa. Já quando o tipo de dependência da primeira tarefa for `out` ou `inout` e da segunda tarefa for `in`, `out` ou `inout`, a segunda tarefa também será dependente da primeira e somente pode ser executada após a finalização da execução da primeira tarefa. Duas tarefas somente podem ser executadas paralelamente quando a dependência de ambas for do tipo `in` para um mesmo endereço na memória.

2.5 Trabalhos Relacionados

Vários trabalhos estudaram diferentes estratégias de paralelização do LBM em uma variedade de arquiteturas *multicore* e *manycore*. Calore et al. (2016) desenvolveram e otimizaram um método de Lattice-Boltzmann para um modelo bidimensional D2Q37 de grau de produção em duas gerações de GPU utilizando CUDA e mostraram que o código funcionou eficientemente em até 32 GPUs. Valero-Lara e Jansson (2016) estudaram dois métodos distintos, Multi-Domínio e Irregular, para o aprimoramento de malha em simulações do LBM para um modelo bidimensional D2Q9 desenvolvendo duas implementações de cada metodologia: uma implementação homogênea GPU utilizando CUDA; e outra implementação heterogênea CPU+GPU utilizando CUDA e a API OpenMP `parallel for` para paralelizar as iterações de laços de repetição. Seus resultados mostraram que o benefício de utilizar uma implementação heterogênea de CPU+GPU é maior no Multi-Domain do

que na abordagem Irregular. Tang et al. (2016) propuseram um modelo paralelo híbrido MPI+OpenMP `parallel for` do LBM para um modelo bidimensional D2Q9 baseado em arquiteturas heterogêneas CPU+MIC. Para realizar a divisão de dados, os autores utilizaram um método de particionamento bidimensional e, para melhorar o desempenho do algoritmo, utilizaram vários métodos de otimização. Seus resultados mostraram que os modelos paralelos híbridos obtiveram melhor desempenho e escalabilidade em uma arquitetura de CPU+MIC de vários nós.

Alguns estudos também exploraram o paralelismo em arquiteturas de memória compartilhada utilizando a API OpenMP. Zhou et al. (2017) desenvolveram uma implementação paralela do Lattice-Boltzmann para um modelo bidimensional D2Q9 utilizando a cláusula OpenMP `parallel for` e obtiveram um *speedup* máximo de 2,95 com 4 *threads* em uma CPU *multicore*. Başağaoğlu et al. (2017a) compararam o desempenho computacional de uma implementação em linguagem de programação puramente funcional SequenceL com o desempenho de outra implementação OpenMP de laços `do-loop` paralelos. Nessas implementações os autores paralelizaram um método de Lattice-Boltzmann utilizando em simulações em mesoescala de um grande número de partículas em um dispositivo microfluídico para um modelo bidimensional D2Q9. Nagar et al. (2015) criaram uma biblioteca paralela LBM-IB para um modelo tridimensional D3Q19 em arquiteturas de memória compartilhada e compararam a paralelização de uma implementação OpenMP baseada no paralelismo de iterações de laços de repetição (`parallel for`) com uma implementação Pthread baseada em cubo.

Ye et al. (2015), Başağaoğlu et al. (2017b) e Meadows e Ishikawa (2017) apresentaram implementações baseadas em tarefas do método de Lattice-Boltzmann. Ye et al. (2015) paralelizaram a simulação do fluxo do método Entrópico de Lattice-Boltzmann (ELBM) para o modelo tridimensional D3Q19 em um ambiente de computação híbrido CPU-GPU utilizando OpenMP e CUDA, projetando um mecanismo de paralelismo em nível de tarefa utilizando filas de tarefas. Başağaoğlu et al. (2017b) apresentaram uma implementação de *pipeline* em nível de tarefas do modelo bidimensional D2Q9 do LBM utilizando a linguagem de programação *Threading Building Blocks* da Intel, implementação essa que alcançou o melhor desempenho em relação à implementação paralela OpenMP utilizando `parallel for`. Meadows e Ishikawa (2017) desenvolveram duas implementações baseadas em tarefas a partir de uma implementação OpenMP+MPI otimizada do método de Lattice Quantum Chromodynamics (QCD) para um modelo tridimensional. Uma das implementações foi desenvolvida com tarefas OpenMP e a outra foi desenvolvida com tarefas codificadas à mão. Os resultados mostraram que a superposição da comunicação MPI e computação em ambos os métodos foram melhores e, além disso, expuseram alguns problemas de desempenho na versão baseada em tarefas OpenMP. Ambas as implementações baseadas em tarefas superam a implementação original quando a escala era forte.

Além disso, outros estudos obtiveram significativos ganhos de desempenho ao paralelizar aplicações utilizando tarefas com dependências. Martínez et al. (2015) paralelizaram uma aplicação para a simulação da propagação de ondas sísmicas (Ondes3D) utilizando tarefas para arquiteturas heterogêneas CPU+GPU utilizando o *framework* StarPU. Agullo et al. (2017) apresentaram a paralelização baseada em tarefas do método multipolar rápido (*Fast Multipolar Method* – FMM) e mostraram que as tarefas e dependências introduzidas nas versões mais recentes da API OpenMP permitem melhorar significativamente o desempenho a níveis antes somente alcançados com APIs especializadas. Gajinov et al. (2014) apresentam um *benchmark* para fluxo de dados híbrido e arquiteturas de memória compartilhada mostrando que modelos de fluxo de dados baseados em grafos de dependência de dados proporcionam maior flexibilidade que grafos de dependência de tarefas, que por sua vez proveem melhor programabilidade e desempenho.

Como é possível observar, a maioria desses estudos utilizaram modelos bidimensionais ao invés de modelos tridimensionais. A utilização de modelos bidimensionais se deve pelo fato da complexidade na paralelização desses modelos ser menor do que a complexidade presente na paralelização de modelos tridimensionais. Uma vez que diferentes modelos no LBM possuem diferentes quantidade de direções de propagação das forças, a medida que o método de Lattice-Boltzmann é paralelizado, essas direções de propagação normalmente se tornam dependências que podem influenciar negativamente no desempenho das implementações.

Além disso, quando esses estudos utilizaram a API OpenMP, a abordagem dominante na paralelização do método de Lattice-Boltzmann foi a paralelização das iterações dos laços de repetição por meio da cláusula `parallel for`. Essa abordagem normalmente é utilizada por se tratar de uma forma relativamente fácil de paralelizar um algoritmo e por oferecer ganhos significativos de desempenho. No entanto, uma abordagem introduzida recentemente na API OpenMP e que se mostra promissora na paralelização de diferentes problemas é o paralelismo de tarefas com dependências. A partir dessa abordam é possível criar tarefas paralelas e determinar quais são suas dependências de dados de modo que o escalonador possa determinar quais e em que ordem as tarefas podem ser executadas paralelamente. Uma vez que modelos mais complexos do LBM introduzem um número elevado de dependências em implementações paralelas, a utilização de tarefas com dependências pode oferecer significativos ganhos de desempenho permitindo que não somente laços de repetição sejam paralelizados e, sim, diferentes trechos de código, os quais são executados paralelamente de acordo com suas dependências de dados.

2.6 Considerações do Capítulo

Nesse capítulo apresentou-se inicialmente a taxonomia de Flynn e as principais características de arquiteturas de memória compartilhada e memória distribuída. Apresentou-se ainda a interface de programação OpenMP, detalhando as principais características que a tornam uma das mais populares interfaces de programação paralela em arquiteturas de memória compartilhada. Além disso, descreveu-se alguns trabalhos que paralelizaram o método em diferentes arquiteturas paralelas, destacando o modelo de reticulado e a abordagem de paralelização utilizadas, e alguns trabalhos que paralelizaram outros problemas utilizando tarefas com dependências, uma abordagem introduzida recentemente na API OpenMP, e que obtiveram ganhos significativos de desempenho. A partir dessas descrições foi possível observar que a maioria desses estudos utilizou modelos menos complexos para a paralelização do método de Lattice-Boltzmann e, quando da utilização da API OpenMP, a abordagem de paralelização apenas das iterações dos laços de repetição presentes nas implementações, sendo que apenas três estudos utilizam tarefas para paralelizar o método.

3 Implementações

Neste capítulo apresentar-se-á detalhadamente o particionamento de domínio adotado, as implementações baseada em tarefas e baseada no paralelismo de iterações de laços de repetição propostas por este trabalho, bem como a implementação MPI na qual ambas as implementações são baseadas. Inicialmente descrever-se-á a forma como o domínio é particionado bem como a distribuição dos subdomínios vizinhos ortogonais e diagonais de um subdomínio. Em seguida apresentar-se-á a implementação MPI, a implementação baseada em tarefas inicialmente proposta neste trabalho e uma implementação otimizada da implementação baseada em tarefas inicial que utiliza estruturas temporárias na etapa de cópia das bordas dos subdomínios vizinhos como uma forma de aumentar o paralelismo na execução das tarefas, especialmente na etapa de cópia das bordas. Pelo fato do particionamento de domínio no método de Lattice-Boltzmann implicar em uma série de dependências entre subdomínios vizinhos que, por sua vez, impõem restrições no processamento paralelo de cada subdomínio, em ambas as implementações baseadas em tarefas apresentar-se-á um cenário hipotético de execução para detalhar o impacto dessas restrições na execução paralela das tarefas. Por fim, apresentar-se-á a implementação baseada no paralelismo de iterações de laços de repetição.

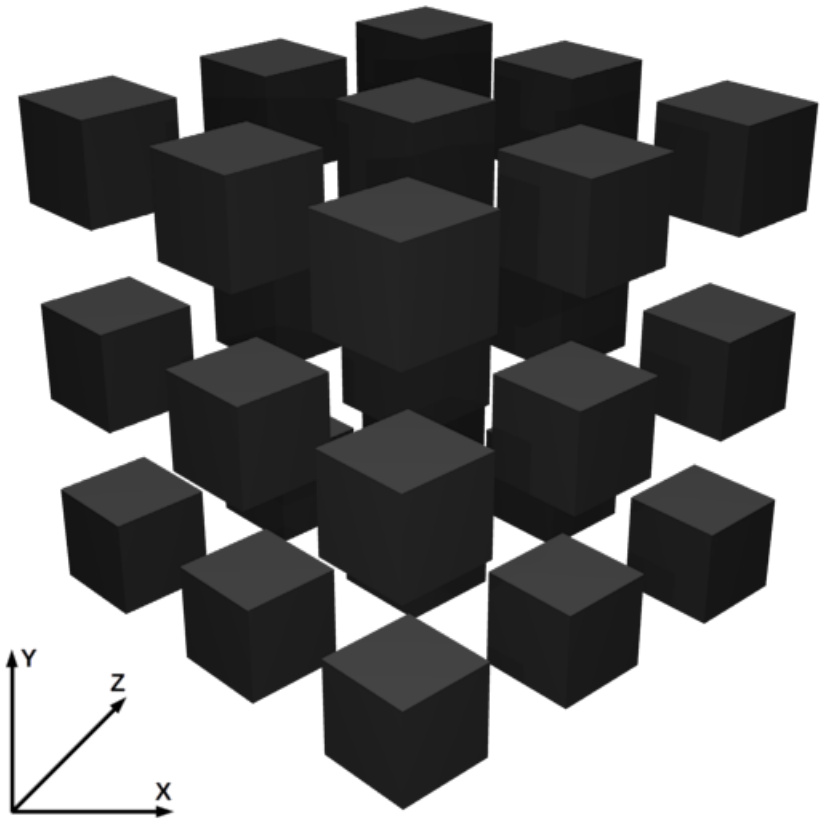
3.1 Particionamento de Domínio

O modelo D3Q19 do LBM caracteriza-se pela representação e simulação de fluidos tridimensionalmente no espaço. Nesse modelo, partículas locomovem-se arbitrariamente no espaço em dezenove direções. Para representar essas partículas do fluxo é utilizada uma estrutura de dados chamada matriz multidimensional. Uma matriz pode ser definida como uma sequência de elementos de um mesmo tipo dispostos de forma contínua na memória e que podem ser acessados individualmente por meio de índices. Dessa forma, uma matriz multidimensional é constituída por uma matriz na qual seus elementos também são matrizes contendo elementos ou ainda outras matrizes. Cada elemento de uma matriz multidimensional pode ser acessado por meio de um índice único.

Para possibilitar um alto nível de paralelismo na computação do método a partir dos dados contidos na matriz é utilizada uma técnica conhecida como particionamento de domínio. No particionamento de domínio, dados necessários para o processamento de determinado algoritmo são divididos em partes menores, também conhecidos como subdomínios, a fim de aumentar o paralelismo de dados. Desse modo, dados contidos na matriz utilizada pelo método são particionados em subdomínios que contem uma parcela do domínio e que podem, consequentemente, ser processados simultaneamente.

Se imaginarmos a matriz de dados utilizada pelo LBM como um cubo tridimensional, o particionamento desse cubo em n cubos menores em cada uma das dimensões resulta em um agrupamento similar ao ilustrado na Figura 8. Nela é possível observar que cada dimensão (x, y e z) foi particionada em três cubos menores resultando em um particionamento de domínio em 27 cubos. Cada cubo corresponde a uma fração do cubo original, ou seja, do domínio original, que no caso do LBM é a matriz de densidade das partículas que compõe o fluxo em questão. Em outras palavras, cada cubo corresponde a uma submatriz idêntica à matriz original diferindo apenas no tamanho da estrutura. Dessa forma, como cada cubo ilustrado na Figura 8 é único e independente dos demais os cubos podem ser processados simultaneamente por processos arbitrários.

Figura 8 – Representação do particionamento do domínio.



Fonte: O autor.

O particionamento do domínio no LBM pode ser realizado em uma, duas ou três dimensões. Cada dimensão pode ser particionada em um número arbitrário de partes menores desde que esse número seja múltiplo do tamanho da dimensão em questão. Por exemplo, em um domínio no qual cada dimensão tenha um tamanho igual a 128, isto é, tanto a dimensão x quanto a dimensão y e z possuem um tamanho igual a 128 (128x128x128), pode ser particionado em 2 partes na dimensão x, 8 partes na dimensão y e 4 partes na dimensão z (2x8x4) resultando em um particionamento de domínio em

um total de 64 partes. Nesse exemplo cada parte corresponde a uma submatriz idêntica à matriz original com exceção do tamanho de cada dimensão, que é igual a 64 na dimensão x, 16 na dimensão y e 32 na dimensão z (64x16x32).

No entanto, como se trata de uma matriz de dados sobre um fluxo no qual partículas se movem por todo o fluxo (portanto, pelo domínio por completo), subdomínios que são resultados do particionamento do domínio no LBM não são completamente independentes. Uma partícula em um subdomínio pode mover-se para outro subdomínio por meio das dezenove direções do modelo introduzindo dependências entre subdomínios vizinhos. Desse modo, cada subdomínio possui uma cópia da borda de seu subdomínio vizinho de forma que possa ser processado levando em consideração a troca de partículas entre subdomínios vizinhos.

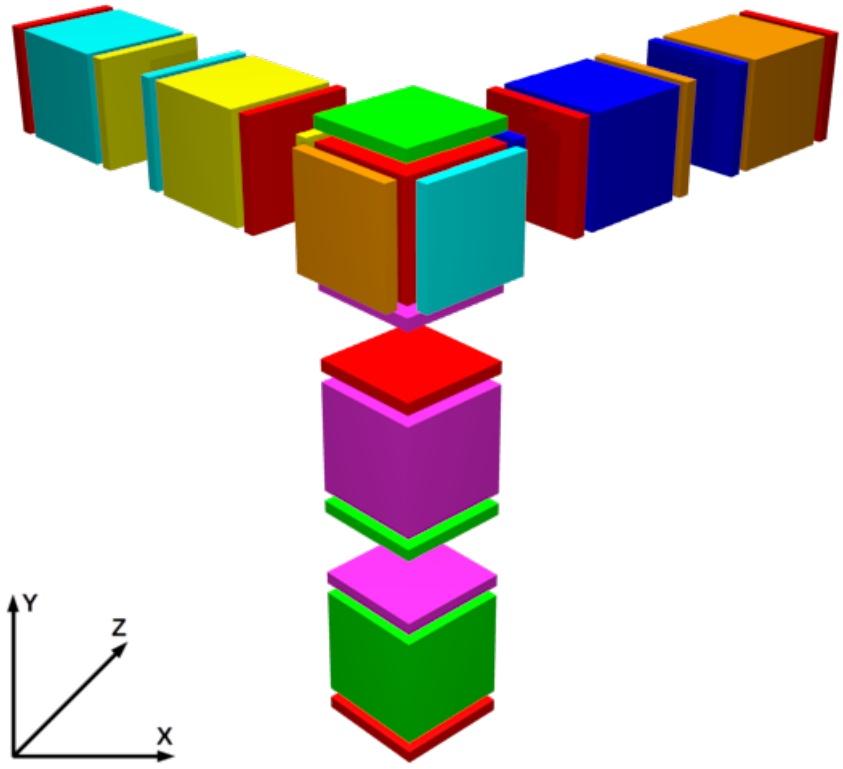
Como o modelo D3Q19 é caracterizado pela existência de dezoito direções de propagação das partículas do fluido (além da posição estática), um subdomínio pode possuir até dezoito subdomínios vizinhos de acordo com o particionamento utilizado. Dos dezoito vizinhos, seis são vizinhos ortogonais (dois por dimensão) e doze são vizinhos diagonais (quatro por dimensão). Para que um subdomínio possa ser processado corretamente pelo modelo é necessária a cópia da borda dos dezoito vizinhos.

3.1.1 Vizinhos Ortogonais

Utilizando a representação do particionamento de domínio em cubos vista anteriormente, a cópia dos dados que compõem as bordas dos vizinhos ortogonais de um cubo pode ser representada como uma camada adicional para cada borda de cada vizinho do cubo. A cópia dessas bordas só é necessária quando existem cubos vizinhos em uma determinada dimensão. Um cubo possui vizinhos em uma determinada dimensão quando nessa há um particionamento em n cubos, com $n > 1$. Nesse caso, para cada dimensão na qual há um particionamento existem dois cubos vizinhos, um cubo vizinho predecessor e outro posterior em uma ordem crescente de posições na dimensão. No entanto, em casos onde n é igual a 1 em determinada dimensão não existem cubos vizinhos e, portanto, não há a necessidade de cópia das bordas entre cubos nessa dimensão.

A representação da cópia das bordas dos vizinhos ortogonais de cada cubo em um particionamento de domínio de três cubos por dimensão é apresentada na Figura 9. Nela, para cada cubo da Figura 8 foram acrescentadas camadas adicionais relativas às bordas de cada cubo vizinho ortogonal. Para facilitar a visualização dos cubos vizinhos bem como das respectivas cópias das bordas apenas três cubos em cada dimensão, ambos posicionados nas extremidades, foram coloridos e apenas as cópias das bordas dos vizinhos ortogonais de um único cubo foram coloridas. O restante dos cubos e das cópias das bordas estão descoloridos e transparentes pois seguem o mesmo princípio dos cubos coloridos.

Figura 9 – Divisão e vizinhança ortogonal.



Fonte: O autor.

Na Figura 9 é possível observar que apenas o cubo de cor vermelha possui todas as seis cópias das bordas de seus vizinhos ortogonais coloridas de acordo com as cores dos respectivos cubos. Para cada cubo precedente e subsequente ao cubo vermelho nas três dimensões há uma camada adicional representando a cópia da borda desses cubos. Na dimensão x há uma camada de cor amarela representando a borda do vizinho predecessor ao cubo vermelho, na dimensão y há uma camada de cor lilás para a borda do seu vizinho predecessor e na dimensão z há uma camada de cor azul para a borda de seu vizinho posterior. Como não existem cubos posteriores nas dimensões x e y e predecessor na dimensão z para o cubo de cor vermelha é utilizada uma definição de vizinhança cíclica. Isto é, quando $n > 1$ em determinada dimensão, cubos nas extremidades tem como seus vizinhos cubos localizados na extremidade inversa da mesma dimensão. Dessa forma, os cubos de cor ciano e verde são os vizinhos posteriores nas dimensões x e y, respectivamente, e o cubo de cor laranja é o cubo predecessor na dimensão z do cubo vermelho. Essa abordagem é utilizada para evitar perdas de densidade durante o processamento.

Uma camada relativa a cópia de uma borda de um vizinho ortogonal consiste na cópia da face do cubo vizinho em uma determinada dimensão. Assim, o tamanho das camadas de cada cubo depende diretamente do tamanho da respectiva face. Apesar do tamanho de todos os cubos ser o mesmo na Figura 9, o tamanho de cada camada relativa

à cópia das bordas dos vizinhos é diretamente dependente do tamanho das faces dos cubos vizinhos. Além disso, como apenas a face dos cubos vizinhos é copiada uma das dimensões de cada camada sempre terá um tamanho igual a 1. Ou seja, o tamanho da dimensão x das camadas que representam as cópias das bordas dos vizinhos ortogonais na dimensão x é igual a 1, bem como o tamanho da dimensão y e z das camadas dos vizinhos nas dimensões y e z, respectivamente.

Para implementar as representações das cópias das bordas dos vizinhos ortogonais realizada por meio de camadas adicionais nos cubos das ilustrações apresentadas na Figura 9 podem ser utilizadas matrizes bidimensionais. Ou seja, utilizando o exemplo anterior, no qual cada cubo possui um tamanho igual a 64x16x32, as camadas correspondentes aos vizinhos ortogonais na dimensão x possuem um tamanho igual ao tamanho y do cubo multiplicado pelo tamanho z (1x16x32), na dimensão y possuem um tamanho igual a x multiplicado por z (64x1x32) e na dimensão z possuem um tamanho igual a x multiplicado por y (64x16x1). Como uma das dimensões dessas camadas é igual a 1 as matrizes bidimensionais podem ser utilizadas para armazenar os dados da borda dos vizinhos ortogonais.

No entanto, ao invés de utilizar matrizes bidimensionais independentes para realizar a cópia das bordas dos vizinhos ortogonais essas são adicionadas diretamente às matrizes nas quais os dados das partes do domínio estão armazenados. Dessa forma, a alocação de memória para as matrizes tridimensionais que armazenarão os dados de cada parte do domínio é realizada com duas unidades adicionais em cada dimensão da matriz para armazenar as duas bordas dos vizinhos em cada dimensão.

3.1.2 Vizinhos Diagonais

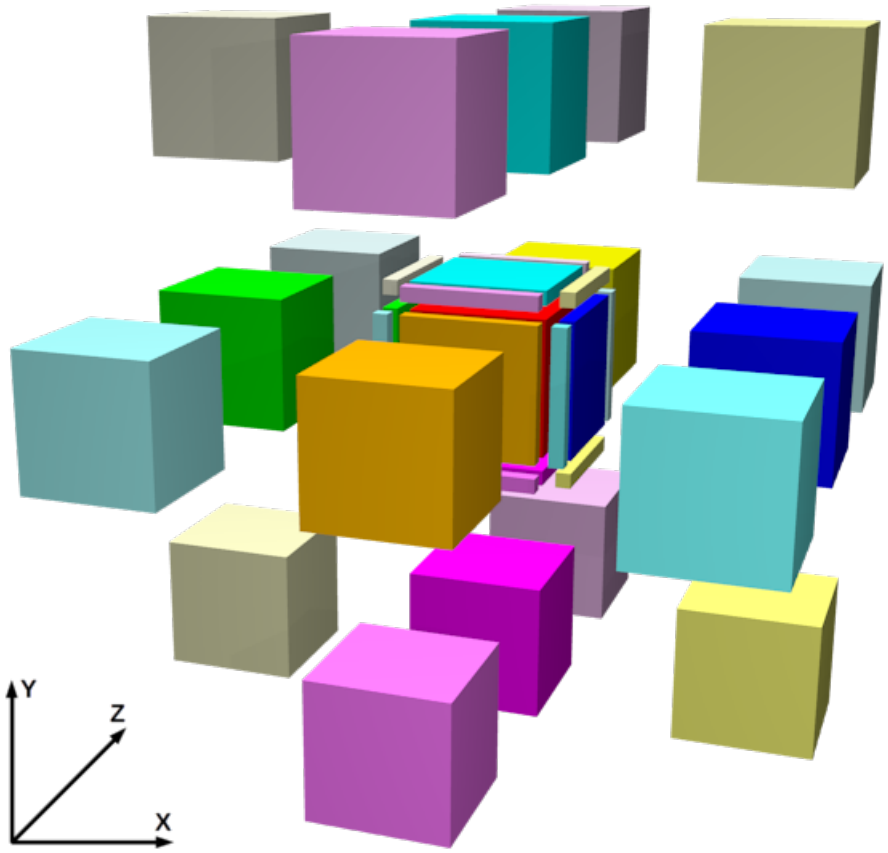
Além dos seis vizinhos ortogonais, um subdomínio ainda pode ter até doze subdomínios vizinhos diagonais, quatro por dimensão. Se apenas uma dimensão de um domínio for particionada em n subdomínios, com $n > 1$, cada subdomínio terá apenas oito subdomínios vizinhos diagonais entre os quais haverá a necessidade de realizar cópias das bordas. Nesse caso, dependendo do valor de n e da posição em que o subdomínio se encontra alguns de seus oito subdomínios vizinhos diagonais podem ser repetidos. Ou seja, um mesmo subdomínio pode ser o vizinho diagonal para mais de uma direção de propagação das forças de um subdomínio devido à definição cíclica da vizinhança. No entanto, se particionadas somente duas ou três dimensões de um domínio em um número arbitrário de subdomínios por dimensão cada subdomínio terá doze subdomínios vizinhos diagonais e em ambos os casos um mesmo subdomínio pode ser um vizinho para mais de uma direção de propagação.

Se utilizarmos novamente a representação do particionamento de domínio em cubos, a cópia das bordas dos vizinhos diagonais de um cubo podem ser representadas como camadas adicionais para cada borda, da mesma forma que a representação das

bordas ortogonais. No entanto, a diferença entre ambas as camadas está no tamanho, uma camada ortogonal possui um tamanho igual ao tamanho da face do cubo vizinho ortogonal enquanto uma camada diagonal possui um tamanho igual ao tamanho da aresta do cubo vizinho diagonal. Desse modo, o tamanho de uma camada diagonal depende diretamente do tamanho da aresta do respectivo cubo vizinho diagonal.

A representação da cópia das bordas dos vizinhos diagonais de um cubo em um particionamento de domínio qualquer é apresentada na Figura 10. Nela são exibidas as posições de cada cubo vizinho ortogonal e diagonal bem como as representações por meio de camadas adicionais das respectivas cópias das bordas ortogonais e diagonais para um cubo. O cubo, seus cubos vizinhos ortogonais e as respectivas camadas adicionais representando as bordas ortogonais e diagonais são exibidos com cores opacas e os demais cubos diagonais são exibidos com cores transparentes a fim de facilitar a visualização da distribuição das posições de cada cubo vizinho em um particionamento nas três dimensões.

Figura 10 – Divisão e vizinhança diagonal.



Fonte: O autor.

O cubo vermelho localizado no centro da Figura 10 está rodeado por camadas adicionais representando as dezoito cópias das bordas de seus vizinhos ortogonais e diagonais. Para facilitar o entendimento da distribuição das posições de cada vizinho de

um cubo em um particionamento de domínio nas três dimensões cada cubo vizinho do cubo vermelho possui uma coloração única. Além disso, cada camada possui a mesma coloração do cubo vizinho correspondente.

Os cubos vizinhos ortogonais predecessores e posteriores do cubo vermelho na dimensão x possuem a coloração verde e azul, na dimensão y a coloração lilás e ciano e na dimensão z a coloração laranja e amarela, respectivamente. Como é possível observar, esses cubos estão distribuídos perpendicularmente em cada dimensão do domínio. Já os cubos vizinhos diagonais estão distribuídos transversalmente entre duas dimensões do domínio. Ou seja, o cubo predecessor do cubo laranja e do cubo lilás na dimensão y e na dimensão z, respectivamente, é um cubo vizinho diagonal do cubo vermelho, por exemplo. Ainda nas dimensões y e z, o cubo posterior ao laranja e predecessor ao ciano, bem como os cubos posteriores ao ciano e ao amarelo e predecessor ao amarelo e posterior ao lilás também são vizinhos diagonais do cubo vermelho.

Essa sequência de distribuições de vizinhos para o cubo vermelho se repete nas dimensões x e z e nas dimensões x e y. Nas dimensões x e z o cubo predecessor aos cubos laranja e verde, predecessor ao azul e posterior ao laranja, posteriores aos cubos azul e amarelo e predecessor ao cubo amarelo e posterior ao cubo verde são vizinhos diagonais do cubo vermelho. Já nas dimensões x e y o cubo predecessor aos cubos lilás e verde, predecessor ao azul e posterior ao lilás, posteriores aos cubos azul e ciano e predecessor ao cubo ciano e posterior ao cubo verde também são vizinhos diagonais do cubo vermelho.

Apesar de na Figura 10 o cubo vermelho não estar localizado na extremidade de nenhuma dimensão, em casos onde um cubo está em uma ou mais extremidades do particionamento do domínio as mesmas cópias das bordas serão realizadas, no entanto, com cubos localizados nas extremidades inversas das extremidades nas quais está localizado. As representações das cópias das bordas representadas na figura e implementadas no LBM ilustram as dependências existentes entre cada cubo e seus vizinhos no modelo D3Q19. Dessa forma, devido à técnica cíclica de definição de vizinhança utilizada todos os cubos terão essas dependências independente de sua localização no particionamento do domínio.

3.2 MPI

A implementação MPI do método de Lattice-Boltzmann apresentada por Schepke e Diverio (2007) foi desenvolvida por meio da linguagem de programação C padrão ANSI. Inicialmente é criada uma topologia cartesiana de três dimensões (3D) na qual cada dimensão possui determinado número de processos MPI de acordo com o especificado por meio de argumentos no momento da execução da implementação. Essa topologia é criada por meio da função `MPI_Cart_create()` que gera um objeto comunicador contendo os identificadores e as coordenadas cartesianas de cada processo. A partir desse objeto, cada

processo MPI é associado à topologia cartesiana por meio da função `MPI_Comm_rank()`. Essa função retorna um identificador para cada processo que, por sua vez, é utilizado para determinar as coordenadas cartesianas de cada processo na topologia por meio da função `MPI_Cart_coords()`.

Após a criação da topologia cartesiana é realizado o particionamento do domínio com base nas coordenadas cartesianas de cada processo nessa topologia, conforme visto na seção 3.1. O particionamento consiste na delimitação da fração em cada uma das dimensões do reticulado correspondente a cada um dos processos MPI. Os limites iniciais e finais do subdomínio de cada processo em cada uma das dimensões do reticulado são determinados a partir das coordenadas cartesianas das respectivas dimensões na topologia e do tamanho das respectivas dimensões do subdomínio, tamanho esse igual à divisão do tamanho de cada dimensão do reticulado pelo número de processos nas respectivas dimensões. Quando o tamanho de uma dimensão é igual em todos os subdomínios o limite inicial de cada subdomínio nessa dimensão é obtido a partir da multiplicação da coordenada topológica nessa dimensão pelo tamanho da dimensão do subdomínio. Além disso, o limite final de cada subdomínio nessa dimensão é obtido a partir do incremento de um na coordenada topológica nessa dimensão e posterior multiplicação pelo tamanho da dimensão do subdomínio.

No entanto, em casos onde o tamanho de determinada dimensão não é igual em todos os subdomínios os limites iniciais e finais de cada subdomínio no reticulado são calculados de modo que o restante da divisão do tamanho dessa dimensão pela quantidade de processo na respectiva dimensão do reticulado seja distribuído entre os subdomínios. Esse balanceamento somente ocorre quando a coordenada topológica de um subdomínio nessa dimensão é maior que o resultado da subtração do resto da divisão do tamanho do reticulado pelo total de processos nessa dimensão do total de processos nessa dimensão subtraído de um. Nesse caso o tamanho da respectiva dimensão do subdomínio é incrementado em um e o limite final de cada subdomínio nessa dimensão é calculado subtraindo do tamanho do reticulado o resultado da multiplicação do tamanho do subdomínio pelo resultado da subtração da coordenada topológica do total de processos após a subtração de um desse total de processos. Por fim, o limite inicial de cada subdomínio nessa dimensão é calculado simplesmente subtraindo do limite final o tamanho da respectiva dimensão do subdomínio.

Ao final do particionamento as informações relacionadas ao subdomínio de cada processo são utilizadas na alocação de memória para o armazenamento tanto dos obstáculos[1] presentes no subdomínio (ou seja, das posições no subdomínio em que se encontram esses obstáculos) quanto das informações produzidas no decorrer da execução do método. As informações sobre a presença de obstáculos no subdomínio são obtidas a partir da leitura

[1] Barreiras sólidas que impedem parcialmente o escoamento regular do fluido em um reticulado, exigindo que o escoamento ocorra somente por meio das áreas não obstruídas pelas barreiras.

de um arquivo texto que contém informações sobre o reticulado que será simulado e, então, armazenadas em uma matriz tridimensional denominada `obst`. Essa matriz armazena valores booleanos que indicam a presença ou não de obstáculos em cada uma das posições do subdomínio e, portanto, possui um tamanho igual ao do subdomínio de cada processo. Já as informações do subdomínio, bem como as informações das bordas dos subdomínios vizinhos são armazenadas em duas matrizes tetradimensionais idênticas denominadas `node` e `temp`. Ambas as matrizes possuem três de suas dimensões com tamanho igual ao tamanho das respectivas dimensões do subdomínio de cada processo, incrementado em dois para o armazenamento das bordas dos subdomínios vizinhos, e a quarta dimensão com tamanho igual ao número de direções de propagação presente no modelo D3Q19. A matriz `node` é a matriz principal na qual são armazenados os resultados da execução do método e a matriz `temp` é a matriz auxiliar na qual são armazenadas informações temporárias durante a execução do método, como por exemplo a cópia das bordas dos subdomínios vizinhos.

Finalizadas as etapas de criação da topologia cartesiana dos processos MPI, de particionamento do domínio em subdomínios de acordo com o total de processos e de alocação da memória necessária para o armazenamento de informações nas três matrizes, a próxima etapa é a de execução do método de Lattice-Boltzmann. A execução do método consiste na aplicação de um conjunto de operações descritas em seis funções arbitrárias por cada processo sobre as informações contidas em seu subdomínio. Essas funções aplicam suas operações somente nas matrizes principais ou auxiliares de cada processo e, além disso, algumas funções ainda utilizam uma ou ambas as matrizes principais e auxiliares para obter informações necessárias para a aplicação das operações. Por outro lado, em relação à utilização das matrizes de obstáculos de cada processo, somente algumas funções utilizam as matrizes de obstáculos para identificar a presença ou não de obstáculos nos subdomínios.

No Código 3.1 é apresentada uma visão geral das seis funções existentes na implementação, bem como a ordem em que as mesmas são executadas em cada subdomínio pelos respectivos processos. Pelo fato de haver um processo MPI para cada subdomínio a execução dessas funções é realizada de forma paralela em todos os subdomínios, com exceção da função de sincronização (`synchronize`). Essa função é responsável pela sincronização das cópias das bordas dos subdomínios de cada processo vizinho e durante essa sincronização um processo permanece bloqueado até que o envio das bordas de seu subdomínio aos seus processos vizinhos e o recebimento das bordas dos respectivos processos vizinhos seja finalizado. Finalizados o envio e recebimento das bordas vizinhas de um subdomínio o respectivo processo MPI é liberado e a execução das funções no processo continuam até que `time` seja igual à `t_max`.

Código 3.1 – Visão geral da implementação MPI.

```
1 initialize();
2 for (time = 0; time < t_max; time++) {
3   redistribute();
4   propagate();
5   synchronize();
6   bounceback();
7   relaxation();
8 }
```

3.3 OpenMP Tarefas

Conforme visto na seção 3.2, existem seis funções arbitrárias responsáveis pela aplicação de um conjunto de operações sobre os dados de um subdomínio na versão MPI bloqueante. Como as operações dessas funções são aplicadas por um processo sobre os dados de um mesmo subdomínio, existem dependências entre as funções que devem ser resolvidas a fim de que os resultados produzidos por cada função sejam corretos. As operações de cada função modificam os dados do subdomínio que, por sua vez, são as entradas das funções subsequentes que dependem dessas alterações para a obtenção dos resultados esperados. Ou seja, a ordem de execução dessas funções é fundamental para que a execução do modelo seja bem sucedida, dependência essa resolvida na versão MPI bloqueante por meio da execução sequencial dessas funções sobre os dados de um mesmo subdomínio.

No entanto, na transcrição da versão MPI bloqueante para uma versão baseada em tarefas OpenMP essas seis funções são transformadas em tarefas que podem ser executadas simultaneamente sobre os dados de um mesmo subdomínio. A medida que essas tarefas são executadas paralelamente podem surgir problemas como concorrência no acesso a esses dados e, consequentemente, inconsistências críticas para a resolução do modelo. Para impedir que tarefas em um mesmo subdomínio concorram no acesso aos dados e, além disso, garantir que sejam executas na ordem correta as dependências de cada tarefa são explicitamente definidas por meio da cláusula OpenMP `depend`.

Na versão MPI bloqueante os subdomínios resultantes do particionamento do LBM são processados paralelamente por processos individuais. Isto é, para cada subdomínio existe um processo que executa as seis funções sequencialmente sobre seus dados de forma que a execução dessas funções em cada subdomínio seja simultânea e paralela. Para que a versão baseada em tarefas OpenMP tenha um comportamento semelhante cada uma das seis tarefas é criada igualmente para cada subdomínio. Ou seja, determinada tarefa é criada para todos os subdomínios e somente então a próxima tarefa é criada, novamente para todos os subdomínios, seguindo assim sucessivamente. Dessa forma, tarefas semelhantes

em subdomínios arbitrários que não possuem dependências entre si podem ser executadas paralelamente da mesma forma que na versão MPI bloqueante.

Além disso, na versão MPI bloqueante existe um processo principal responsável pela geração dos processos secundários e na versão baseada em tarefas OpenMP não é diferente. Tarefas OpenMP obrigatoriamente devem ser geradas e executadas em uma região paralela por um time de *threads*. Dessa forma, cada *thread* em um time gera cada uma das tarefas existentes em uma região paralela resultando em réplicas de tarefas que aplicam as mesmas operações sobre os mesmos conjuntos de dados. Para evitar a geração replicada de tarefas no LBM somente a *thread* principal é encarregada da geração de cada uma das tarefas de forma que o restante das *threads* no time apenas execute essas tarefas.

No Código 3.2 é apresentada uma visão geral da geração das tarefas realizada pela *thread* principal de um time de *threads* no LBM. Nele é possível observar que em uma região paralela (criada na linha 1) somente a *thread* principal do time (linha 3) processa os laços de repetição nos quais as seis tarefas são geradas para cada subdomínio. Se a linha 3 fosse removida todas as *threads* do time processariam os laços e gerariam tantas replicações das mesmas tarefas para cada subdomínio quanto o número de *threads* no time.

Observando o Código 3.2 é possível notar claramente que as mesmas funções presentes na versão MPI bloqueante estão localizadas cada uma no laço mais interno de três laços de repetição do tipo `for` aninhados. Essa nítida diferença na estrutura de invocação das funções se deve ao particionamento do domínio. Na versão MPI bloqueante o particionamento de domínio é realizada de forma distribuída, ou seja, a definição dos subdomínios não é pré-requisito para a geração dos processos secundários pelo processo principal uma vez que cada processo define qual é seu subdomínio de acordo com a sua identificação. Já na versão baseada em tarefas a definição dos subdomínios é pré-requisito para a geração das tarefas pela *thread* principal pois cada tarefa é gerada especificamente para determinado subdomínio. Ou seja, os laços de repetição são utilizados para iterar sobre os subdomínios a fim de que a *thread* possa gerar as tarefas para cada subdomínio.

A primeira tarefa gerada pela *thread* principal é a de inicialização da densidade. Para cada subdomínio a *thread* principal gera uma tarefa (linha 8) que aplicará as operações contidas na função de inicialização (linha 9) sobre os dados do subdomínio para o qual a tarefa foi gerada. Apesar de ser a primeira tarefa gerada na execução do método e, portanto, não haver tarefas predecessoras das quais ela possa depender, é necessária a definição das dependências de dados existentes na tarefa de inicialização por meio da cláusula OpenMP `depend` para que a ordem de execução das seis tarefas nos subdomínios seja a mesma ordem em que são geradas pela *thread* principal para cada subdomínio. As dependências das seis tarefas do LBM são detalhadas na subseção 3.3.1.

Código 3.2 – Visão geral da implementação baseada em tarefas.

```
1 #pragma omp parallel
2 {
3 #pragma omp master
4   {
5     for (bx = 0; bx < nbx; bx++)
6       for (by = 0; by < nby; by++)
7         for (bz = 0; bz < nbz; bz++) {
8 #pragma omp task depend()
9           initialize();
10        }
11    for (time = 0; time < t_max; time++) {
12      for (bx = 0; bx < nbx; bx++)
13        for (by = 0; by < nby; by++)
14          for (bz = 0; bz < nbz; bz++) {
15 #pragma omp task depend()
16            redistribute();
17          }
18      for (bx = 0; bx < nbx; bx++)
19        for (by = 0; by < nby; by++)
20          for (bz = 0; bz < nbz; bz++) {
21 #pragma omp task depend()
22            propagate();
23          }
24      for (bx = 0; bx < nbx; bx++)
25        for (by = 0; by < nby; by++)
26          for (bz = 0; bz < nbz; bz++) {
27 #pragma omp task depend()
28            position_copy();
29          }
30      for (bx = 0; bx < nbx; bx++)
31        for (by = 0; by < nby; by++)
32          for (bz = 0; bz < nbz; bz++) {
33 #pragma omp task depend()
34            bounceback();
35          }
36      for (bx = 0; bx < nbx; bx++)
37        for (by = 0; by < nby; by++)
38          for (bz = 0; bz < nbz; bz++) {
39 #pragma omp task depend()
40            relaxation();
41          }
42    }
43  }
44 }
```

3.3.1 Dependências entre Tarefas

Como visto na seção 2.4, as dependências de uma tarefa OpenMP podem ser definidas explicitamente por meio da cláusula `depend` e podem ser de três tipos: `in`, `out` e `inout`. Esses três tipos de dependência são utilizados para especificar o tipo de ação que uma tarefa executa sobre os dados armazenados em determinado endereço da memória de forma que o escalonamento das tarefas seja realizado com base nas ações e nos endereços acessados por cada tarefa. Normalmente tarefas sem a definição explícita de dependências por meio da cláusula `depend` são escalonadas de forma não determinística. No entanto, por meio da cláusula `depend` é possível impor restrições adicionais no escalonamento das tarefas de modo que o mesmo se torne parcialmente determinístico.

Na versão baseada em tarefas foram mantidas as mesmas estruturas de dados existentes na versão MPI bloqueante e é sobre os dados contidos nessas estruturas que as seis tarefas aplicam suas operações. Apesar de em ambas as versões haver o compartilhamento dessas estruturas entre as tarefas e entre as funções, respectivamente, a principal diferença entre a versão baseada em tarefas e a versão MPI bloqueante é a possibilidade de execução simultânea de ambas as tarefas sobre as mesmas estruturas de dados de um subdomínio, fato esse que pode resultar em concorrências no acesso aos dados e inconsistências críticas na execução do método. Portanto, utilizou-se das possibilidades de definição de restrições no escalonamento das tarefas oferecidas pela cláusula `depend` para que o escalonamento das tarefas siga a ordem de geração (que é a ordem correta de execução das tarefas) e para que, ao mesmo tempo, o paralelismo na execução das tarefas em diferentes subdomínios seja explorado ao máximo.

Para definir as dependências de uma tarefa por meio da cláusula `depend` é necessário especificar uma ou mais listas de elementos dos quais a tarefa depende, bem como o tipo de dependência existente entre a tarefa e os elementos em cada lista. Os elementos de uma lista de dependências podem ser variáveis de diferentes tipos de dados, inclusive matrizes e ponteiros. No LBM as duas estruturas de dados utilizadas pelas tarefas são matrizes multidimensionais alocadas na memória e acessadas por meio de variáveis do tipo ponteiro, ou seja, por meio dos endereços na memória nos quais os dados das estrutura estão armazenados. Como o acesso a essas estruturas é realizado por meio de ponteiros a definição de dependências entre as tarefas e as estruturas deve ser realizado por meio da sintaxe de seção de vetores.

3.3.1.1 Tarefa de Inicialização

Conforme mencionado na seção 3.3, a tarefa de inicialização é a primeira tarefa gerada em cada subdomínio e, apesar de não depender da execução de nenhuma tarefa, a definição das operações realizadas pela tarefa sobre os dados dos quais depende é fundamental para o escalonamento bem sucedido das tarefas subsequentes. O fato da

geração das seis tarefas ser realizada em uma mesma região paralela permite que ambas as tarefas sejam executadas simultaneamente em um mesmo subdomínio quando não houver a declaração das operações realizadas por cada tarefa, o que resulta em inconsistências na resolução do método uma vez que as duas estruturas de dados que armazenam as informações de densidade de um subdomínio são compartilhadas entre as seis tarefas. Além disso, se dentre as seis tarefas existir uma ou mais tarefas sem a definição das operações que realizam sobre os dados armazenados nas duas estruturas de dados o escalonamento dessas tarefas não será determinístico e, dessa forma, a execução dessas tarefas será em simultâneo com as demais tarefas, independentemente das operações realizadas por elas. Portanto, como os dados produzidos pela tarefa de inicialização são os dados de entrada da segunda tarefa, se somente as operações realizadas pela tarefa de redistribuição fossem definidas as duas tarefas seriam executadas simultaneamente e de forma não determinística em um mesmo subdomínio, o que, consequentemente, resultaria em inconsistências nos resultados produzidos por ambas as tarefas.

Visto que a tarefa de redistribuição deve ser executada somente após a execução da tarefa de inicialização é necessário determinar, por meio da cláusula `depend`, as operações aplicadas pela tarefa sobre os dados armazenados na memória. Como o escalonamento das tarefas OpenMP é realizado de acordo com o tipo de operações aplicadas sobre os dados armazenados em endereços na memória, se somente as operações aplicadas pela tarefa de redistribuição fossem definidas e as operações da tarefa de inicialização não o escalonador assumiria que a tarefa de redistribuição não depende da execução da tarefa de inicialização pois essa não possui nenhuma declaração de operações realizadas sobre os dados armazenados no mesmo endereço na memória sobre os quais a tarefa de redistribuição aplica suas operações, mesmo que a geração da tarefa de inicialização seja anterior à da tarefa de redistribuição e que ambas apliquem suas operações sobre os dados armazenados no mesmo endereço na memória. Dessa forma, ao determinar as operações aplicadas pela tarefa de inicialização por meio da cláusula `depend` garante-se que a execução da tarefa de redistribuição dar-se-á de acordo com o tipo das operações aplicadas pela tarefa de inicialização sobre os dados armazenados na memória.

A tarefa de inicialização da densidade depende somente da matriz principal de um subdomínio para armazenar os resultados produzidos pela sua execução. Para cada subdomínio é gerada uma tarefa de inicialização que tem como incumbência atribuir um valor inicial para a densidade de cada partícula do subdomínio. Pelo fato da matriz principal não conter nenhum dado de entrada para a tarefa de inicialização e, sim, ser utilizada apenas para armazenar a densidade de cada partícula do subdomínio no endereço na memória em que a matriz principal está alocada, o tipo de dependência existente entre a tarefa e o endereço na memória relativo à matriz principal é `out`.

No Código 3.3 é apresentada a diretiva OpenMP completa utilizada para a geração

da tarefa de inicialização da densidade para cada subdomínio. É possível observar que nessa diretiva existe apenas uma lista de dependências contendo um único elemento, denominado `node`. Como a memória necessária para armazenar as informações sobre a densidade dos subdomínios em estruturas multidimensionais é alocada dinamicamente, o acesso a essas estruturas é realizada por meio do endereço na memória no qual as estruturas estão armazenadas e, portanto, o elemento `node` nada mais é do que uma variável do tipo ponteiro que armazena esse endereço. Dessa forma, para a explicitação da dependência existente entre a tarefa e a estrutura na cláusula `depend` é utilizada a sintaxe de seção de matriz por meio da qual é definido que a tarefa possui uma dependência do tipo `out` sobre os dados armazenados no endereço referenciado pelo ponteiro `node`.

Código 3.3 – Diretiva para a geração da tarefa de inicialização da densidade.

```
1 #pragma omp task untied shared(node) depend(out : node[ : 1])
```

Como o valor armazenado na variável `node` é o endereço na memória no qual as informações da densidade de um subdomínio estão armazenadas, a sintaxe de seção de matriz possibilita a explicitação da dependência existente entre a tarefa e a estrutura de dados para a qual a variável aponta. Sem a utilização dessa sintaxe a dependência da tarefa seria relacionada ao endereço na memória no qual o valor da variável `node` está armazenado e não ao endereço no qual as informações da densidade do subdomínio estão armazenadas. Ou seja, a tarefa dependeria da variável e não das informações da densidade do subdomínio para qual o endereço armazenado na variável aponta.

3.3.1.2 Tarefa de Redistribuição

Após a finalização da execução da tarefa de inicialização da densidade em determinado subdomínio, a próxima tarefa a ser executada é a de redistribuição das forças do subdomínio. Essa tarefa aplica suas operações somente sobre as informações de densidade do subdomínio para o qual foi gerada e depende do resultado das operações aplicadas pela tarefa de inicialização sobre as informações de densidade do subdomínio para redistribuir as forças com sucesso no subdomínio. Ou seja, a tarefa de redistribuição depende dos dados contidos na estrutura que armazena as informações de densidade do subdomínio para que possa aplicar suas operações e, dessa forma, modificar essas informações. Portanto, o tipo da dependência existente entre a tarefa de redistribuição e a estrutura é `inout`.

No Código 3.4 é apresentada a diretiva OpenMP utilizada na geração da tarefa de redistribuição das forças para cada subdomínio. Apesar de possuir uma dependência do tipo `inout` essa diretiva é idêntica à diretiva utilizada para a geração das tarefas de inicialização, uma vez que tanto o tipo `out` quanto `inout` impõem restrições no escalonamento de forma que as tarefas sejam executadas somente após a finalização de todas as tarefas geradas anteriormente e que dependem de endereços de memória em

comum, independente do tipo de dependência. Desse modo, utilizou-se o tipo `inout` para tornar transparente quais ações são tomadas pela tarefa.

Código 3.4 – Diretiva para a geração da tarefa de redistribuição.

```
1 #pragma omp task untied shared(node) depend(inout : node[ : 1])
```

3.3.1.3 Tarefa de Propagação

Finalizadas as operações da tarefa de redistribuição, seguindo a ordem de execução das tarefas para um mesmo subdomínio, a tarefa que deve obrigatoriamente ser executada na sequência é a tarefa de propagação das forças. Diferentemente das tarefas de inicialização e redistribuição que possuem uma única dependência, a tarefa de propagação depende de duas estruturas de dados: da matriz principal `density`; e de uma matriz auxiliar idêntica à matriz principal. A alocação de memória para a matriz auxiliar é realizada dinamicamente e, portanto, o acesso a essa matriz é realizado por meio do endereço na memória relativo à matriz e que está armazenado em uma variável do tipo ponteiro denominada `aux`.

A tarefa de propagação é responsável pela propagação das forças de cada partícula em um subdomínio nas dezoito direções de propagação de acordo com o modelo D3Q19 apresentado na seção 1.2. No entanto, a propagação das forças somente é possível após a finalização da execução da tarefa de redistribuição, uma vez que os resultados produzidos pelas operações dessa tarefa e armazenados na matriz principal são necessários para a execução da tarefa de propagação e consequente armazenamento de seus resultados na matriz auxiliar. Ou seja, a tarefa de propagação não altera as informações contidas na matriz principal, mas as utiliza para realizar a propagação das forças e armazenar os resultados na matriz auxiliar modificando, assim, apenas a matriz auxiliar independentemente das informações armazenadas na mesma. Dessa forma, o tipo da dependência existente entre a tarefa de propagação e as matrizes principal e auxiliar é `in` e `out`, respectivamente.

No Código 3.5 é apresentada a diretiva OpenMP utilizada para a geração da tarefa de propagação das forças para cada subdomínio. Como mencionado anteriormente, a tarefa de propagação das forças depende das modificações realizadas pela tarefa de redistribuição na matriz principal e, portanto, ao especificar essa dependência por meio da cláusula `depend`, são impostas restrições no escalonamento dessas tarefas que impedem a execução de ambas as tarefas paralelamente sobre a matriz principal de um mesmo subdomínio. Apesar de depender não só da matriz principal mas também da matriz auxiliar, por ser a primeira dentre as três tarefas a utilizar essa matriz e, além disso, já possuir uma dependência em relação à matriz principal que as impeça de executar em paralelo, essa dependência não altera as restrições entre a tarefa de redistribuição e de propagação. Dessa forma, a diretiva de geração da tarefa de propagação é composta por

Código 3.5 – Diretiva para a geração da tarefa de propagação das forças.

```
1 #pragma omp task untied shared(density, aux)
2                         depend(in : density[ : 1])
3                         depend(out : aux[ : 1])
```

duas listas de dependências, uma do tipo `in` contendo apenas o ponteiro para a matriz principal e outra do tipo `out` contendo apenas o ponteiro para a matriz auxiliar.

Ao gerar as tarefas de redistribuição e de propagação das forças nessa ordem e definir suas dependências por meio da cláusula `depend` o escalonamento dessas duas tarefas se torna determinístico. Isso se deve pelo fato da tarefa de redistribuição ser gerada anteriormente à tarefa de propagação e a dependência existente entre ambas e a matriz principal impedir que as tarefas sejam executadas simultaneamente em um mesmo subdomínio ou que sejam executas em qualquer outra ordem a não ser a ordem em que são geradas. Dessa forma, garante-se a consistência das informações durante a execução das tarefas de redistribuição e propagação das forças.

3.3.1.4 Tarefa de Cópia de Bordas

Diferentemente das três tarefas vistas até então, a tarefa a ser executada na sequência não depende somente do subdomínio para o qual foi gerada, mas também dos subdomínios vizinhos a esse subdomínio. No LBM partículas movem-se livremente por todo o domínio em dezoito direções distintas e, ao particionarmos esse domínio em subdomínios, essa movimentação das partículas por todo o domínio deve ser mantida. No entanto, partículas localizadas nas extremidades de um subdomínio podem mover-se para uma posição que não pertence ao seu subdomínio e, sim, a outro subdomínio. Portanto, esse subdomínio é considerado um subdomínio vizinho e, para que a execução do método seja bem sucedida, são realizadas cópias das partículas em determinadas extremidades dos dezoito subdomínios vizinhos de cada subdomínio (denominadas bordas vizinhas), de acordo com as dezoito direções de propagação das forças do modelo D3Q19 apresentado na seção 1.2.

Como mencionado na seção 3.2, tanto na matriz principal quanto na matriz auxiliar são alocados espaços adicionais na memória para o armazenamento da cópia de cada borda vizinha. Apesar disso, para a cópia das bordas somente são utilizadas as matrizes auxiliares de cada subdomínio. Ou seja, as bordas dos subdomínios vizinhos de um subdomínio são copiadas a partir da matriz auxiliar de cada subdomínio vizinho ou, em casos onde não existem subdomínios vizinhos em algumas das direções de propagação das forças, a partir da própria matriz auxiliar do subdomínio e armazenadas na mesma. Dessa forma, o tipo de dependência existente entre a tarefa de cópia das bordas vizinhas e a matriz auxiliar do subdomínio para o qual foi gerada e as matrizes auxiliares de seus subdomínios vizinhos

Código 3.6 – Diretiva para a geração da tarefa de cópia de bordas dos subdomínios vizinhos.

```
1  #pragma omp task untied shared(aux, east, north_east, north,
2                                 north_west, west, south_west,
3                                 south, south_east, bottom, top,
4                                 east_bottom, east_top,
5                                 north_bottom, north_top,
6                                 west_bottom, west_top,
7                                 south_bottom, south_top)
8                          depend(inout : aux[ : 1])
9                          depend(in : east[ : 1], north[ : 1],
10                                 west[ : 1], south[ : 1],
11                                 bottom[ : 1], top[ : 1],
12                                 north_east[ : 1],
13                                 north_west[ : 1],
14                                 south_west[ : 1],
15                                 south_east[ : 1],
16                                 east_bottom[ : 1],
17                                 east_top[ : 1],
18                                 west_bottom[ : 1],
19                                 west_top[ : 1],
20                                 north_bottom[ : 1],
21                                 north_top[ : 1],
22                                 south_bottom[ : 1],
23                                 south_top[ : 1])
```

é `inout` e `in`, respectivamente.

No Código 3.6 é apresentada a diretiva OpenMP utilizada para a geração da tarefa de cópia das bordas dos subdomínios vizinhos para cada subdomínio. Para que a cópia das bordas dos subdomínios vizinhos possa ser realizada em um subdomínio é necessário não só que a execução da tarefa de propagação tenha finalizado suas operações mas que, além disso, em cada subdomínio vizinho as respectivas tarefas de propagação das forças também tenham finalizado suas operações. No entanto, dependendo do particionamento de domínio adotado, em algumas direções de propagação das forças pode não haver subdomínios vizinhos e, nesse caso, a cópia das bordas é realizada entre as respectivas bordas do próprio subdomínio. Desse modo, existe uma lista de dependências do tipo `inout` contendo apenas o ponteiro da matriz auxiliar do subdomínio para o qual a tarefa foi gerada e outra lista de dependências do tipo `in` contendo os ponteiros para as matrizes auxiliares dos dezoito subdomínios vizinhos desse subdomínio.

Por possuir uma dependência do tipo `inout` em relação à matriz auxiliar do subdomínio para o qual a tarefa de cópia das bordas dos subdomínios vizinhos é gerada, algumas restrições adicionais podem ser impostas no escalonamento das tarefas de cópia das bordas vizinhas à medida que são executadas em cada subdomínio. Ao passo que a

Código 3.7 – Diretiva para a geração da tarefa de bounceback.

```
1 #pragma omp task untied shared(density, aux)
2                         depend(out : density[ : 1])
3                         depend(in : aux[ : 1])
```

execução de uma tarefa de cópia das bordas vizinhas é iniciada em um subdomínio, as demais tarefas de cópia das bordas vizinhas geradas posteriormente e que dependem das informações contidas na matriz auxiliar desse subdomínio somente poderão ser executadas após a finalização da execução dessa tarefa. Apesar de não depender dos resultados das operações de tarefas de cópia das bordas vizinhas geradas anteriormente, não é possível realizar operações de leitura e escrita em uma mesma estrutura de dados simultaneamente. Dessa forma, durante a execução das tarefas de cópia das bordas vizinhas há uma certa sequencialização imposta pelas dependências de modo que não haja inconsistências no acesso às informações das matrizes auxiliares vizinhas. O escalonamento e execução dessa e das demais tarefas será discutido detalhadamente na subseção 3.3.2.

3.3.1.5 Tarefa de Bounceback

Após geradas as tarefas de cópia das bordas é a vez da *thread* principal gerar as tarefas de bounceback para cada subdomínio. Essa tarefa realiza a inversão da direção da velocidade das partículas em colisão com as bordas ou com os obstáculos presentes no reticulado a partir dos resultados da execução da tarefa de propagação e da tarefa de cópia das bordas em um mesmo subdomínio. Conforme visto na subseção 3.3.1.3 e na subseção 3.3.1.4, tanto a tarefa de propagação quanto a tarefa de cópia das bordas armazenam seus resultados na matriz auxiliar e, portanto, é a partir das informações contidas nessa matriz que a tarefa de bounceback aplica suas operações. Apesar da tarefa depender das informações contidas na matriz auxiliar, os resultados de sua execução são armazenados na matriz principal de cada subdomínio.

A diretiva OpenMP com a declaração das respectivas dependências utilizada na geração das tarefas de bounceback é apresentada no Código 3.7. Pelo fato da tarefa de bounceback não modificar as informações contidas na matriz auxiliar e, sim, apenas realizar a leitura dessas informações, a dependência existente entre a tarefa e a matriz auxiliar é do tipo `in`. Já em relação à matriz principal, ao utilizá-la apenas para a escrita dos resultados independentemente das informações armazenadas a dependência existente entre a tarefa e a matriz é do tipo `out`. Portanto, a execução da tarefa de bounceback somente pode ser realizada quando não houverem tarefas geradas anteriormente em execução que modificam ambas as estruturas ou que realizam a leitura das informações armazenadas na matriz principal, ou seja, que possuam dependências do tipo `out` ou `inout` em relação a ambas as matrizes ou do tipo `in` em relação à matriz principal.

Código 3.8 – Diretiva para a geração da tarefa de relaxação.

```
1 #pragma omp task untied shared(density, aux)
2                         depend(out : density[ : 1])
3                         depend(in : aux[ : 1])
```

3.3.1.6 Tarefa de Relaxação

A última tarefa gerada pela *thread* principal antes de repetir a geração das tarefas localizadas no interior do laço, no caso de `t_max` ser maior que 1, é a tarefa de relaxação. Essa tarefa realiza a distribuição de equilíbrio do fluido de acordo com uma taxa de aproximação do equilíbrio (viscosidade do fluido) em um subdomínio. Como a tarefa de relaxação, assim como a de bounceback, depende dos resultados das tarefas de propagação e cópia das bordas, é a partir da matriz auxiliar de cada subdomínio que a tarefa obtém as informações necessárias para realizar a distribuição de equilíbrio. Os resultados dessas operações, por sua vez, são armazenados na matriz principal de cada subdomínio da mesma forma que na tarefa de bounceback.

No Código 3.8 é apresentada a diretiva OpenMP com a declaração das respectivas dependências utilizada na geração das tarefas de relaxação. Como é possível observar, a diretiva utilizada na geração das tarefas de relaxação é a mesma utilizada na geração das tarefas de bounceback. Isso se deve pelo fato da leitura das informações necessárias para a aplicação das operações em ambas as tarefas ser realizada a partir da matriz auxiliar e da escrita dos resultados ser realizada na matriz principal. Dessa forma, a dependência existente entre a tarefa de relaxação e as matrizes auxiliar e principal é `in` e `out`, respectivamente, impedindo que a tarefa seja executada antes da finalização da execução das tarefas geradas anteriormente e que modificam uma ou ambas as matrizes ou que realizam a leitura das informações contidas na matriz principal.

3.3.2 Escalonamento de Tarefas

Na subseção 3.3.1 foram detalhadas as operações realizadas por cada tarefa em relação às informações armazenadas nas matrizes auxiliares e principais a partir das quais foram definidos os tipos de dependência existentes entre as tarefas e as matrizes e, consequentemente, utilizadas na declaração das diretivas de geração de cada tarefa. É a partir dessas diretivas que a *thread* principal gera as seis tarefas para cada subdomínio para que as demais *threads* do time possam executá-las. No entanto, a execução dessas tarefas é controlada pelo escalonador de tarefas do OpenMP de acordo com a ordem de geração e as dependências existentes entre as tarefas e o endereço na memória em que ambas as matrizes são alocadas.

Na Figura 11 é apresentado um Grafo Acíclico Dirigido (*Directed Acyclic Graph* –

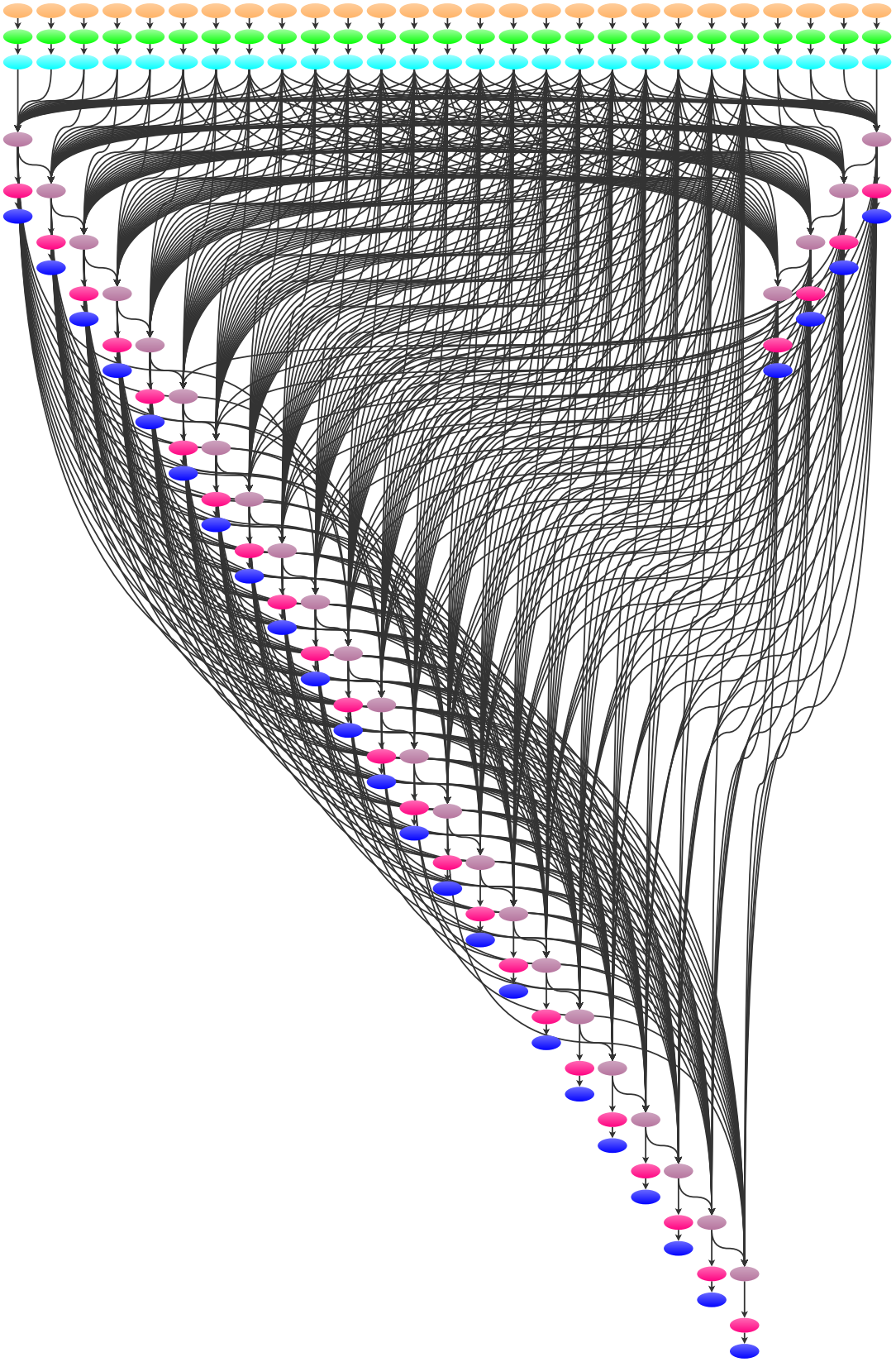
DAG) do escalonamento das seis tarefas em uma execução hipotética do particionamento de um domínio qualquer em 27 subdomínios com `t_max` igual a 1. Nesse grafo assume-se que existe uma quantidade suficiente de *threads* no time de modo que a execução das tarefas seja realizada com o máximo de paralelismo possível nesse particionamento de acordo com as dependências de cada tarefa. Além disso, a ordem de geração das tarefas bem como a vizinhança de cada subdomínio foram definidos com o intuito de evidenciar o motivo de haver ou não paralelismo em determinados pontos da execução.

Nesse grafo as seis tarefas são representadas por vértices com cores únicas para cada tarefa. Esses vértices são organizados em sequências verticais e sequências horizontais. As sequências verticais representam a execução de diferentes tarefas em um mesmo subdomínio de forma ordenada. Já as sequências horizontais representam a execução tanto de tarefas semelhantes quanto de tarefas distintas em diferentes subdomínios de forma paralela. Além disso, as dependências existentes entre as tarefas são representadas por arestas que partem do vértice de dependência em direção ao vértice dependente.

A primeira tarefa executada pelas *threads* do time é a de inicialização. O fato de ser a primeira tarefa gerada em cada subdomínio e depender apenas da matriz principal do subdomínio para o qual foi gerada permite que a tarefa de inicialização seja a primeira tarefa executada em cada subdomínio. No entanto, apesar de ser a primeira tarefa gerada isso não impede que o escalonador eventualmente programe alguma outra tarefa gerada após a tarefa de inicialização para ser a primeira tarefa executada em algum subdomínio. O que realmente impede que outras tarefas sejam executadas anteriormente à tarefa de inicialização é o tipo de dependência que a mesma possui em relação à matriz principal de cada subdomínio. Como a tarefa de inicialização possui uma dependência do tipo `out` em relação à matriz principal do subdomínio, tarefas que possuem o mesmo ou qualquer outro tipo de dependência em relação à matriz principal não são escalonadas antes da finalização da execução da tarefa de inicialização. Portanto, ao depender somente da matriz principal do subdomínio para o qual foi gerada, a sequência horizontal de vértices na cor laranja representa a execução das 27 tarefas de inicialização simultaneamente em cada subdomínio.

Finalizadas as execuções das tarefas de inicialização, a próxima tarefa escalonada para ser executada pelas *threads* do time é a tarefa de redistribuição. Essa tarefa é gerada logo após a tarefa de inicialização e, assim como a tarefa de inicialização, depende somente da matriz principal do seu subdomínio. O fato da tarefa de redistribuição possuir uma dependência do tipo `inout` em relação à matriz principal a impede de ser executada antes da finalização da execução da tarefa de inicialização. Essa dependência é evidenciada no grafo pela aresta que parte do vértice que representa a tarefa de inicialização em direção ao vértice da tarefa de redistribuição em cada subdomínio. Além disso, essa dependência também impede que tarefas geradas posteriormente sejam executadas antes da finalização da tarefa de redistribuição, independentemente do tipo de dependência existente entre as

Figura 11 – Gráfico de dependências da implementação OpenMP Tarefas.



Fonte: O autor.

tarefas e a matriz principal de um subdomínio. Dessa forma, a sequência horizontal de vértices na cor verde representa a execução simultânea das 27 tarefas de redistribuição em cada subdomínio uma vez que a tarefa de redistribuição depende somente da matriz principal do subdomínio para o qual foi gerada.

A tarefa de propagação, por sua vez, é executada pelas *threads* do time somente após a finalização da execução da tarefa de redistribuição. Além de ser gerada após a tarefa de redistribuição, essa tarefa também depende da matriz principal bem como da matriz auxiliar do subdomínio para o qual foi gerada. Em relação à matriz principal, por possuir uma dependência do tipo `in` a tarefa de propagação é impedida de ser executada antes da finalização da execução da tarefa de redistribuição. Essa dependência é representada no grafo por uma aresta que parte do vértice da tarefa de redistribuição em direção ao vértice da tarefa de propagação em cada subdomínio. Já em relação à matriz auxiliar, pelo fato de ser a primeira tarefa a depender da matriz auxiliar não são impostas restrições sobre o escalonamento das tarefas geradas anteriormente à tarefa de propagação, somente sobre o escalonamento das tarefas geradas posteriormente que possuem algum tipo de dependência com essa matriz. Tarefas que dependem da matriz auxiliar e que são geradas após a tarefa de propagação somente podem ser executadas ao final da execução da tarefa de propagação devido à dependência do tipo `out` existente entre a tarefa e a matriz auxiliar. Portanto, ao depender somente das matrizes do subdomínio para o qual foram geradas, as 27 tarefas de propagação podem ser executadas simultaneamente em cada subdomínio conforme representado pela sequência horizontal de vértices da cor ciano no grafo.

A execução da tarefa de cópia das bordas pelas *threads* do time, diferentemente das demais tarefas, somente é realizada após a finalização da execução de cada uma das tarefas de propagação das quais depende. Essa tarefa depende de 19 matrizes auxiliares distintas: da matriz auxiliar do subdomínio para o qual foi gerada; e das 18 matrizes auxiliares dos subdomínios vizinhos do seu subdomínio. O fato da tarefa ser gerada após a tarefa de propagação juntamente com o fato da dependência existente entre a tarefa e a matriz auxiliar do seu subdomínio ser do tipo `inout` e entre a tarefa e as matrizes auxiliares dos subdomínios vizinhos do seu subdomínio ser do tipo `in` impede que a tarefa de cópia das bordas em cada subdomínio seja executada antes da finalização da execução das 19 tarefas de propagação relativas à esses subdomínios. Cada uma dessas dependências é representada no grafo por arestas com origem nos vértices das tarefas de propagação tanto do próprio subdomínio da tarefa quanto dos subdomínios vizinhos e término no vértice na cor lilás que representa a tarefa de cópia das bordas em cada subdomínio.

Ao contrário das demais tarefas, a ordem de execução das tarefas de cópia das bordas depende diretamente da ordem de geração de cada uma das tarefas para os diferentes subdomínios. Isso se deve pelo fato de cada tarefa de cópia das bordas modificar as informações contidas na matriz auxiliar do subdomínio para o qual foi gerada e que,

por sua vez, é uma das dependências dos seus subdomínios vizinhos. A medida que uma tarefa de cópia das bordas é gerada para um subdomínio após a geração da tarefa de cópia das bordas para um de seus subdomínios vizinhos, o fato da tarefa gerada anteriormente possuir uma dependência do tipo `in` em relação à matriz desse subdomínio e a tarefa gerada posteriormente possuir uma dependência do tipo `inout` em relação à mesma matriz essa somente pode ser executada após a finalização da tarefa gerada anteriormente. Nesse caso, a tarefa gerada posteriormente não depende mais da finalização da tarefa de propagação em seu subdomínio e, sim, da tarefa de cópia das bordas gerada anteriormente para um de seus subdomínios vizinhos. Essa dependência é representada no grafo por arestas com origem em uma tarefa de cópia das bordas de um subdomínio e término em outra tarefa de cópia das bordas de outro subdomínio executada posteriormente. Como pode ser observado, apenas duas tarefas de cópia das bordas no grafo não dependem de outras tarefas de cópia das bordas, o restante das tarefas depende de pelo menos uma outra tarefa de cópia das bordas executada anteriormente e até mesmo somente de outras tarefas de cópia das bordas, que é o caso da última tarefa executada.

Para evidenciar as possibilidades de paralelismo existentes na execução da tarefa de cópia das bordas a vizinhança de alguns subdomínios no grafo foi definida de modo que exista o mínimo de dependências possível. A partir do subdomínio na extremidade esquerda do grafo, os dezoito subdomínios localizados à direita de cada subdomínio foram definidos como os seus subdomínios vizinhos dos quais cada tarefa de cópia das bordas depende. No entanto, apenas para os quatro subdomínios na extremidade à direita do grafo foram definidos como os dezoito subdomínios vizinhos os subdomínios localizados à esquerda a partir do subdomínio da extremidade direita. Dessa forma, as tarefas de cópia das bordas que se encontram nas duas extremidades horizontais são as primeiras tarefas executadas pelas *threads* do time pois, hipoteticamente, foram as duas primeiras tarefas geradas pela *thread* principal e não possuem subdomínios vizinhos em comum. Isso também ocorre para as três tarefas geradas na sequência em cada uma das extremidades horizontais do grafo e, a partir de então, as demais tarefas são executadas sequencialmente devido à existência de subdomínios comuns em ambas as tarefas.

Após finalizada a execução da tarefa de cópia das bordas em um subdomínio, independentemente da execução das demais tarefas de cópia das bordas, as *threads* do time executam a tarefa de bounceback. O fato de ser gerada após a tarefa de cópia das bordas e possuir uma dependência do tipo `in` em relação à matriz auxiliar a impede de ser executada antes da finalização da tarefa de cópia das bordas em seu subdomínio. Essa dependência é representada no grafo por meio de uma aresta com origem no vértice da tarefa de cópia das bordas e término no vértice da tarefa de bounceback de um mesmo subdomínio. No grafo o vértice que representa a tarefa de bounceback em cada subdomínio possui uma cor rosa e a execução praticamente é sequencial devido às dependências da tarefa de cópia das bordas.

A última tarefa executada pelas *threads* do time nesse grafo é a tarefa de relaxação. Ao ser gerada após a tarefa de bounceback e possuir uma dependência do tipo `out` em relação à matriz principal do subdomínio, da mesma forma que a tarefa de bounceback, essa somente é executada após a finalização da execução da tarefa de bounceback. Essa dependência é representada no grafo por meio de uma aresta com origem no vértice da tarefa de bounceback e término no vértice da tarefa de relaxação de um mesmo subdomínio. Apesar de possuir ainda uma dependência do tipo `in` em relação à matriz auxiliar do seu subdomínio, essa não adiciona restrições no escalonamento da tarefa. Por fim, a execução da tarefa de relaxação é representada no grafo por um vértice na azul em cada subdomínio.

3.4 OpenMP Tarefas com Buffers

Por meio da execução hipotética da implementação do LBM baseada em tarefas em um particionamento de domínio com um total de 27 subdomínios apresentada na subseção 3.3.2 foi possível observar que as dependências da tarefa de cópia das bordas limitam expressivamente o paralelismo na execução dessas tarefas pelas *threads* de um time. O fato da leitura das bordas dos subdomínios vizinhos ser realizada a partir de suas matrizes auxiliares e da cópia dessas bordas ser armazenada na matriz auxiliar de cada subdomínio impede que tarefas de cópia das bordas em subdomínios mutuamente vizinhos sejam executadas em paralelo. Essa restrição torna o escalonamento das tarefas de cópia das bordas nos diferentes subdomínios dependente da ordem de geração de cada tarefa de modo que o paralelismo na execução seja limitado às tarefas não mutuamente vizinhas e geradas em uma ordem em que não existam tarefas de cópia das bordas para os subdomínios dos quais dependem entre ambas. Portanto, com o intuito de minimizar as restrições impostas no escalonamento da tarefa de cópia das bordas por suas dependências, desenvolveu-se uma versão baseada em tarefas que utiliza matrizes temporárias (`buffers`) para realizar a cópia das bordas dos subdomínios vizinhos de cada subdomínio.

A implementação baseada em tarefas com `buffers` é similar à implementação baseada em tarefas apresentada na seção 3.3 com exceção da tarefa de cópia das bordas que foi dividida em duas tarefas distintas. Essas duas tarefas utilizam matrizes temporárias para realizar a cópia das bordas dos subdomínios vizinhos de cada subdomínio de forma totalmente paralela. Isso é possível pelo fato da leitura das bordas dos subdomínios vizinhos de um subdomínio ser realizada separadamente do armazenamento dessas bordas em cada subdomínio. Ao isolar a leitura realizada nas matrizes auxiliares dos subdomínios vizinhos do armazenamento das cópias das bordas desses subdomínios na matriz auxiliar em cada subdomínio remove-se as restrições que dificultam a execução paralela das tarefas de cópia das bordas existente na versão baseada em tarefas.

As matrizes temporárias utilizadas pelas duas tarefas para armazenar temporaria-

mente as bordas dos subdomínios vizinhos em cada subdomínio são idênticas às respectivas bordas que armazenam, isto é, possuem o mesmo tamanho. Para as bordas dos 6 subdomínios vizinhos ortogonais são alocadas na memória matrizes temporárias bidimensionais enquanto que para as bordas dos 12 subdomínios vizinhos diagonais são alocadas na memória matrizes temporárias unidimensionais. Os endereços na memória dessas matrizes são, então, armazenados na estrutura de dados `lattice` de cada subdomínio para que, posteriormente, sejam utilizados na declaração das dependências das duas tarefas de cópia das bordas.

No Código 3.9 é apresentada uma simplificação do Código 3.2 na qual é possível observar a substituição da geração da tarefa de cópia das bordas presente na implementação baseada em tarefas pela geração das duas tarefas de cópia das bordas utilizando `buffers`. Após a geração das tarefas de inicialização, redistribuição e propagação para cada subdomínio, a *thread* principal do time de *threads* gera a tarefa `position_copy_to_buffer` para cada subdomínio e, em seguida, gera a tarefa `position_copy_from_buffer` também para cada um dos subdomínios. Ao finalizar a geração dessas duas tarefas a *thread* principal ainda gera as tarefas de bounceback e, posteriormente, as tarefas de relaxação para cada subdomínio. Essas tarefas são geradas `t_max` vezes para cada subdomínio, com exceção da tarefa de inicialização que é gerada uma única vez para cada subdomínio.

3.4.1 Dependências das Tarefas

Na implementação baseada em tarefas com `buffers` as tarefas de inicialização, redistribuição, propagação, bounceback e relaxação possuem as mesmas dependências da implementação baseada em tarefas apresentadas na subseção 3.3.1. Dessa forma, na subseção 3.4.1.1 e na subseção 3.4.1.2 são detalhadas apenas as dependências das tarefas de cópia das bordas para as matrizes temporárias e de cópia das bordas a partir das matrizes temporárias para as matrizes auxiliares de cada subdomínio, respectivamente. Essas dependências são determinadas por meio da cláusula `depend` a partir dos endereços na memória tanto das matrizes temporárias quanto das matrizes auxiliares dos subdomínios, ambos os endereços armazenados na estrutura de dados `lattice`.

3.4.1.1 Tarefa de Cópia das Bordas para os Buffers

Após gerar as tarefas de propagação a *thread* principal do time de *threads* gera as tarefas de cópia das bordas para as matrizes temporárias para cada subdomínio. Essa tarefa, da mesma forma que a tarefa de cópia das bordas da implementação baseada em tarefas, depende tanto do resultado da execução da tarefa de propagação no subdomínio para o qual foi gerada quanto dos resultados das execuções das tarefas de propagação nos dezoito subdomínios vizinhos ao seu subdomínio. Os resultados da execução das tarefas de propagação são armazenados nas matrizes auxiliares de cada subdomínio e é a partir

Código 3.9 – Visão geral da implementação baseada em tarefas com buffers.

```
1 #pragma omp parallel
2 {
3 #pragma omp master
4   {
5     .
6     .
7     .
8     for (bx = 0; bx < nbx; bx++)
9       for (by = 0; by < nby; by++)
10        for (bz = 0; bz < nbz; bz++) {
11 #pragma omp task depend()
12          position_copy_to_buffer();
13        }
14    for (bx = 0; bx < nbx; bx++)
15      for (by = 0; by < nby; by++)
16        for (bz = 0; bz < nbz; bz++) {
17 #pragma omp task depend()
18          position_copy_from_buffer();
19        }
20    .
21    .
22    .
23  }
24 }
```

dessas matrizes que a tarefa obtém as informações contidas nas extremidades de cada subdomínio vizinho e as transcreve para as matrizes temporárias correspondentes a cada extremidade vizinha. Portanto, além de possuir uma dependência com a matriz auxiliar de cada subdomínio essa tarefa também possui uma dependência com cada matriz temporária que armazena a borda de um de seus subdomínios vizinhos.

No Código 3.10 é apresentada a diretiva OpenMP a partir da qual são geradas as tarefas de cópia das bordas para os `buffers` em cada subdomínio. Pelo fato da tarefa apenas realizar a leitura das informações armazenadas nas matrizes auxiliares dos subdomínios vizinhos ao seu subdomínio existe uma lista de dependências do tipo `in` contendo os endereços na memória de cada matriz auxiliar da qual a tarefa depende. Além disso, existe uma segunda lista de dependências do tipo `out` que contém os endereços na memória das matrizes temporárias do subdomínio para o qual a tarefa foi gerada em que apenas é realizado o armazenamento das cópias das bordas dos subdomínios vizinhos.

Código 3.10 – Diretiva para a geração da tarefa de cópia de bordas dos subdomínios vizinhos para os buffers.

```
1 #pragma omp task untied
2               shared(east, north_east, north, north_west, west,
3                      south_west, south, south_east, bottom, top,
4                      east_bottom, east_top, north_bottom,
5                      north_top, west_bottom, west_top,
6                      south_bottom, south_top, east_buffer,
7                      north_east_buffer, north_buffer,
8                      north_west_buffer, west_buffer,
9                      south_west_buffer, south_buffer,
10                     south_east_buffer, bottom_buffer,
11                     top_buffer, east_bottom_buffer,
12                     east_top_buffer, north_bottom_buffer,
13                     north_top_buffer, west_bottom_buffer,
14                     west_top_buffer, south_bottom_buffer,
15                     south_top_buffer)
16              depend(in : east[ : 1], west[ : 1], north[ : 1],
17                          south[ : 1], top[ : 1], bottom[ : 1],
18                          north_west[ : 1], south_west[ : 1],
19                          north_east[ : 1], south_east[ : 1],
20                          west_bottom[ : 1], west_top[ : 1],
21                          east_bottom[ : 1], east_top[ : 1],
22                          north_bottom[ : 1], north_top[ : 1],
23                          south_bottom[ : 1], south_top[ : 1])
24              depend(out : east_buffer[ : 1], west_buffer[ : 1],
25                           north_buffer[ : 1], south_buffer[ : 1],
26                           top_buffer[ : 1], bottom_buffer[ : 1],
27                           north_west_buffer[ : 1],
28                           south_west_buffer[ : 1],
29                           north_east_buffer[ : 1],
30                           south_east_buffer[ : 1],
31                           west_bottom_buffer[ : 1],
32                           west_top_buffer[ : 1],
33                           east_bottom_buffer[ : 1],
34                           east_top_buffer[ : 1],
35                           north_bottom_buffer[ : 1],
36                           north_top_buffer[ : 1],
37                           south_bottom_buffer[ : 1],
38                           south_top_buffer[ : 1])
```

3.4.1.2 Tarefa de Cópia das Bordas a Partir dos Buffers

Ao finalizar a geração das tarefas de cópia das bordas dos subdomínios vizinhos para as matrizes temporárias a *thread* principal gera as tarefas de cópia das bordas a partir das matrizes temporárias para as matrizes auxiliares de cada subdomínio. Essa tarefa depende do resultado da execução da tarefa de cópia das bordas dos subdomínios vizinhos para as matrizes temporárias uma vez que realiza a transferência das informações armazenadas nas matrizes temporárias para a matriz auxiliar do subdomínio para o qual foi gerada. Portanto, essa tarefa possui duas dependências, uma em relação às matrizes temporárias e outra em relação à matriz auxiliar, ambas pertencentes ao subdomínio para o qual a tarefa é gerada.

No Código 3.11 é apresentada a diretiva OpenMP a partir da qual são geradas as tarefas de cópia das bordas dos `buffers` para as matrizes auxiliares em cada subdomínio. Pelo fato dessa tarefa apenas realizar a leitura das informações armazenadas nas matrizes temporárias do subdomínio para o qual foi gerada existe uma lista de dependências do tipo `in` que contém os endereços na memória de cada matriz temporária do subdomínio. Além disso, essa tarefa ainda possui uma lista de dependências do tipo `out` que contém o endereço na memória da matriz auxiliar do subdomínio na qual são armazenadas as cópias das bordas dos subdomínios vizinhos obtidas a partir da leitura das matrizes temporárias.

3.4.2 Escalonamento das Tarefas

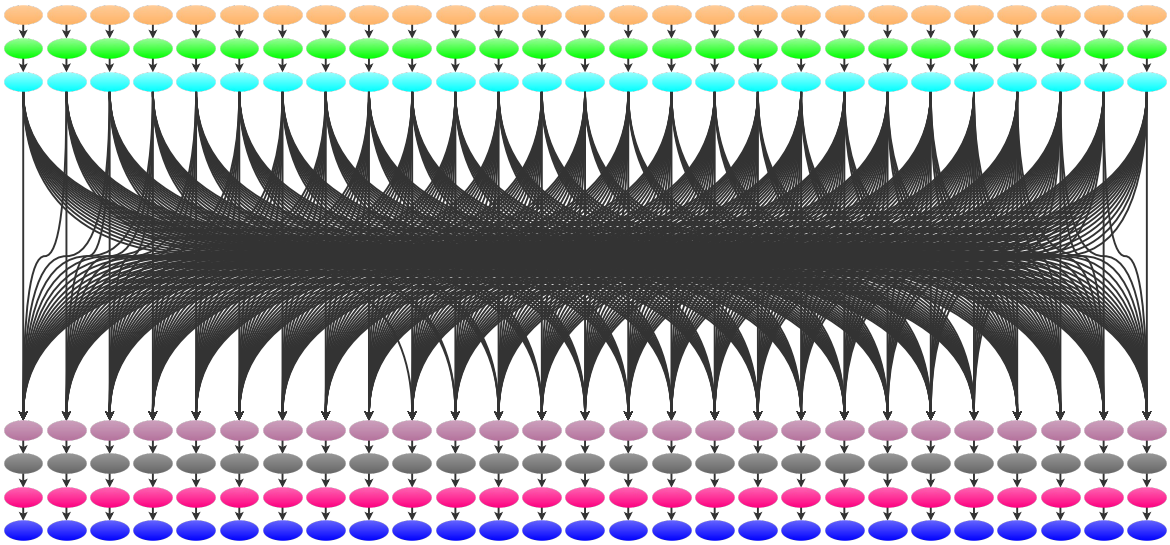
Na subseção 3.4.1 é possível observar, a partir das diretivas utilizadas para a geração das duas tarefas de cópia das bordas em cada subdomínio, que as dependências existentes na tarefa de cópia das bordas da implementação baseada em tarefas foram divididas entre as duas tarefas. Enquanto a primeira somente realiza a leitura das informações dos subdomínios vizinhos armazenadas nas respectivas matrizes auxiliares, a segunda tarefa somente armazena as bordas vizinhas na matriz auxiliar do subdomínio para o qual foi gerada. No entanto, além das dependências existentes entre as duas tarefas e as matrizes auxiliares ainda existem as dependências mútuas em relação às matrizes temporárias de cada subdomínio. Dessa forma, é com base nas dependências em relação aos endereços na memória em que estão alocadas ambas as matrizes e na ordem de geração das duas tarefas que o escalonador de tarefas do OpenMP controla a execução dessas e das demais tarefas.

O DAG do escalonamento das sete tarefas da implementação baseada em tarefas com `buffers` em uma execução hipotética do particionamento de um domínio qualquer em 27 subdomínios com `t_max` igual a 1 é apresentado na Figura 12. Esse grafo segue as mesmas premissas do grafo da implementação baseada em tarefas apresentadas na subseção 3.3.2, entre elas, de que existe um número suficientemente grande de *threads* de modo que seja possível executar todas as tarefas paralelamente. Além disso, a ordem de geração das tarefas nesse grafo é idêntica à utilizada no grafo da versão baseada em tarefas.

Código 3.11 – Diretiva para a geração da tarefa de cópia de bordas dos subdomínios vizinhos a partir dos buffers.

```
1  #pragma omp task untied shared(aux, east_buffer,
2                                 north_east_buffer,
3                                 north_buffer,
4                                 north_west_buffer,
5                                 west_buffer,
6                                 south_west_buffer,
7                                 south_buffer,
8                                 south_east_buffer,
9                                 bottom_buffer,
10                                top_buffer,
11                                east_bottom_buffer,
12                                east_top_buffer,
13                                north_bottom_buffer,
14                                north_top_buffer,
15                                west_bottom_buffer,
16                                west_top_buffer,
17                                south_bottom_buffer,
18                                south_top_buffer)
19                         depend(in : east_buffer[ : 1],
20                                     west_buffer[ : 1],
21                                     north_buffer[ : 1],
22                                     south_buffer[ : 1],
23                                     top_buffer[ : 1],
24                                     bottom_buffer[ : 1],
25                                     north_west_buffer[ : 1],
26                                     south_west_buffer[ : 1],
27                                     north_east_buffer[ : 1],
28                                     south_east_buffer[ : 1],
29                                     west_bottom_buffer[ : 1],
30                                     west_top_buffer[ : 1],
31                                     east_bottom_buffer[ : 1],
32                                     east_top_buffer[ : 1],
33                                     north_bottom_buffer[ : 1],
34                                     north_top_buffer[ : 1],
35                                     south_bottom_buffer[ : 1],
36                                     south_top_buffer[ : 1])
37                         depend(out : aux[ : 1])
```

Figura 12 – Gráfico de dependências da implementação OpenMP Tarefas com Buffers.



Fonte: O autor.

Como pode ser observado no grafo, tanto as tarefas de inicialização (elipses da cor laranja) quanto as tarefas de redistribuição (elipses da cor verde) e de propagação (elipses da cor ciano) são executadas paralelamente nos 27 subdomínios. Assim como na implementação baseada em tarefas, o fato dessas três tarefas dependerem da matriz principal do subdomínio para o qual foram geradas permite que ambas sejam executadas paralelamente em diferentes subdomínios (representado no grafo por sequências horizontais de elipses com as respectivas cores das tarefas). No entanto, uma vez que a tarefa de redistribuição depende dos resultados produzidos pela execução da tarefa de inicialização e a tarefa de propagação, por sua vez, depende dos resultados da execução da tarefa de redistribuição em um mesmo subdomínio, a execução dessas três tarefas somente pode ser realizada de forma sequencial (representado no grafo por sequências verticais de elipses com arestas que simbolizam as dependências entre as tarefas). A sequencialização da execução dessas tarefas se deve ao tipo de dependência existente entre as tarefas e o endereço na memória em que a matriz principal de cada subdomínio está alocada que, conforme pôde ser visto na subseção 3.3.1, é `out`, `inout` e `in`, respectivamente.

Finalizadas as execuções das tarefas de propagação, a próxima tarefa executada pelas *threads* do time em cada subdomínio é a tarefa de cópia das bordas para matrizes temporárias. Apesar dessa tarefa ser gerada após a tarefa de propagação em cada subdomínio pela *thread* principal, o que impede essa tarefa de ser executada antes da finalização da tarefa de propagação, assim como na implementação baseada em tarefas, são as dependências existentes entre a tarefa e os endereços na memória em que as matrizes auxiliares são armazenadas. Uma vez que a tarefa de cópia das bordas para matrizes temporárias possui uma dependência do tipo `in` em relação às matrizes auxiliares e a

tarefa de propagação possui uma dependência do tipo `out` a execução dessa tarefa somente pode ser realizada após a finalização da execução das tarefas de propagação tanto no subdomínio para o qual a tarefa foi gerada quanto nos dezoito subdomínios vizinhos dos quais a tarefa depende.

Como pode ser observado no grafo, apesar da tarefa de cópia das bordas para matrizes temporárias depender dos resultados da execução tanto da tarefa de propagação do subdomínio para o qual a tarefa foi gerada quanto das tarefas de propagação dos dezoito subdomínios vizinhos ao seu subdomínio, essa tarefa é executada paralelamente em todos os subdomínios. Isso se deve pelo fato dessa tarefa armazenar as informações obtidas a partir da leitura da matriz auxiliar em matrizes temporárias pertencentes ao seu subdomínio e que não são dependências das demais tarefas de cópia das bordas para as matrizes temporárias dos subdomínios vizinhos. Uma vez que essa tarefa não modifica as informações contidas na sua matriz auxiliar, da forma como a tarefa de cópia das bordas na implementação baseada em tarefas modifica, todas as tarefas podem ler as informações paralelamente sem que haja inconsistências durante as execuções. A execução paralela dessa tarefa em todos os subdomínios é representada no grafo por uma sequência horizontal de elipses da cor lilás com 19 arestas cada simbolizando as dependências existentes entre as tarefas de cópia das bordas para matrizes temporárias e as tarefas de propagação.

A tarefa de cópia das bordas das matrizes temporárias para a matriz auxiliar, por sua vez, somente é executada após a finalização da execução da tarefa de cópia das bordas dos subdomínios vizinhos para as matrizes temporárias. O fato dessa tarefa possuir uma dependência do tipo `in` e ser gerada após a tarefa de cópia das bordas para matrizes temporárias que possui uma dependência do tipo `out` em relação às matrizes temporárias de um mesmo subdomínio impede que a execução da tarefa seja realizada antes da finalização ou durante a execução da tarefa de cópia das bordas para matrizes temporárias. Além disso, ao possuir uma dependência do tipo `out` enquanto a tarefa de cópia das bordas para matrizes temporárias gerada anteriormente possui uma dependência do tipo `in` em relação à matriz auxiliar de um mesmo subdomínio também é adicionada uma restrição que impede que a tarefa seja executada antes ou durante a execução da tarefa de cópia das bordas para matrizes temporárias. Dessa forma, ao depender apenas das matrizes temporárias e da matriz auxiliar do subdomínio para o qual a tarefa é gerada a execução em todos os subdomínios pode ser realizada de forma paralela. No grafo a execução da tarefa de cópia das bordas a partir das matrizes temporárias para a matriz principal em cada subdomínio é representada por uma sequência horizontal de elipses da cor cinza com arestas que indicam a existência da dependência em relação à finalização da execução da tarefa de cópia das bordas para matrizes temporárias.

Por fim, assim como na implementação baseada em tarefas, as tarefas de bounceback e relaxação são executadas de forma paralela em diferentes subdomínios. Essas duas tarefas

são geradas nessa ordem após a geração da tarefa de cópia das bordas a partir das matrizes temporárias para as matrizes temporárias e a ordem de execução é definida somente pelas suas dependências. Uma vez que a tarefa de bounceback depende dos resultados da execução da tarefa de cópia das bordas a partir das matrizes temporárias para a matriz auxiliar e, portanto, possui uma dependência do tipo `in` em relação à matriz auxiliar a execução dessa tarefa somente é realizada após a finalização da execução dessa tarefa. Da mesma forma, ao depender dos resultados da tarefa de bounceback armazenados na matriz principal de cada subdomínio e, portanto, possuir uma dependência do tipo `in` em relação à matriz auxiliar sobre a qual a tarefa de bounceback possui uma dependência do tipo `out` essa somente pode ser executada após a finalização da execução da tarefa de bounceback.

3.5 OpenMP For

Tanto na implementação MPI quanto nas implementações baseadas em tarefas o paralelismo na execução do método depende diretamente do particionamento do reticulado em partes menores denominadas subdomínios. Os subdomínios resultantes do particionamento são, então, processados paralelamente por um determinado número de processos remotos na implementação MPI ou por um determinado número de tarefas nas implementações OpenMP baseadas em tarefas. Apesar do particionamento possibilitar a execução simultânea do método em diferentes subdomínios, pelo fato de um subdomínio ser apenas uma parte de todo o reticulado, partículas localizadas nas extremidades de um subdomínio podem deslocar-se para posições no reticulado pertencentes a outros subdomínios tornando necessária a sincronização das extremidades de cada subdomínio com os seus subdomínios vizinhos, de acordo com as direções de propagação existentes no modelo D3Q19. Dessa forma, com o intuito de avaliar o desempenho das implementações baseadas em tarefas e particionamento de domínio desenvolveu-se uma implementação OpenMP baseada no paralelismo de iterações de laços de repetição do tipo `for` que dispensa o particionamento de domínio e, consequentemente, a sincronização das extremidades dos subdomínios denominada OpenMP `for`.

Essa implementação possui as mesmas funções existentes nas implementações MPI e baseadas em tarefas OpenMP com exceção, é claro, da função de sincronização das extremidades dos subdomínios. Cada função é constituída basicamente por laços de repetição do tipo `for` da linguagem de programação C triplamente aninhados, ou seja, existem três laços `for` localizados um no interior do outro. Esses três laços correspondem às três dimensões do modelo e são utilizados para acessar cada posição das matrizes principais e auxiliares dos subdomínios durante a execução do método. Portanto, são as iterações desses três laços de repetição presentes em cada uma das cinco funções que serão paralelizadas na implementação OpenMP for por meio da diretiva `parallel for`.

Código 3.12 – Visão geral da implementação baseada no paralelismo de iterações de laços de repetição.

```
1 #pragma omp parallel for
2 initialize();
3 for (time = 0; time < t_max; time++) {
4 #pragma omp parallel for
5   redistribute();
6 #pragma omp parallel for
7   propagate();
8 #pragma omp parallel for
9   bounceback();
10 #pragma omp parallel for
11   relaxation();
12 }
```

No Código 3.12 é apresentada uma representação da paralelização das iterações dos laços de repetição presentes em cada função realizada na implementação OpenMP for. Como é possível observar, a estrutura da execução das funções que compõe o método nessa implementação é similar à estrutura presente nas implementações MPI e baseadas em tarefas OpenMP, no entanto, sem a presença da função de sincronização. Além disso, é possível observar que em cada função é ilustrada a utilização da diretiva OpenMP `parallel for` de modo que as iterações dos três laços de repetição sejam distribuídos entre um time de *threads* de modo que cada *thread* execute determinada iteração ou intervalo de iterações. Dessa forma, cada *thread* aplica as operações de cada função somente nas posições das matrizes correspondentes as suas iterações de forma paralela.

3.6 Considerações do Capítulo

Como pôde ser observado, a transcrição da implementação MPI para uma implementação baseada em tarefas consistiu na transformação das funções executadas por cada processo no seu subdomínio em tarefas geradas por uma *thread* principal para cada subdomínio resultante do particionamento. Para que as tarefas sejam executadas na ordem em que são geradas em um mesmo subdomínio mas paralelamente nos demais subdomínios definiu-se as dependências de dados de cada tarefa e a partir de um cenário hipotético de execução foi possível compreender com maior clareza o escalonamento e execução de cada tarefa de acordo com a ordem de geração e as dependências de dados. Além disso, apresentou-se uma implementação otimizada da implementação baseada em tarefas proposta inicialmente na qual foi possível observar, no cenário hipotético de execução apresentado, que as estruturas temporárias adicionadas para a cópia das bordas dos subdomínios vizinhos aumentam o paralelismo na execução das tarefas, especialmente de cópia das bordas. Por fim, descreveu-se ainda a implementação baseada no paralelismo de

iterações de laços de iterações.

4 Resultados Experimentais

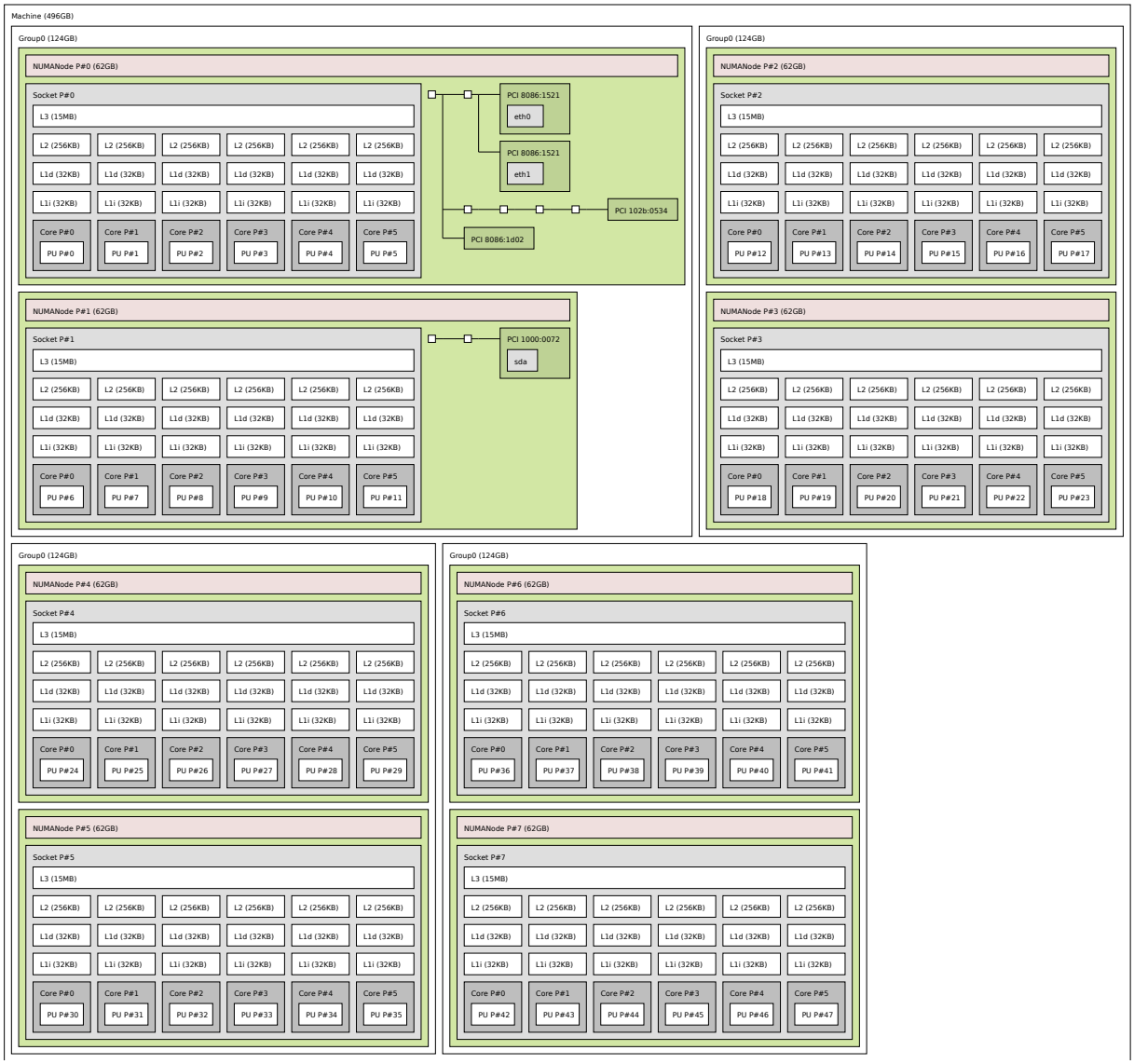
Neste trabalho apresentar-se-á os resultados dos experimentos realizados em um sistema de memória compartilhada composto por 8 nós NUMA e um total de 48 núcleos de processamento. Serão apresentados os resultados dos experimentos realizados variando o tamanho do reticulado utilizado nas simulações do método, a quantidade de núcleos de processamento e a quantidade de subdomínios em que cada dimensão dos reticulados foi particionado. Além disso apresentar-se-á alguns experimentos realizados para compreender a diferença notada no tempo de execução de ambas as implementações quando utilizados dois compiladores distintos para a linguagem C, a diferença no tempo de execução quando utilizada a intercalação na alocação de memória, bem como os resultados de alguns experimentos realizados para avaliar o desempenho das diferentes técnicas de otimização do desempenho utilizadas nas implementações. Por fim, analisar-se-á o desempenho obtido por ambas as implementações nos diferentes experimentos realizados.

4.1 Ambiente de Teste e Plataformas de Experimentação

Os experimentos foram realizados em um sistema de memória compartilhada NUMA denominado SGI UV2000. Esse sistema é composto por um total de 8 nós NUMA. Em cada nó existe um chip de processador conectado cujo modelo é Xeon SandyBridge EN/EP E5-4617 da fabricante de circuitos integrados Intel. Os processadores possuem 6 núcleos de processamento físicos e exiguidade da tecnologia de virtualização de *threads* denominada *hyper-threading*, totalizando 48 núcleos de processamento. A frequência base desses processadores é 2.90GHz e pode ser ampliada para até 3.40GHz por meio da tecnologia *Turbo Boost*. Cada processador possui 64GB de memória RAM local com interface DDR3, totalizando 512GB de memória RAM global. O sistema operacional em execução era um Debian GNU/Linux 9 (stretch) com *kernel* versão 4.9.0-1-amd64. Na Figura 13 é apresentado a organização de cada nó NUMA do sistema.

Para avaliar o desempenho das implementações OpenMP propostas neste trabalho realizou-se experimentos variando a quantidade de *threads* utilizadas nesse sistema, o tamanho dos reticulados utilizados nas simulações do método, a quantidade de subdomínios nos quais esses reticulados foram particionados e o compilador para linguagem C utilizado na compilação das implementações. A variação da quantidade de *threads* utilizada nos experimentos foi realizada por meio da variável de ambiente `OMP_PLACES` com base nos nós NUMA do sistema, correspondendo à 12, 24, 36 e 48 *threads*, ou seja, à utilização de 2, 4, 6 e 8 nós NUMA, respectivamente. Em cada uma dessas variações de *threads* realizou-se experimentos com reticulados de tamanho 32, 64, 96, 128, 192, 256, 512 e 1024 e ambos

Figura 13 – Arquitetura do sistema SGI UV2000.



Fonte: O autor.

particionados em cada uma de suas 3 dimensões em 2, 4, 8, 16 e 32 subdomínios, ou seja, particionados em um total de 8, 64, 512, 4096 e 32768 subdomínios, respectivamente. Apesar da utilização somente desses oito tamanhos de reticulados nos experimentos realizados neste trabalho devido ao grande número de combinações de experimentos necessários, qualquer outro tamanho pode ser utilizado nas simulações do LBM em ambas as implementações.

Os experimentos foram repetidos no mínimo 10 vezes para cada combinação de variações de *threads*, reticulados e particionamentos experimentada em cada uma das implementações. As combinações de reticulados e particionamentos experimentadas variaram de acordo com o tamanho do reticulado utilizado, visto que determinados particionamentos são inviáveis devido ao alto custo computacional, e de acordo com a

implementação utilizada, uma vez que o desempenho de cada implementação para uma mesma combinação de tamanho do reticulado e particionamento difere significativamente. Em um reticulado de tamanho 32, por exemplo, a adoção de um particionamento de cada dimensão em 32 subdomínios é inviável computacionalmente pois o tempo de execução em relação às demais variações de particionamento é expressivamente maior uma vez que nesse particionamento cada subdomínio é composto por uma única posição da matriz. Todos os experimentos foram realizados com um número de iterações `t_max` igual à 500.

As implementações OpenMP do método foram compiladas a partir de dois compiladores distintos para a linguagem C: o compilador Clang versão 4.0 juntamente com a *runtime* OpenMP libKOMP (BROQUEDIS; GAUTIER; DANJEAN, 2012), uma extensão da *runtime* OpenMP libomp presente em compiladores Intel e Clang que inclui algumas características como roubo de trabalho, escalonamento de tarefas com afinidade em arquiteturas NUMA e escalonamento de laços de repetição ausentes na *runtime* libomp; e o compilador GCC versão 7.2. Devido à expressiva variação de desempenho apresentada na execução das implementações compiladas com ambos os compiladores utilizou-se para os experimentos deste trabalho somente o compilador que proporcionou o melhor desempenho para cada implementação. Dessa forma, os experimentos com a implementação OpenMP for do método foram realizados a partir da compilação dessa implementação utilizando o compilador GCC e com as demais implementações OpenMP baseadas em tarefas a partir da compilação dessas implementações utilizando o compilador Clang.

Além disso, por meio do comando `numactl` (disponível na grande maioria das distribuições Linux) determinou-se a política de alocação de memória adotada em todos os experimentos. Mediante a opção `--interleave` do comando `numactl` estabeleceu-se uma política de alocação de memória intercalada que segue uma distribuição do tipo *round robin* apenas entre os nós NUMA empregados nas execuções. Dessa forma, ao invés da alocação da memória ser realizada somente no nó NUMA em que um processo se origina a alocação é realizada de forma intercalada entre todos os nós envolvidos, o que reduz o tempo de acesso à memória nos diferentes nós NUMA uma vez que as distâncias entre as memórias de cada nó variam.

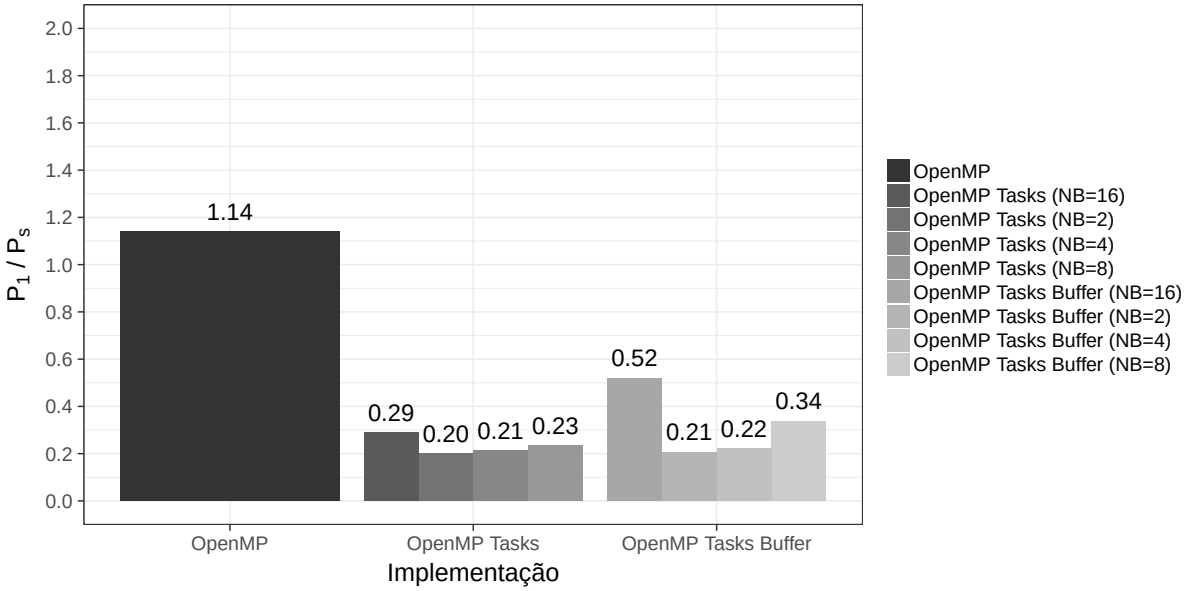
Em todos os experimentos realizados no sistema SGI UV2000 utilizou-se a ferramenta de monitoramento de desempenho Likwid (TREIBIG; HAGER; WELLEIN, 2010) e determinou-se como o esquema de energia (governador) para todos os processadores a execução em frequência máxima, ou seja, o governador `performance`. A ferramenta Likwid foi utilizada no monitoramento do desempenho das implementações durante os experimentos, especialmente para registrar a quantidade de ciclos por instrução necessária para executar cada implementação e a quantidade de memória transferida pelas mesmas. Já o esquema de energia foi determinado de modo a utilizar o máximo da capacidade de processamento de cada núcleo dos processadores por meio da ferramenta `cpupower`,

disponível na maioria das distribuições Linux.

4.2 Sobrecarga Paralela

Nesse experimento avaliamos a sobrecarga paralela das implementações OpenMP a partir do tempo de execução das implementações paralelas em 1 *thread* (T_1) sobre o tempo de execução sequencial (T_s). Na Figura 14 são apresentados os resultados experimentais da sobrecarga paralela T_1/T_s das implementações OpenMP em um reticulado de tamanho 128 particionado em 2, 4, 8 e 16 (NB) subdomínios nas versões baseadas em tarefas.

Figura 14 – Sobrecarga paralela T_1/T_s das implementações paralelas OpenMP.



Fonte: O autor.

Aparentemente ambas as implementações baseadas em tarefas OpenMP podem introduzir significativos ganhos de desempenho. Tanto a implementação baseada em tarefas OpenMP quanto a implementação baseada em tarefas OpenMP com `buffers` com particionamento do reticulado em 2 subdomínios por dimensão tiveram a menor sobrecarga em relação à implementação sequencial, respectivamente 0,20 e 0,21. Ou seja, particionar o reticulado em um total de 8 subdomínios e processá-los utilizando tarefas OpenMP com dependências introduz, respectivamente, um ganho de desempenho de 79,93 e 79,32% em ambas as implementações em comparação com o tempo sequencial.

As demais variações do particionamento do reticulado tiveram uma sobrecarga maior em ambas as implementações baseadas em tarefas OpenMP. Em relação ao tempo de execução sequencial, o particionamento de cada dimensão do reticulado em 4, 8 e 16 subdomínios introduz uma sobrecarga de 0,21, 0,23 e 0,29 na implementação baseada em

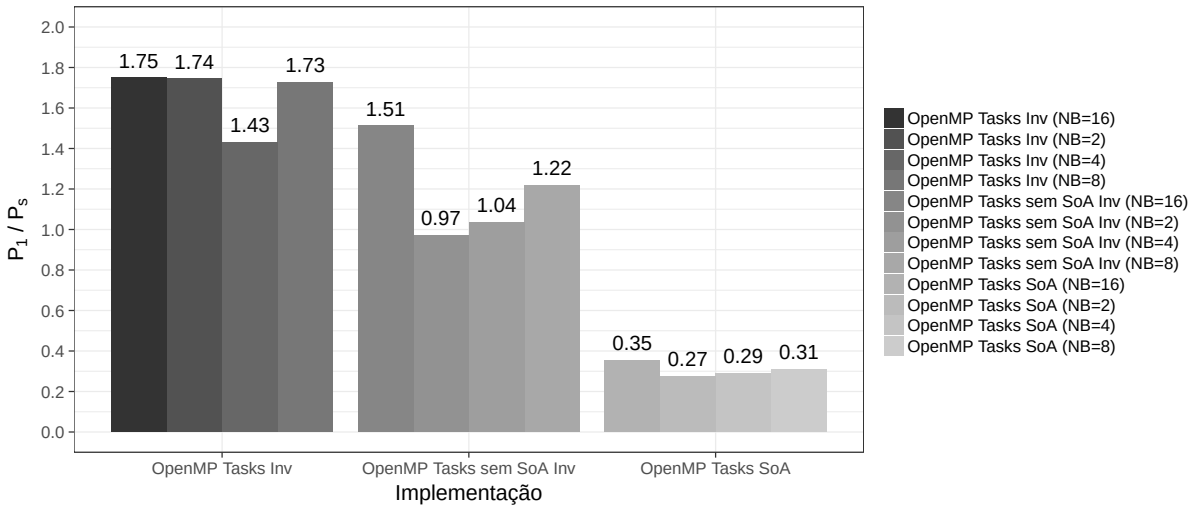
tarefas OpenMP e de 0,22, 0,34 e 0,52 na implementação baseada em tarefas OpenMP com `buffers`. Ou seja, particionar o reticulado em um total de 64, 512 e 4096 subdomínios e processá-los utilizando tarefas OpenMP com dependências introduz, respectivamente, um ganho de desempenho de 78,67, 76,65 e 71,17% na implementação baseada em tarefas OpenMP e de 77,77, 66,27 e 47,97% na implementação baseada em tarefas OpenMP com `buffers` em comparação com o tempo sequencial. A implementação OpenMP for, por sua vez, teve uma sobrecarga de 1,14, que representa 14,02% em relação ao tempo sequencial e é maior que a sobrecarga introduzida pelas implementações OpenMP baseadas em tarefas em todas as variações do particionamento do reticulado.

Como mencionado na seção 3.3 e na seção 3.4, tanto na implementação baseada em tarefas OpenMP quanto na implementação baseada em tarefas OpenMP com `buffers` utilizou-se um padrão de acesso à memória denominado estrutura de matriz (*Structure of Array* – SoA), no qual os dados de cada subdomínio são armazenados sequencialmente em uma mesma matriz unidimensional, bem como a inversão da ordem de execução dos laços que percorrem as três dimensões de cada reticulado. Ao analisar a sobrecarga paralela introduzida na implementação baseada em tarefas OpenMP utilizando somente a inversão dos laços de repetição, somente o padrão de acesso à memória SoA e sem a utilização de nenhuma dessas duas técnicas apresentada na Figura 15, é possível observar que o padrão SoA de acesso à memória introduz a menor sobrecarga paralela. Comparando a sobrecarga utilizando apenas SoA e utilizando SoA em conjunto da inversão da ordem de execução dos laços de repetição apresentada na Figura 14 é possível observar que há uma redução na sobrecarga introduzida utilizando ambas as técnicas. Isso se deve provavelmente ao fato do padrão de acesso SoA otimizar o uso da memória *cache* e, dessa forma, proporcionar um ganho de desempenho significativo em relação ao tempo de execução sequencial.

4.3 Avaliação de Desempenho

Conforme mencionado na seção 4.1, cada tamanho de reticulado foi particionado em 2, 4, 8, 16 e 32 subdomínios por dimensão, de acordo com a viabilidade em termos de tempo computacional, e experimentado em 12, 24, 36 e 48 *threads* (respectivamente 2, 4, 6 e 8 nós NUMA) utilizando uma estratégia de alocação de memória intercalada entre os nós NUMA envolvidos no experimento. Por meio do ambiente de *software* para computação estatística R (R Core Team, 2018), extraiu-se os experimentos que obtiveram os menores tempos de execução em cada uma das variações de *threads* dentre as diferentes variações de particionamento para cada tamanho de reticulado. Além disso, extraiu-se ainda os experimentos que obtiveram os menores tempos de execução dentre as diferentes variações de *threads* e variações de particionamentos para cada tamanho de reticulado. Ou seja, para cada tamanho de reticulado extraiu-se os resultados das 10 repetições dos experimentos cujo particionamento resultou no menor tempo de execução em cada uma das variação

Figura 15 – Sobrecarga paralela T_1/T_s da implementação baseada em tarefas OpenMP (sem `buffers`) utilizando apenas inversão da ordem dos laços de repetição (Inv), apenas matrizes 1D (SoA) e sem inversão de laços e com matrizes 3D (sem SoA Inv). O tamanho do reticulado utilizado nos experimentos foi de 128.



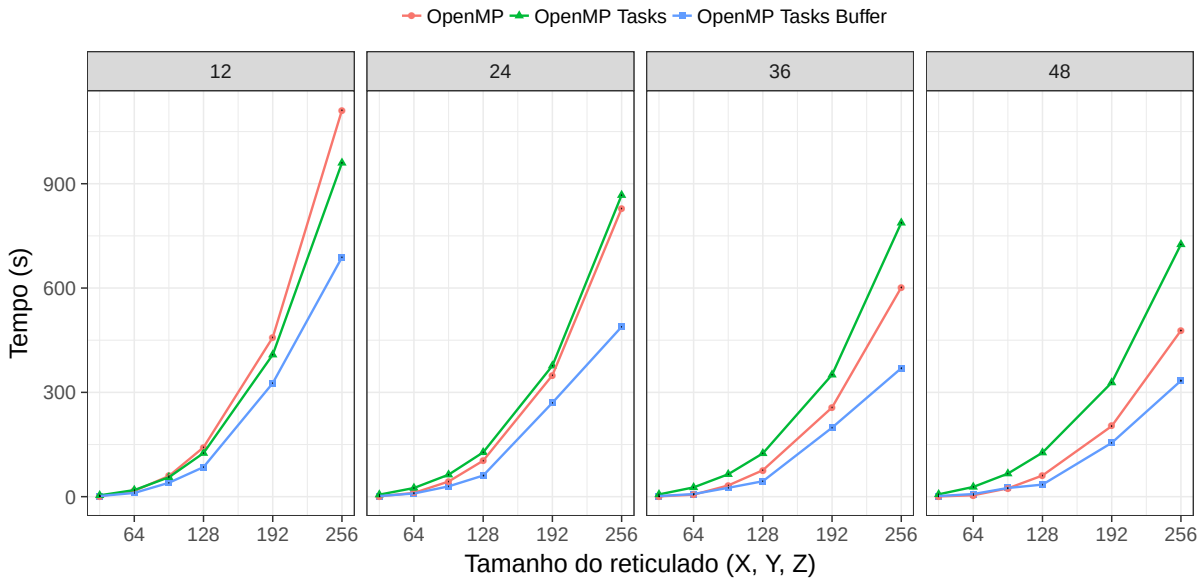
Fonte: O autor.

de *threads* ou dentre as diferentes variações de *threads*. A partir desses experimentos elaborou-se um conjunto de gráficos nos quais são apresentados os valores médios das 10 repetições desses experimentos e que serão avaliados em detalhes nesta seção.

No gráfico da Figura 16 são apresentados os melhores tempos de execução obtidos a partir da execução das implementações baseada no paralelismo de laços de repetição, baseada em tarefas e baseada em tarefas com `buffers` nos reticulados de tamanho 32, 64, 96, 128, 192 e 256 utilizando 12, 24, 36 e 48 *threads*. Como é possível observar, a diferença entre os tempos de execução de cada implementação se torna mais evidente a medida que o tamanho do reticulado aumenta, especialmente quando utilizadas 36 e 48 *threads*. Além disso, em ambas as combinações de *threads* o tempo de execução da implementação OpenMP baseada em tarefas com `buffers` a partir do reticulado de tamanho 128 é significativamente menor do que o tempo de execução das outras duas implementações.

No caso da utilização de 48 *threads* e reticulado de tamanho 256, o tempo de execução da implementação OpenMP baseada em tarefas com `buffers` foi significativamente menor do que o tempo de execução das outras duas implementações. Enquanto o tempo de execução dessa implementação foi de 333,51 segundos o tempo de execução das implementações OpenMP for e baseada em tarefas foi de 477,82 e 725,47 segundos, respectivamente. Isto é, o tempo de execução das implementações OpenMP for e baseada em tarefas foi aproximadamente 1,43 e 2,17 vezes o tempo de execução da implementação

Figura 16 – Tempo de execução das implementações OpenMP nos particionamentos que obtiveram os melhores tempos de execução em cada tamanho de reticulado em 12, 24, 36 e 48 threads.



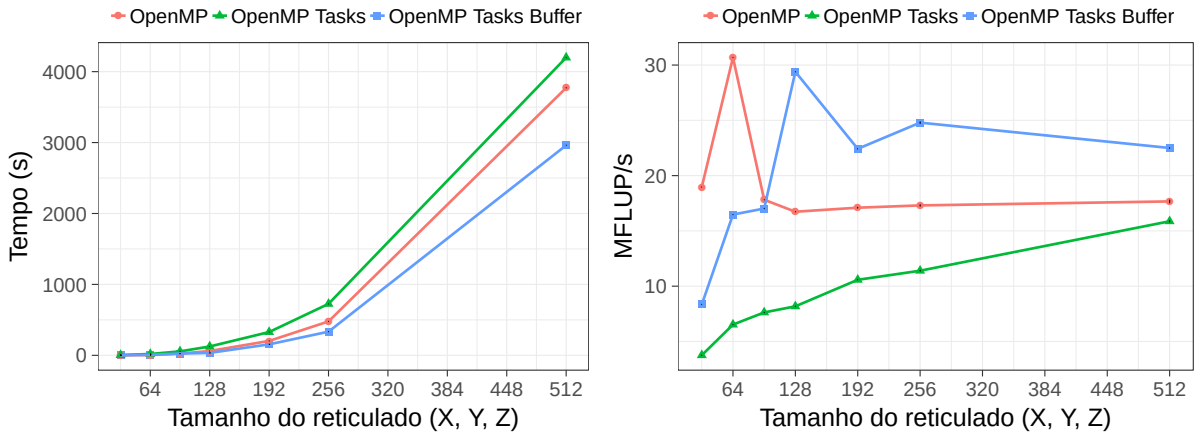
Fonte: O autor.

OpenMP baseada em tarefas com `buffers`. O tempo de execução da implementação OpenMP baseada em tarefas somente não é maior do que o tempo de execução das outras duas implementações nos experimentos realizados apenas com 12 *threads*, no qual é menor do que o tempo de execução da implementação OpenMP for.

Na Figura 17 são apresentados os melhores tempos de execução obtidos nos reticulados de tamanho 32, 64, 96, 128, 192, 256 e 512 dentre todas as combinações experimentadas de variação da quantidade de *threads* e de variação do particionamento do reticulado em ambas as implementações OpenMP. No reticulado de tamanho 512 o tempo de execução das implementações OpenMP for, baseada em tarefas e baseada em tarefas com `buffers` foi de 3771,35, 4198,44 e 2961,13 segundos, respectivamente. Apesar da diferença entre os tempos de execução de cada implementação ser relativamente menor no reticulado de tamanho 512 em comparação com a diferença no reticulado de tamanho 256, o tempo de execução das implementações OpenMP for e baseada em tarefas foi igual à 1,27 e 1,41 vezes o tempo de execução da implementação OpenMP baseada em tarefas com `buffers`.

Além do tempo de execução das implementações OpenMP, na Figura 17 também é apresentado o desempenho de cada implementação OpenMP em termos de milhões de atualizações de treliça de fluidos por segundo (*Million Fluid Lattice Updates Per Second* – MFLUP/s). Comparando os melhores tempos de execução de cada implementação nos diferentes tamanhos de reticulado com os respectivos MFLUP/s obtidos é possível observar

Figura 17 – Melhores tempos de execução e respectivos MFLUP/s obtidos dentre as variações de threads e particionamentos de domínio utilizados nas implementações OpenMP for, baseada em tarefas e baseada em tarefas com buffers.



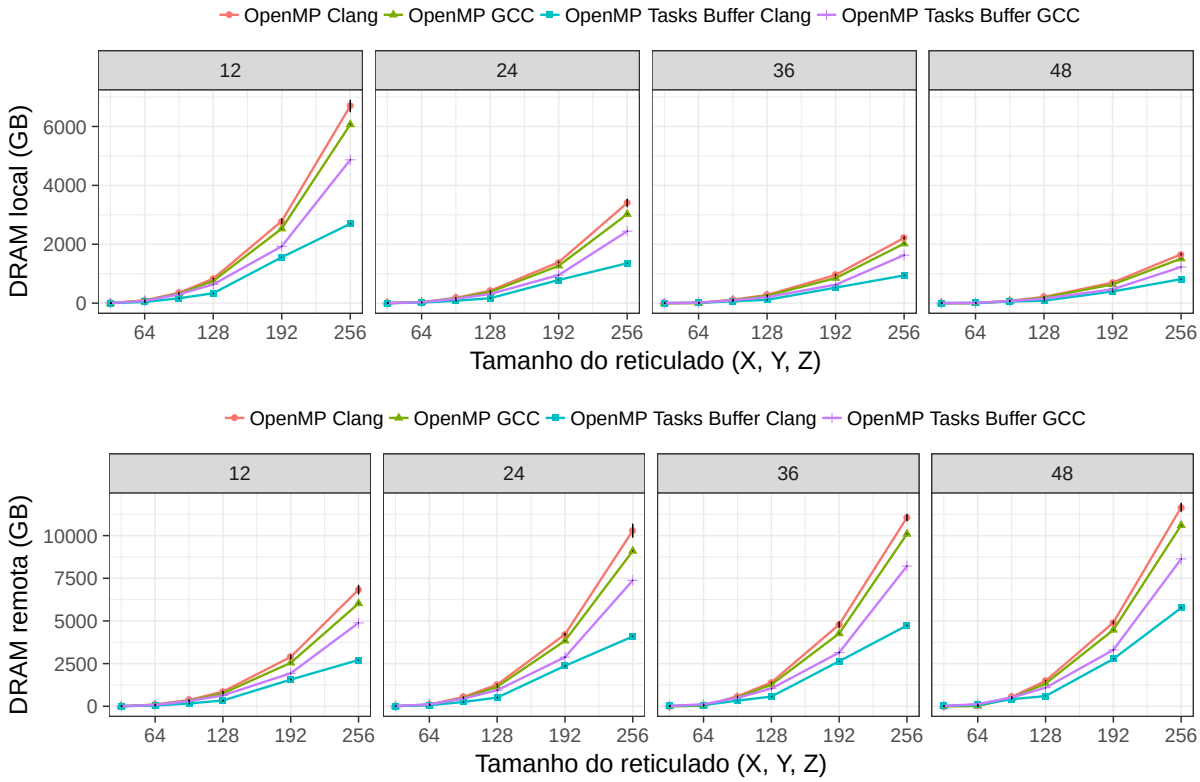
Fonte: O autor.

que nesse caso o tempo de execução e MFLUP/s são inversamente proporcionais. Como pôde ser observado na Figura 16, a partir do reticulado de tamanho 128 o tempo de execução da implementação OpenMP baseada em tarefas com `buffers` se torna significativamente menor em comparação ao tempo de execução das outras duas implementações a medida que o tamanho dos reticulados aumenta, fato esse refletido nos valores de MFLUP/s obtidos pela implementação nos reticulados de tamanho maior e igual à 128 uma vez que são expressivamente maiores que os obtidos pelas outras duas implementações nesses mesmos tamanhos de reticulados. Também é possível notar a relação entre a diferença nos tempos de execução e valores de MFLUP/s obtidos pelas implementações OpenMP, que em ambos os casos diminui do reticulado de tamanho 256 para o de tamanho 512, conforme previamente mencionado nessa seção.

Na Figura 18 é apresentada a quantidade de memória DRAM local e remota transferida durante a execução de cada uma das implementações. Esses valores são relativos à média da quantidade de memória transferida nos experimentos em que se obteve os melhores tempos de execução dentre as diferentes variações de particionamento experimentadas em cada tamanho de reticulado. Ou seja, é a média da quantidade de memória transferida nas 10 repetições do experimento cujo particionamento resultou nos menores tempos de execução em cada tamanho de reticulado. A medida que a quantidade de nós NUMA utilizados nos experimentos aumenta, a quantidade de memória local transferida por cada implementação diminui, ao passo que a quantidade de memória remota transferida aumenta. Com exceção dos experimentos que utilizam apenas 12 *threads* (2 nós NUMA), nos quais a quantidade de memória local e remota transferida é praticamente a mesma, em ambas as variações de *threads* a quantidade de memória remota transferida pelas

implementações OpenMP é significativamente maior que a quantidade de memória local.

Figura 18 – Quantidade de memória DRAM local e remota transferida por ambas as implementações OpenMP nos particionamentos que obtiveram os melhores tempos de execução em 12, 24, 36 e 48 threads.

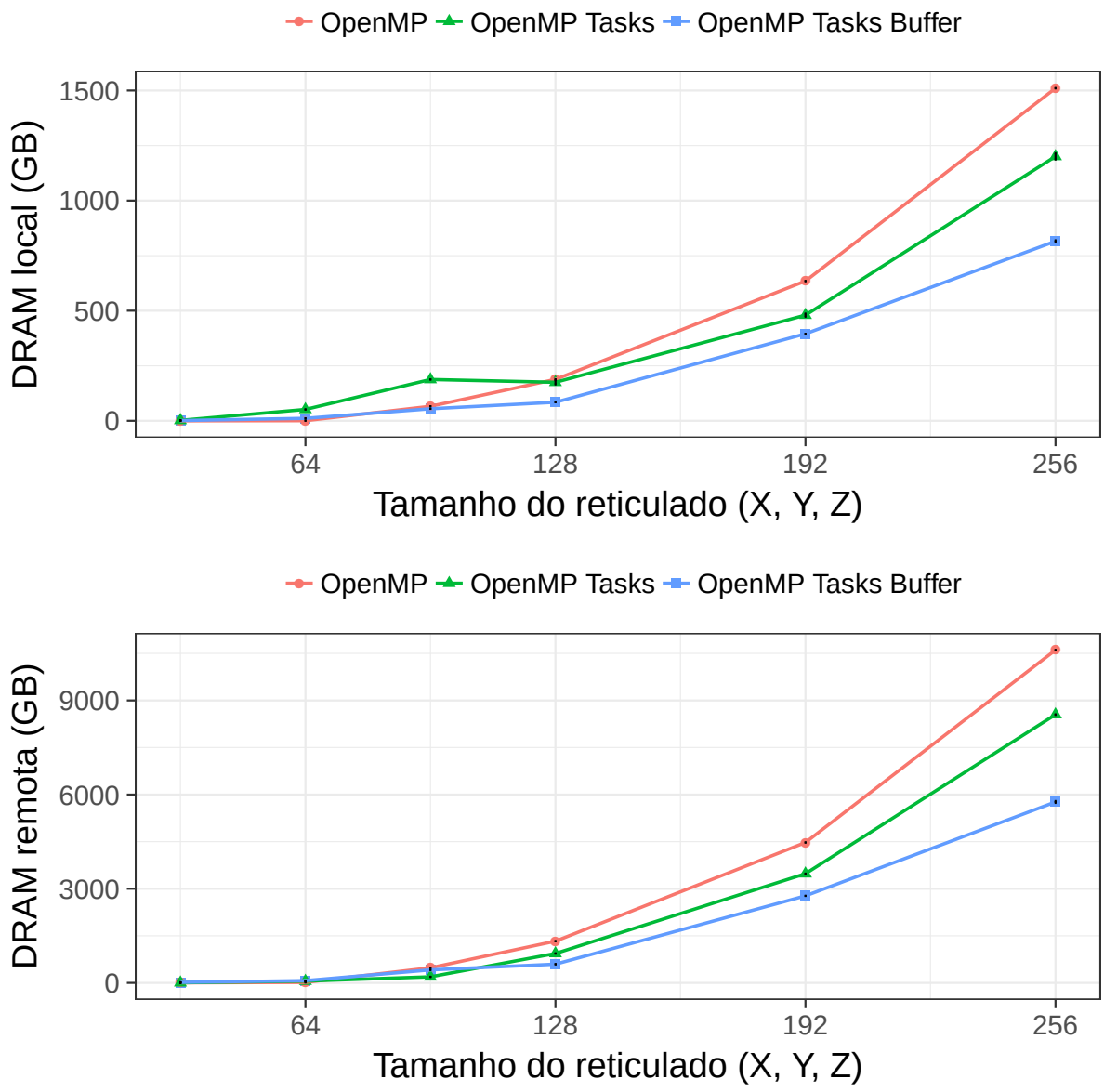


Fonte: O autor.

Ao analisar a quantidade de memória local e remota transferida pelas implementações nos experimentos que obtiveram os melhores tempos de execução dentre as diferentes variações de *threads* e particionamento dos reticulados, apresentados na Figura 19, é possível observar que a implementação OpenMP baseada em tarefas com `buffers` é a que transferiu a menor quantidade de memória local e remota, especialmente nos tamanhos de reticulado 128, 192 e 256. Além disso, nesses tamanhos de reticulado também é visível a transferência significativamente maior tanto da quantidade de memória local quanto da quantidade de memória remota pela implementação OpenMP for. Enquanto a implementação OpenMP baseada em tarefas com `buffers` transferiu 815,41 e 5764,45 GB de memória local e remota no reticulado de tamanho 256, respectivamente, a implementação OpenMP for transferiu 1510,3 e 10605,71 GB de memória local e remota, aproximadamente 1,85 e 1,83 vezes a quantidade transferida pela implementação OpenMP baseada em tarefas com `buffers`.

A quantidade total de memória transferida por cada implementação em 12, 24, 36 e 48 *threads* é apresentado na Figura 20. Conforme o número de *threads* utilizadas nos

Figura 19 – Quantidade de memória DRAM local e remota transferida por ambas as implementações OpenMP nos experimentos que obtiveram os melhores tempos de execução dentre as variações de threads e particionamento dos reticulados.

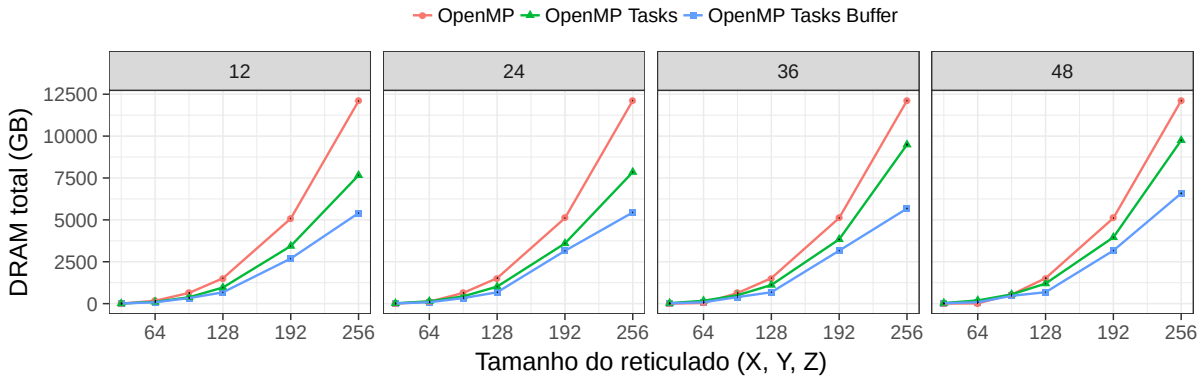


Fonte: O autor.

experimentos aumenta, a quantidade total de memória transferida pela implementação OpenMP for permanece praticamente a mesma. No entanto, nas implementações OpenMP baseadas em tarefas a medida que aumenta o número de *threads* utilizadas nos experimentos a transferência total de memória de ambas também aumenta. Isso se deve à cópia das bordas dos subdomínios vizinhos existente em ambas as implementações baseadas em tarefas que, por sua vez, não existe na implementação baseada no paralelismo de iterações de laços de repetição. A medida que o número de *threads* utilizadas nos experimentos

aumenta, devido a utilização da técnica de alocação de memória intercalada entre os nós NUMA, há uma quantidade menor de subdomínios armazenados localmente em cada nó NUMA que, consequentemente, exige que mais transferências de dados sejam realizadas entre os nós para que esses tenham acesso aos dados das bordas dos subdomínios vizinhos e, assim, possam realizar suas computações.

Figura 20 – Quantidade total de memória DRAM transferida pelas implementações OpenMP nos particionamentos que obtiveram os melhores tempos de execução em 12, 24, 36 e 48 threads.

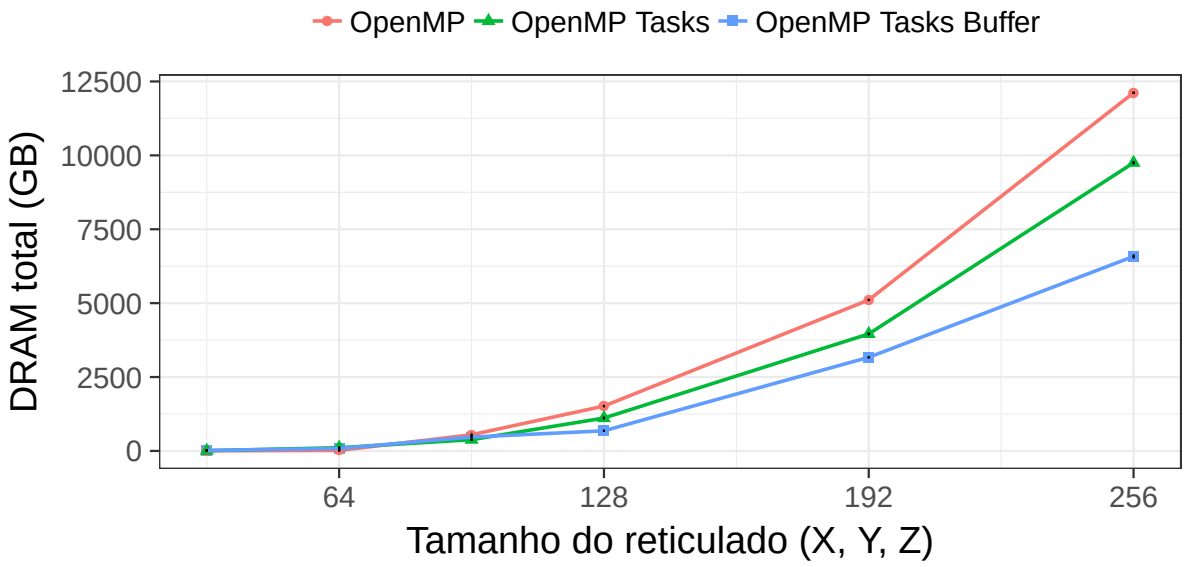


Fonte: O autor.

Na Figura 21 é apresentado a quantidade total de memória transferida pelas implementações OpenMP nos experimentos que obtiveram os melhores tempos de execução dentre as diferentes variações de *threads* e particionamento dos reticulados. Como a quantidade total de memória transferida é a soma da quantidade de memória local e remota transferida, uma vez que a implementação OpenMP baseada em tarefas com `buffers` teve a menor transferência de memória local e também de memória remota (conforme visto na Figura 19) essa teve a menor transferência total de memória. As implementações OpenMP for, baseada em tarefas e baseada em tarefas transferiram um total de 6579,86, 9749,27 e 12116,01 GB de memória, respectivamente.

Para uma última avaliação do desempenho das implementações OpenMP em relação à taxa de transferência de instruções apresenta-se a Figura 22, na qual é exibida a razão da quantidade de ciclos de *clock* dos processadores utilizados pela quantidade de instruções executadas nos experimentos, denominada Ciclos Por Instrução (*Cycles Per Instruction* – CPI). Como pode ser visto, dentre as implementações OpenMP experimentadas a for foi a que utilizou a menor quantidade de ciclos para executar suas instruções em 12, 24 e 36 *threads* aumentando expressivamente em 48 *threads*, especialmente a partir dos tamanhos de reticulado 96. As implementações baseadas em tarefas, por sua vez, tiveram uma utilização de ciclos por instrução maior em 12, 24 e 36 *threads*. Em 48 *threads*, no entanto, o CPI tanto da implementação baseada em tarefas quanto da implementação

Figura 21 – Quantidade total de memória DRAM transferida por ambas as implementações OpenMP nos experimentos que obtiveram os melhores tempos de execução dentre as variações de threads e particionamento dos reticulados.



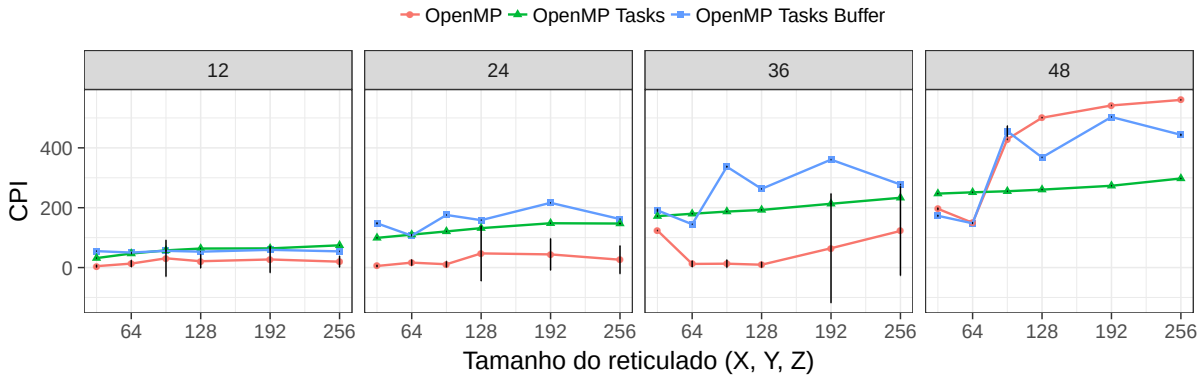
Fonte: O autor.

baseada em tarefas com `buffers` foi menor em tamanhos de reticulado maiores comparado ao CPI da implementação for.

A implementação baseada em tarefas com `buffers` foi a implementação que utilizou uma quantidade relativamente maior de ciclos para executar suas instruções na maioria dos experimentos em comparação às outras duas implementações. Além disso, tanto a implementação for quanto a implementação baseada em tarefas com `buffers` tiveram flutuações nos CPIs para diferentes tamanhos de reticulado nas variações de *threads* 24, 36 e 48, permanecendo relativamente contínuo em 12 *threads*. Somente a implementação baseada em tarefas manteve uma estabilidade relativa nos CPIs para diferentes tamanhos de reticulado, aumentando gradualmente a medida que o tamanho dos reticulados aumenta.

Na Figura 23 é exibida a quantidade de ciclos utilizada por instrução nas implementações OpenMP nos experimentos que obtiveram os melhores tempos de execução dentre as variações de *threads* e particionamento dos reticulados utilizados. Nesse caso a implementação que utilizou a menor quantidade de CPI na execução dos experimentos nos reticulados de tamanho 32, 64, 96, 128, 192 e 256 foi a baseada em tarefas. Já as implementações for e baseada em tarefas com `buffers` utilizaram uma quantidade relativamente maior nesses tamanhos de reticulado. Nos tamanhos 32, 64 e 96 as duas implementações utilizaram uma quantidade similar de CPI e a partir do tamanho 128 a implementação for utilizou uma quantidade relativamente maior de ciclos para executar suas instruções

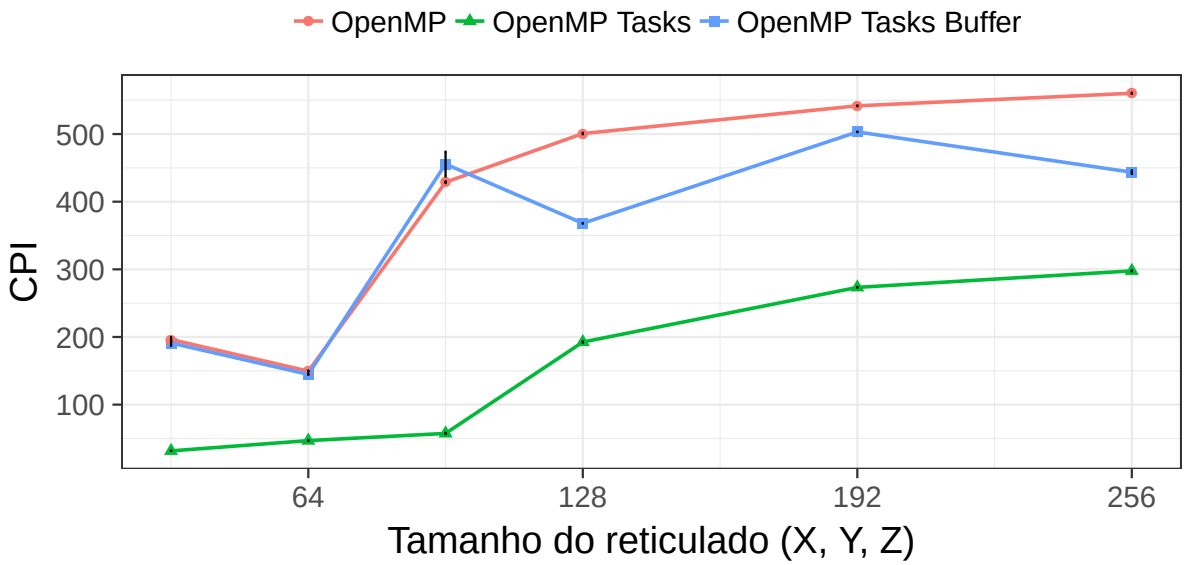
Figura 22 – Ciclos Por Instrução (CPI) utilizados pelas implementações OpenMP nos particionamentos que obtiveram os melhores tempos de execução em 12, 24, 36 e 48 threads.



Fonte: O autor.

comparado com os ciclos utilizados pela implementação baseada em tarefas com `buffers`. Em ambas as implementações os reticulados de tamanho maior exigiram a utilização de uma quantidade maior de ciclos para executar as instruções.

Figura 23 – Ciclos Por Instrução (CPI) utilizados pelas implementações OpenMP nos experimentos que obtiveram os melhores tempos de execução dentre as variações de threads e particionamento dos reticulados.



Fonte: O autor.

4.4 Análise e Discussão dos Resultados

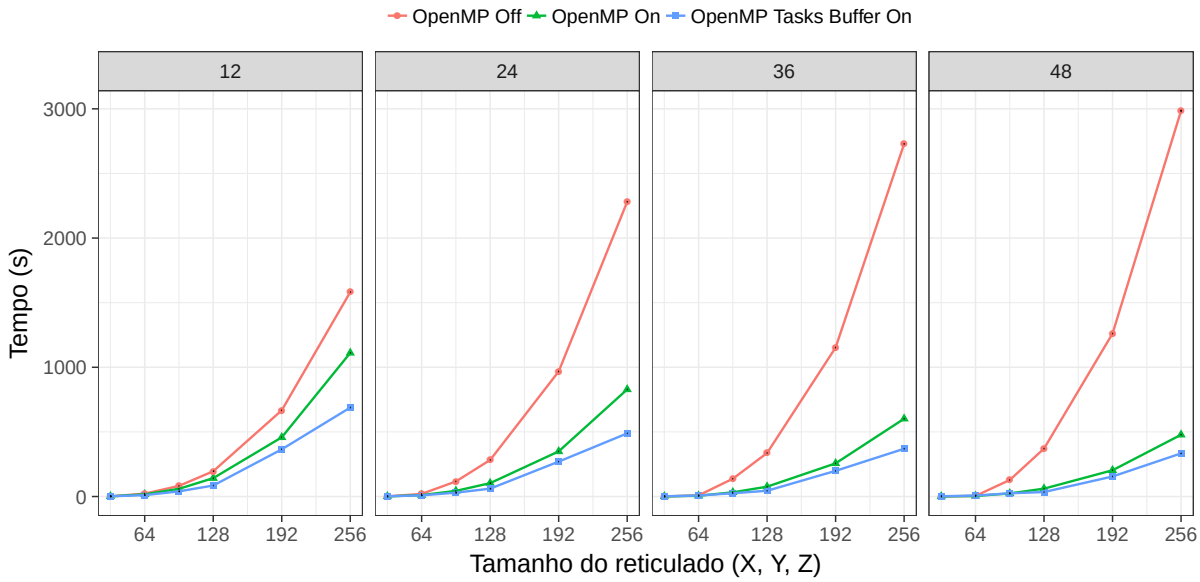
Na seção 4.1 mencionou-se a utilização da estratégia de alocação de memória intercalada entre todos os nós NUMA envolvidos na execução dos experimentos como uma forma de reduzir o tempo de acesso à memória. Essa decisão foi tomada a partir da análise dos resultados de experimentos preliminares nos quais utilizou-se a estratégia de alocação padrão bem como a estratégia de intercalação da alocação da memória entre os nós envolvidos nas execuções dos experimentos. Na Figura 24 são apresentados os tempos de execução obtidos utilizando ambas as estratégias de alocação de memória nos particionamentos mais eficientes para cada tamanho de reticulado. No gráfico da figura a utilização da estratégia é identificada por meio da adição da palavra *On* e a utilização da estratégia padrão é identificada por meio da adição da palavra *Off* aos identificadores das implementações.

Como pode ser observado no gráfico, o tempo de execução da implementação OpenMP for utilizando a técnica de alocação intercalada da memória introduz ganhos expressivos de desempenho comparado ao tempo de execução utilizando a estratégia padrão de alocação. Esse ganho se torna mais visível a medida que a quantidade de *threads* utilizadas nos experimentos e o tamanho dos reticulados aumenta. No tamanho de reticulado 256 e utilizando 48 *threads* o tempo de execução com alocação intercalada é de 477,82 segundos enquanto que o tempo de execução com alocação padrão é de 2984,77 segundos, aproximadamente 6,25 vezes o tempo de execução com intercalação.

Já na implementação baseada em tarefas com `buffers`, apesar da diferença não ser tão significativa, a alocação intercalada de memória também introduz ganhos de desempenho. Enquanto que o tempo de execução no reticulado de tamanho 256 utilizando 48 *threads* é de 333,51 segundos com alocação de memória intercalada, o tempo de execução com a estratégia de alocação de memória padrão é 419,64 segundos, aproximadamente 1,26 vezes o tempo de execução com alocação intercalada. Portanto, a alocação de memória de forma distribuída entre os nós NUMA envolvidos no processamento ao invés da alocação somente no nó NUMA em que o processo se origina reduz o tempo de execução das implementações ao distribuir, consequentemente, o tempo de acesso às alocações despendido por tarefas nos diferentes nós NUMA.

Além disso, na seção 4.1 mencionou-se ainda a utilização do compilador GCC para a compilação da implementação OpenMP for e do compilador Clang em conjunto da *runtime* libKOMP para a compilação das implementações OpenMP baseadas em tarefas. Na Figura 25 são apresentados os tempos de execução das implementações for e baseada em tarefas com `buffer` nos particionamentos dos reticulados em que se obteve o melhor desempenho utilizando os compiladores GCC e Clang, respectivamente. Como pode ser visto no gráfico da figura, na implementação for os melhores tempos de execução foram obtidos a partir da compilação por meio do compilador GCC e na implementação

Figura 24 – Tempo de execução com o melhor particionamento para cada reticulado e com alocação de memória intercalada (On) e padrão (Off) nas implementações OpenMP for e baseada em tarefas com buffers em 12, 24, 36 e 48 threads.



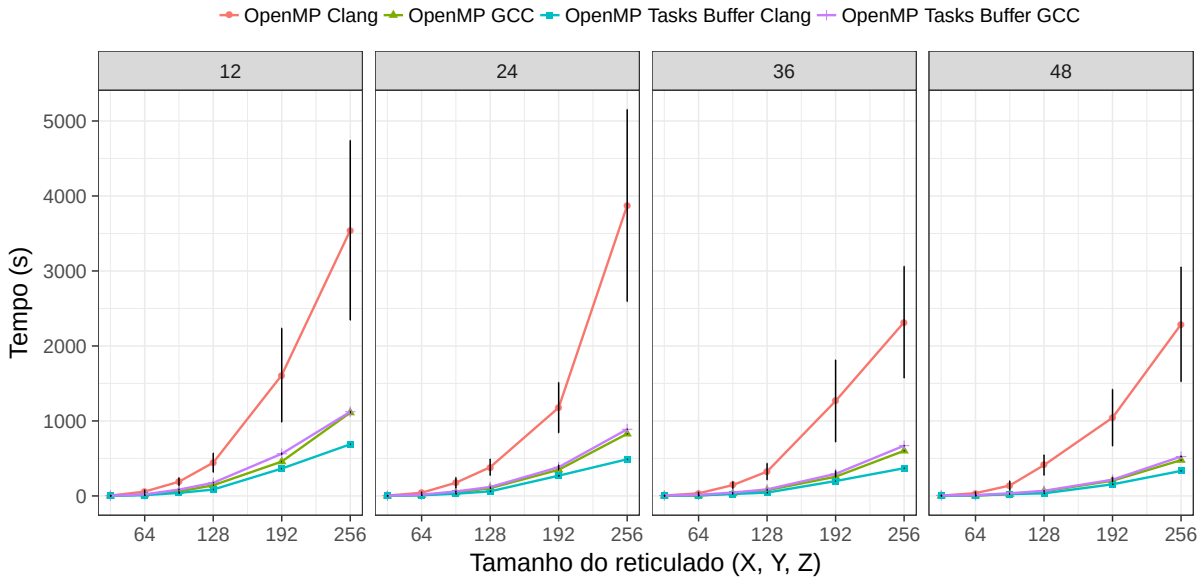
Fonte: O autor.

baseada em tarefas com `buffers` os melhores tempos de execução foram obtidos a partir da compilação por meio do compilador Clang. Em ambas as implementações as diferenças nos tempos de execução entre as compilações utilizando GCC e Clang são nítidas, especialmente na implementação for.

Na Figura 26 são apresentados os CPI das implementações for e baseada em tarefas com `buffers`. Analisando os ciclos por instrução utilizados por ambas as implementações nas compilações com GCC e Clang para os experimentos cujos melhores tempos de execução foram apresentados na Figura 25, nota-se que a implementação for compilada com Clang utilizou uma quantidade expressivamente maior de CPI comparado à compilação com GCC e que a implementação baseada em tarefas com `buffers` compilada com GCC, por sua vez, utilizou uma quantidade relativamente maior de CPI na maioria dos tamanhos de reticulados em comparação com a compilação com Clang. Os melhores tempos de execução e a utilização de um número menor de ciclos por instrução na implementação for compilada com GCC e na implementação baseada em tarefas com `buffers` compilada com Clang sugerem, por um lado, que o compilador GCC possui otimizações para o processamento paralelo de iterações ausentes no compilador Clang e, por outro, que o compilador Clang em conjunto com a *runtime* libKOMP possuem otimizações para o escalonamento de tarefas ausentes no compilador GCC.

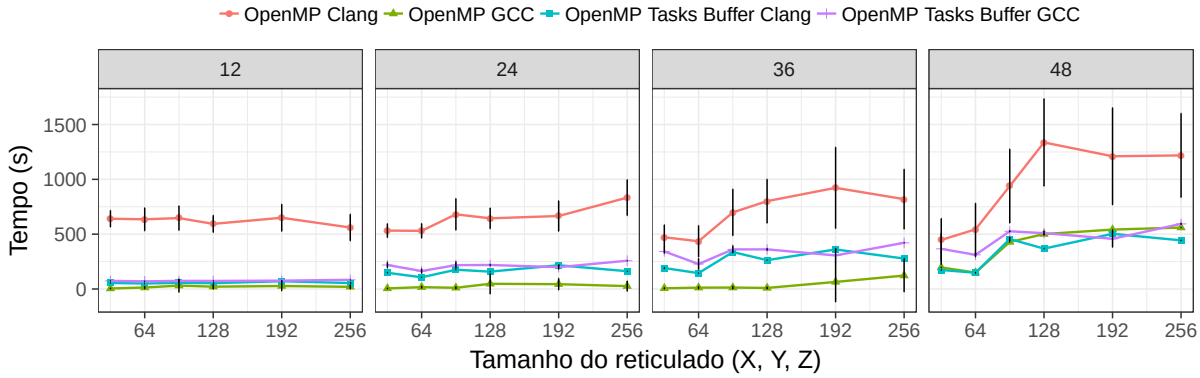
Na seção 4.2 foram apresentadas as sobrecargas paralelas introduzidas pelas imple-

Figura 25 – Tempo de execução das implementações OpenMP for e baseada em tarefas com buffers compiladas utilizando GCC e Clang em 12, 24, 36 e 48 threads.



Fonte: O autor.

Figura 26 – Ciclos Por Instrução (CPI) nas implementações OpenMP for e baseada em tarefas com buffers compiladas utilizando GCC e Clang em 12, 24, 36 e 48 threads.



Fonte: O autor.

mentações OpenMP do método a partir das quais foi possível observar que a utilização do padrão de acesso à memória SoA em conjunto com a inversão da ordem de execução dos laços de repetição que percorrem cada posição dos subdomínios na implementação baseada em tarefas OpenMP resulta em um ganho de desempenho significativo. Dessa forma, na Figura 27 são apresentados os tempos de execução das variações da utilização do padrão de acesso à memória SoA e da utilização da inversão dos laços de repetição. No gráfico dessa figura, além das implementações OpenMP baseadas em tarefas avaliadas até

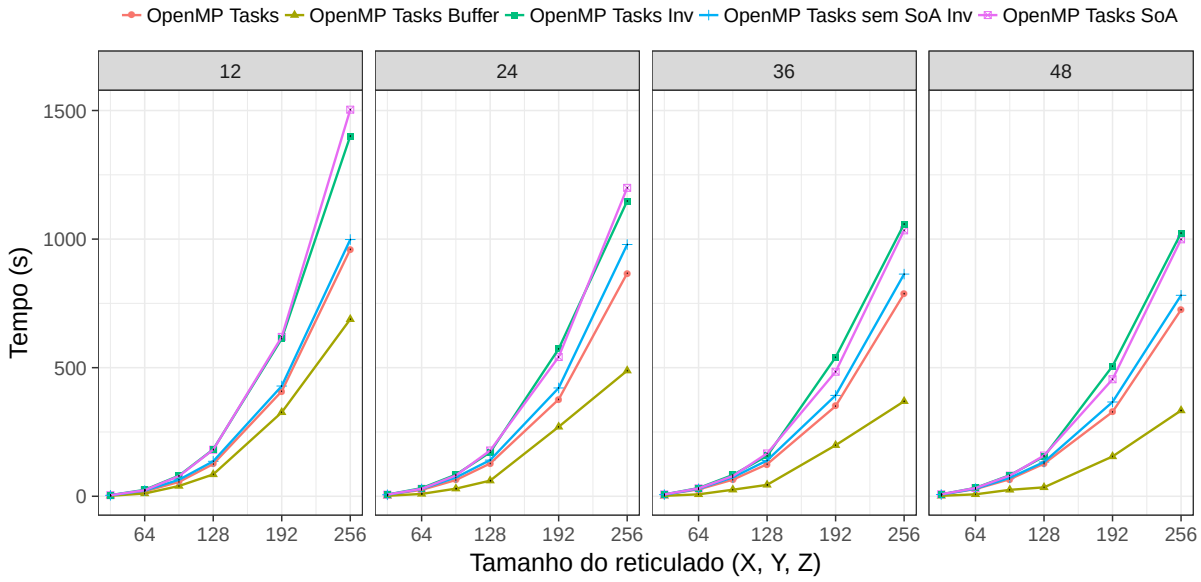
o presente momento e que utilizam tanto o padrão de acesso SoA à memória quanto a inversão dos laços de repetição que percorrem as matrizes unidimensional dos subdomínios, existem outras 3 implementações: uma na qual é utilizado apenas o padrão de acesso à memória SoA; outra na qual é utilizada apenas a inversão da ordem dos laços de repetição que percorrem as matrizes tridimensionais que armazenam as informações dos subdomínios; e outra implementação na qual não é utilizada nenhuma das duas estratégias de otimização.

Ao analisar o tempo de execução de ambas as implementações, especialmente nos reticulados de tamanho 256, é possível observar que as implementações que utilizam somente o padrão SoA ou somente a inversão dos laços de repetição possuem o maior tempo de execução entre todas as implementações. No entanto, ao comparar o tempo de execução da implementação que não utiliza nenhuma das estratégias com o tempo de execução da implementação que utiliza ambas as estratégias é possível notar que ambas possuem um desempenho significativamente melhor que o desempenho dessas implementações. Além disso, a implementação que utiliza ambas as estratégias ainda possui um desempenho relativamente melhor que a implementação que não utiliza nenhuma das estratégias de otimização. Por fim, com um tempo de execução expressivamente menor em comparação com as demais, a implementação que utiliza ambas as otimizações e ainda utiliza `buffers` na etapa de cópia das bordas dos subdomínios vizinhos se mostra como a implementação que proporciona o melhor desempenho.

A partir dos resultados apresentados na seção 4.3 é possível observar que a implementação OpenMP baseada em tarefas com `buffers`, além de possuir um tempo de execução significativamente menor em relação ao tempo de execução das implementações for e baseada em tarefas, transfere uma quantidade significativamente menor de memória, especialmente em reticulados de tamanho maior. O desempenho superior alcançado pela implementação é justificado pela utilização de matrizes temporárias durante a etapa de cópia das bordas dos subdomínios vizinhos, uma vez que essa é a única diferença existente entre essa implementação e a implementação baseada em tarefas que possui um desempenho significativamente inferior, tanto em relação à implementação baseada em tarefas com `buffers` quanto em relação à implementação for.

Conforme exemplificado por meio de execuções hipotéticas das implementações OpenMP baseadas em tarefas na subseção 3.3.2 e na subseção 3.4.2, a utilização de `buffers` remove as dependências existentes entre as tarefas de cópia das bordas em subdomínios vizinhos. Enquanto que na implementação baseada em tarefas a ordem de geração das tarefas de cópia das bordas dos subdomínios vizinhos nos diferentes subdomínios do reticulado interfere na posterior ordem de execução dessas tarefas por *threads*, na implementação baseada em tarefas com `buffers` a ordem de geração das tarefas não interfere na ordem de execução das mesmas. Dessa forma, tarefas de cópia das bordas na implementação baseada em tarefas com `buffers` podem ser executadas ambas

Figura 27 – Melhores tempos de execução obtidos com os melhores particionamentos para cada reticulado nas implementações utilizando somente SoA, utilizando somente inversão dos laços de repetição (Inv), não utilizando nem SoA nem a inversão dos laços, utilizando SoA juntamente com a inversão dos laços e utilizando SoA juntamente com a inversão dos laços e buffers em 12, 24, 36 e 48 threads.



Fonte: O autor.

paralelamente, o que possibilita a implementação alcançar um desempenho superior em relação à implementação baseada em tarefas mesmo possuindo uma tarefa adicional.

Um indicador de que a utilização de `buffers` nessa implementação aumenta o paralelismo no processamento das tarefas pelas *threads* é a diferença na quantidade de subdomínios em que cada tamanho de reticulado foi particionado nos experimentos em que ambas as implementações baseadas em tarefas obtiveram os melhores tempos de execução, apresentados na Tabela 4. Utilizando como exemplo o reticulado de tamanho 256, enquanto que a implementação baseada em tarefas obteve os melhores tempos de execução particionando o reticulado em 4 subdomínios por dimensão a implementação baseada em tarefas com `buffers` obteve os melhores tempos de execução particionando o reticulado em 16 subdomínios por dimensão. Ou seja, o particionamento desse reticulado em um total de 4096 subdomínios na implementação baseada em tarefas com `buffers` resultou em um tempo de execução 2,17 vezes menor comparado ao tempo de execução obtido a partir do particionamento desse reticulado em um total de 64 subdomínios na implementação baseada em tarefas. Comparando a quantidade de subdomínios em que cada reticulado foi particionado nos experimentos em que as implementações obtiveram os melhores tempos de execução é possível notar que, com exceção do tamanho 64, em todos

os tamanhos o particionamento na implementação baseada em tarefas com `buffers` foi expressivamente maior.

Tabela 4 – Configurações nas quais foram obtidos os melhores tempos de execução em cada implementação OpenMP em alguns tamanhos de reticulado.

	Tamanho do Reticulado					
	64	128	192	256	512	1024
OpenMP Tasks Buffer						
Tempo (s)	7,50	34,65	154,81	333,51	2961,13	23758,35
Threads	36	48	48	48	48	48
Subdomínios	4	8	8	16	16	32
OpenMP Tasks						
Tempo (s)	18,94	124,57	328,17	725,47	4198,45	32116,45
Threads	12	36	48	48	48	48
Subdomínios	4	4	4	4	8	16
OpenMP For						
Tempo (s)	4,03	60,87	203,02	477,82	3771,35	30653,72
Threads	48	48	48	48	48	48

Fonte: O autor.

Em relação ao número de *threads* utilizadas nos experimentos em que foram obtidos os melhores tempos de execução em ambas as implementações, apresentado na Tabela 4, é possível observar que na maioria das variações de tamanho de reticulado foi com 48 *threads*. Nas implementações baseadas em tarefas os melhores tempos de execução em reticulados maiores foram obtidos com 48 *threads* enquanto que em reticulados menores os melhores tempos de execução foram obtidos com quantidades menores de *threads*. Somente na implementação for os melhores tempos de execução em todos os tamanhos de reticulado foram obtidos com 48 *threads*.

Por fim, analisando o *speedup* de ambas as implementações nos experimentos realizados em um reticulado de tamanho 1024 utilizando 48 *threads*, observa-se que a utilização de tarefas OpenMP com dependências em conjunto com `buffers` possibilitou a obtenção de ganhos significativos de desempenho se comparado à paralelização de iterações de laços de repetição. Enquanto que o *speedup* da implementação baseada em tarefas foi de 0,95, ou seja, um aumento no tempo de execução de aproximadamente 4,77%, o *speedup* da implementação baseada em tarefas com `buffers` foi de 1,29, uma diminuição no tempo de execução de aproximadamente 22,49%. Além disso, se comparado o desempenho da implementação baseada em tarefas com `buffers` em relação à implementação baseada em tarefas, o *speedup* foi de 1.35, uma diminuição no tempo de execução de aproximadamente 26.02%. A utilização de `buffers` na etapa de cópia das bordas dos subdomínios vizinhos de cada subdomínio na implementação baseada em tarefas OpenMP possibilitou a remoção

das dependências que causavam a serialização da execução das tarefas, o que resultou em ganhos significativos de desempenho tanto em relação à implementação baseada em tarefas inicialmente proposta quanto em relação à implementação baseada no paralelismo de laços de repetição.

4.5 Considerações do Capítulo

Nesse capítulo apresentou-se os resultados dos experimentos realizados em um sistema de memória compartilhada *multicore*. Como foi possível observar, a implementação baseada em tarefas inicialmente proposta obteve resultados significativamente piores em relação aos obtidos pela implementação baseada no paralelismo de iterações de laços de repetição. No entanto, ao remover restrições que limitavam a execução paralela de tarefas, especialmente na etapa de cópia das bordas dos subdomínios vizinhos, foi possível observar um ganho de desempenho significativo tanto em relação à implementação baseada em tarefas inicial quanto em relação à implementação baseada no paralelismo de iterações de laços de repetição. Além disso, foi possível observar também que a compilação com dois compiladores distintos resulta em diferenças expressivas no desempenho em alguns casos e que a alocação intercalada de memória em arquiteturas NUMA resulta em ganhos significativos de desempenho.

Conclusão

Este trabalho teve como objetivo apresentar uma implementação baseada em tarefas e fluxo de dados do método de Lattice-Boltzmann utilizando a API OpenMP e avaliar o desempenho da implementação em comparação com uma implementação baseada no paralelismo de iterações de laços de repetição em uma arquitetura de memória compartilhada. Após realizar alguns experimentos iniciais com ambas as implementações constatou-se que o desempenho das implementações variava de acordo com o compilador para a linguagem C utilizado. Além disso, constatou-se ainda que, utilizando o compilador que proporcionava o melhor desempenho em cada implementação, o desempenho da implementação baseada em tarefas era consideravelmente inferior ao desempenho da implementação baseada no paralelismo das iterações do método.

Analisando as dependências de dados em cada tarefa da implementação baseada em tarefas notou-se que na tarefa de cópia das bordas dos subdomínios vizinhos a leitura e escrita das bordas em uma mesma estrutura de dados resultava em uma serialização na execução dessas tarefas em diferentes subdomínios, prejudicando o desempenho da implementação como um todo. Essa serialização foi o resultado da ordem de geração das tarefas associado à essa dependência, tornando as tarefas de cópia das bordas dependentes entre si uma vez que as bordas dos vizinhos são copiadas da matriz dos subdomínios vizinhos nas quais esses armazenam suas próprias bordas vizinhas. Dessa forma, para evitar conflitos no acesso aos dados na memória o escalonador de tarefas do OpenMP garantia que a execução fosse realizada sem causar inconsistências nos dados.

Ao compreender o motivo do desempenho inferior em relação à implementação baseada no paralelismo de iterações de laços de repetição desenvolveu-se uma segunda implementação baseada em tarefas do método de Lattice-Boltzmann, no entanto, utilizando matrizes temporárias na tarefa de cópia das bordas dos subdomínios vizinhos. Por meio dessas matrizes temporárias foi possível remover a dependência existente entre as tarefas de cópia das bordas de modo que as tarefas de cópia das bordas em todos os subdomínios pudessem ser realizadas simultaneamente. A inclusão de matrizes temporárias, denominadas `buffers` exigiu a inclusão de uma segunda tarefa para a realização da cópia das bordas. Enquanto a primeira tarefa copia as bordas vizinhas para os `buffers`, a segunda tarefa copia as bordas vizinhas armazenadas nos `buffers` para as matrizes de cada subdomínio. Dessa forma, ambas as tarefas que constituem a etapa de cópia das bordas vizinhas podem ser executadas simultaneamente.

Analisando o desempenho da implementação baseada em tarefas com `buffers` observou-se que o desempenho melhorou significativamente, não só em relação à tarefa de

cópia das bordas sem as matrizes temporárias, mas inclusive em relação ao desempenho da implementação baseada no paralelismo de iterações dos laços de repetição do método. Apesar de requerer a utilização de uma tarefa extra para a cópia das bordas dos `buffers` para a matriz do subdomínio, o fato das duas tarefas que compõem a etapa de cópia das bordas vizinhas não possuírem mais dependências entre tarefas em diferentes subdomínios permitiu que ambas fossem executadas paralelamente nos diferentes subdomínios respeitando apenas a ordem de geração de cada tarefa. O *speedup* da implementação baseada em tarefas com `buffers` em relação à implementação baseada no paralelismo de iterações de laços de repetição foi de aproximadamente 22,49, enquanto que o desempenho sem a utilização de `buffers` na implementação baseada em tarefas resultou em perdas de desempenho consideráveis em relação à tarefa baseada no paralelismo de iterações de laços de repetição.

Como trabalhos futuros sugere-se o desenvolvimento de uma implementação do método de Lattice-Boltzmann baseada em tarefas com dependências utilizando o conceito de zonas fantasmas e pré-processamento das bordas de modo que, ao invés da cópia das bordas ser realizada apenas utilizando uma posição em cada dimensão, a cópia das bordas seja realizada utilizando duas ou mais posições em cada dimensão dos subdomínios vizinhos. Além disso, sugere-se ainda o desenvolvimento de uma implementação mista MPI+OpenMP baseada em tarefas com dependências utilizando trocas de mensagens para arquiteturas de memória híbrida (distribuída e compartilhada) e o desenvolvimento de uma segunda implementação mista MPI+OpenMP baseada em tarefas utilizando, no entanto, o conceito de janelas (*windows*) compartilhadas para a troca de informações entre processos.

Referências

AGULLO, E. et al. Bridging the gap between openmp and task-based runtime systems for the fast multipole method. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 28, n. 10, p. 2794–2807, 2017. Citado na página 38.

AIDUN, C. K.; CLAUSEN, J. R. Lattice-Boltzmann Method for Complex Flows. *Annual Review of Fluid Mechanics*, v. 42, n. 1, p. 439–472, 2010. ISSN 0066-4189. Disponível em: <http://www.annualreviews.org/doi/10.1146/annurev-fluid-121108-145519>. Citado 3 vezes nas páginas 20, 25 e 27.

AKI, S. G. *The design and analysis of parallel algorithms*. Englewood Cliffs, New Jersey: Prentice Hall Inc., 1989. ISBN 0132000563. Citado na página 30.

BAŞAĞAOĞLU, H. et al. Computational performance of SequenceL coding of the lattice Boltzmann method for multi-particle flow simulations. *Computer Physics Communications*, Elsevier B.V., v. 213, p. 92–99, 2017. ISSN 00104655. Disponível em: <http://dx.doi.org/10.1016/j.cpc.2016.12.012>. Citado na página 37.

BAŞAĞAOĞLU, H. et al. Enhanced computational performance of the lattice Boltzmann model for simulating micron- and submicron-size particle flows and non-Newtonian fluid flows. *Computer Physics Communications*, Elsevier B.V., v. 213, p. 64–71, 2017. ISSN 00104655. Disponível em: <http://dx.doi.org/10.1016/j.cpc.2016.12.008>. Citado 2 vezes nas páginas 17 e 37.

BAUER, M. et al. Massively parallel phase-field simulations for ternary eutectic directional solidification. In: IEEE. *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*. [S.l.], 2015. p. 1–12. Citado na página 17.

BENZI, R.; SUCCI, S.; VERGASSOLA, M. The lattice Boltzmann equation: theory and applications. *Physics Reports*, v. 222, n. 3, p. 145–197, 1992. ISSN 03701573. Citado na página 19.

BHATNAGAR, P. L.; GROSS, E. P.; KROOK, M. A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems. *Physical Review*, v. 94, n. 3, p. 511–525, 1954. ISSN 0031899X. Citado na página 24.

BLOCKEN, B. Computational fluid dynamics for urban physics: Importance, scales, possibilities, limitations and ten tips and tricks towards accurate and reliable simulations. *Building and Environment*, Elsevier, v. 91, p. 219–245, 2015. Citado na página 17.

BROQUEDIS, F.; GAUTIER, T.; DANJEAN, V. libKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms. In: *IWOMP*. Rome, Italy: [s.n.], 2012. p. 102–115. Disponível em: <https://hal.inria.fr/hal-00796253>. Citado na página 77.

BUNTEMEYER, L. et al. Radiation hydrodynamics using characteristics on adaptive decomposed domains for massively parallel star formation simulations. *New Astronomy*, Elsevier, v. 43, p. 49–69, 2016. Citado na página 17.

CALORE, E. et al. Massively parallel lattice–boltzmann codes on large gpu clusters. *Parallel Computing*, Elsevier, v. 58, p. 1–24, 2016. Citado na página 36.

CHAPMAN, B.; JOST, G.; PAS, R. V. D. *Using OpenMP: portable shared memory parallel programming*. [S.l.]: MIT press, 2008. v. 10. Citado 3 vezes nas páginas 33, 34 e 35.

CHEN, S.; DOOLEN, G. D. Lattice Boltzmann Method for Fluid Flows. *Annual Review of Fluid Mechanics*, v. 30, n. 1, p. 329–364, 1998. ISSN 0066-4189. Disponível em: <http://www.annualreviews.org/doi/10.1146/annurev.fluid.30.1.329>. Citado 4 vezes nas páginas 19, 20, 23 e 27.

CHIESI, M. *Heterogeneous Multi-core Architectures for High Performance Computing*. Tese (Doutorado) — University of Bologna, 2014. Citado 2 vezes nas páginas 30 e 33.

CLAUSEN, J. R.; REASOR, D. A.; AIDUN, C. K. Parallel performance of a lattice-Boltzmann/finite element cellular blood flow solver on the IBM Blue Gene/P architecture. *Computer Physics Communications*, Elsevier B.V., v. 181, n. 6, p. 1013–1020, 2010. ISSN 00104655. Disponível em: <http://dx.doi.org/10.1016/j.cpc.2010.02.005>. Citado na página 17.

DONATH, S. et al. Performance comparison of different parallel lattice Boltzmann implementations on multi-core multi-socket systems. *Int. J. Comput. Sci. Eng.*, v. 4, n. 1, p. 3–11, 2008. ISSN 17427185. Disponível em: <http://portal.acm.org/citation.cfm?id=1457169>. Citado na página 17.

ELLIOTT, J. et al. The parallel system for integrating impact models and sectors (psims). *Environmental modelling & software*, Elsevier, v. 62, p. 509–516, 2014. Citado na página 16.

ENDO, T.; TAKASAKI, Y.; MATSUOKA, S. Realizing extremely large-scale stencil applications on gpu supercomputers. In: IEEE. *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*. [S.l.], 2015. p. 625–632. Citado na página 16.

FEICHTINGER, C. et al. Performance modeling and analysis of heterogeneous lattice Boltzmann simulations on CPU-GPU clusters. *Parallel Computing*, Elsevier B.V., v. 46, p. 1–13, 2015. ISSN 01678191. Disponível em: <http://dx.doi.org/10.1016/j.parco.2014.12.003>. Citado na página 17.

FLYNN, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21, n. 9, p. 948–960, 1972. ISSN 00189340. Citado 2 vezes nas páginas 28 e 29.

FRISCH, U.; HASSLACHER, B.; POMEAU, Y. Lattice as Automata for the Navier-Stokes Equation. *Physical Review Letters*, v. 56, n. 14, p. 1505–1508, 1986. ISSN 0031-9007. Citado na página 21.

GAJINOV, V. et al. Dash: A benchmark suite for hybrid dataflow and shared memory programming models: with comparative evaluation of three hybrid dataflow models. In: ACM. *Proceedings of the 11th ACM Conference on Computing Frontiers*. [S.l.], 2014. p. 4. Citado na página 38.

HARDY, J.; De Pazzis, O.; POMEAU, Y. Molecular dynamics of a classical lattice gas: Transport properties and time correlation functions. *Physical Review A*, v. 13, n. 5, p. 1949–1961, 1976. ISSN 10502947. Citado na página 21.

HE, X.; LUO, L.-S. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Physical Review E*, v. 56, n. 6, p. 6811–6817, 1997. ISSN 1063-651X. Disponível em: <https://link.aps.org/doi/10.1103/PhysRevE.56.6811>. Citado na página 20.

HENNESSY, L. J.; PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 012383872X, 9780123838728. Citado 3 vezes nas páginas 28, 29 e 31.

HWANG, K.; BRIGGS, F. A. *Computer architecture and parallel processing*. 1. ed. [S.l.]: McGraw-Hill, 1984. ISBN 0070315566. Citado 2 vezes nas páginas 29 e 32.

HWANG, K.; JOTWANI, N. *Advanced Computer Architecture*. 3rd. ed. [S.l.]: McGraw-Hill Education, 2011. Citado 3 vezes nas páginas 30, 31 e 32.

KRAUS, J. et al. Benchmarking GPUs with a parallel Lattice-Boltzmann code. *Proceedings - Symposium on Computer Architecture and High Performance Computing*, p. 160–167, 2013. ISSN 15506533. Citado na página 17.

LARUS, J. Spending Moore's dividend. *Commun. ACM*, v. 52, n. 5, p. 62–69, 2009. ISSN 0001-0782. Citado na página 16.

LIMA, J. V. et al. Performance and energy analysis of openmp runtime systems with dense linear algebra algorithms. In: IEEE. *Computer Architecture and High Performance Computing Workshops (SBAC-PADW), 2017 International Symposium on*. [S.l.], 2017. p. 7–12. Citado na página 16.

LUO, L.-s.; KRAFCZYK, M.; SHYY, W. Lattice Boltzmann Method for Computational Fluid Dynamics. *Encyclopedia of Aerospace Engineering*, n. 0, p. 651–660, 2010. Citado na página 27.

MARTÍNEZ, V. et al. Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system. In: IEEE. *Computer Architecture and High Performance Computing (SBAC-PAD), 2015 27th International Symposium on*. [S.l.], 2015. p. 1–8. Citado 2 vezes nas páginas 17 e 38.

MCNAMARA, G. R.; ZANETTI, G. Use of the boltzmann equation to simulate lattice-gas automata. *Physical Review Letters*, v. 61, n. 20, p. 2332–2335, 1988. ISSN 00319007. Citado na página 19.

MEADOWS, L.; ISHIKAWA, K.-i. OpenMP Tasking and MPI in a Lattice QCD Benchmark. In: *International Workshop on OpenMP*. [S.l.]: Springer, 2017. p. 77–91. Citado na página 37.

MOORE, G. E. Cramming more components onto integrated circuits. *Electronics*, v. 38, n. 8, p. 114–117, 1965. ISSN 00189219. Citado na página 16.

NAGAR, P. et al. LBM-IB: A parallel library to solve 3D fluid-structure interaction problems on manycore systems. *Proceedings of the International Conference on Parallel Processing*, v. 2015-December, p. 51–60, 2015. ISSN 01903918. Citado na página 37.

OpenMP Architecture Review Board. *OpenMP Application Programming Interface Version 4.5*. [S.l.], 2015. Citado 4 vezes nas páginas 33, 34, 35 e 36.

PACHECO, P. *An Introduction to Parallel Programming*. Elsevier, 2011. 1–370 p. ISSN 01928651. ISBN 9780123742605. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/C20090184714>. Citado 3 vezes nas páginas 30, 31 e 32.

PAS, R. van der; STOTZER, E.; TERBOVEN, C. *Using OpenMP – The Next Step: Affinity, Accelerators, Tasking, and SIMD*. Cambridge, MA: The MIT Press, 2017. 392 p. ISBN 9780262534789. Citado na página 35.

R Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2018. Disponível em: <http://www.r-project.org/>. Citado na página 79.

RAUBER, T.; RÜNGER, G. *Parallel Programming: for Multicore and Cluster Systems*. 1st. ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN 978-3-642-04817-3. Disponível em: <http://link.springer.com/10.1007/978-3-642-04818-0>. Citado 2 vezes nas páginas 30 e 32.

SADASIVAM, S. K. et al. IBM Power9 Processor Architecture. *IEEE Micro*, v. 37, n. 2, p. 40–51, 2017. ISSN 0272-1732. Citado na página 16.

SCHEPKE, C.; DIVERIO, T. A. *Distribuição de Dados para Implementações Paralelas do Método de Lattice Boltzmann*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, 2007. Citado 5 vezes nas páginas 17, 23, 25, 26 e 46.

SCHEPKE, C.; MAILLARD, N.; NAVAUX, P. O. A. Parallel lattice boltzmann method with blocked partitioning. *International Journal of Parallel Programming*, v. 37, n. 6, p. 593–611, 2009. ISSN 08857458. Citado na página 17.

SHAN, X.; CHEN, H. Lattice Boltzmann model for simulating flows with multi phases and components. *Physical Review E*, v. 47, n. 3, p. 1815–1819, 1993. ISSN 1063651X. Citado na página 19.

STERLING, J. D.; CHEN, S. Stability analysis of lattice boltzmann methods. *Journal of Computational Physics*, v. 123, n. 1, p. 196–206, 1996. ISSN 00219991. Citado na página 23.

TANG, P. et al. An Implementation and Optimization of Lattice Boltzmann Method Based on the Multi-Node CPU+MIC Heterogeneous Architecture. *2016 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, n. 1, p. 315–320, 2016. Disponível em: <http://ieeexplore.ieee.org/document/7864252/>. Citado na página 37.

TREIBIG, J.; HAGER, G.; WELLEIN, G. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In: IEEE. *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. [S.l.], 2010. p. 207–216. Citado na página 77.

VALERO-LARA, P.; JANSSON, J. Heterogeneous cpu+ gpu approaches for mesh refinement over lattice-boltzmann simulations. *Concurrency and Computation: Practice and Experience*, Wiley Online Library, 2016. Citado na página 36.

WEI, X. et al. The Lattice-Boltzmann Method for Simulating Gaseous Phenomena. *IEEE Transactions on Visualization and Computer Graphics*, v. 10, n. 2, p. 164–176, 2004. ISSN 10772626. Citado 3 vezes nas páginas 21, 22 e 23.

YE, Y. et al. Parallel computation of entropic lattice boltzmann method on hybrid CPU-GPU accelerated system. *Computers and Fluids*, v. 110, p. 114–121, 2015. ISSN 00457930. Citado na página 37.

ZHOU, W. et al. Lattice boltzmann parallel simulation of microflow dynamics over structured surfaces. *Advances in Engineering Software*, Elsevier, v. 107, p. 51–58, 2017. Citado na página 37.

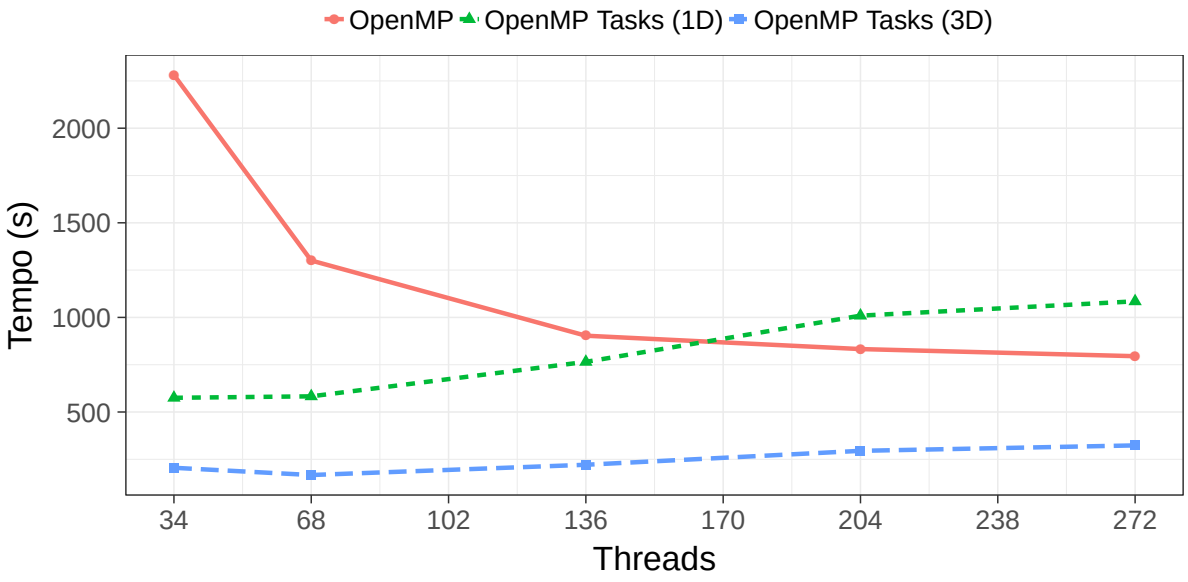
ZYUBAN, V. et al. Ibm power8 circuit design and energy optimization. *IBM Journal of Research and Development*, IBM, v. 59, n. 1, p. 9–1, 2015. Citado na página 16.

Anexos

ANEXO A – Implementação OpenMP Tarefas

Na Figura 28 é apresentado o tempo de execução da implementação baseada no paralelismo de iterações de laços de repetição e da implementação baseada no paralelismo de tarefas com particionamento do reticulado em apenas 1 dimensão (1D) e com particionamento do reticulado nas 3 dimensões (3D) em um sistema equipado com acelerador Xeon Phi KNL.

Figura 28 – Tempo de execução em acelerador Xeon Phi KNL.



Fonte: O autor.

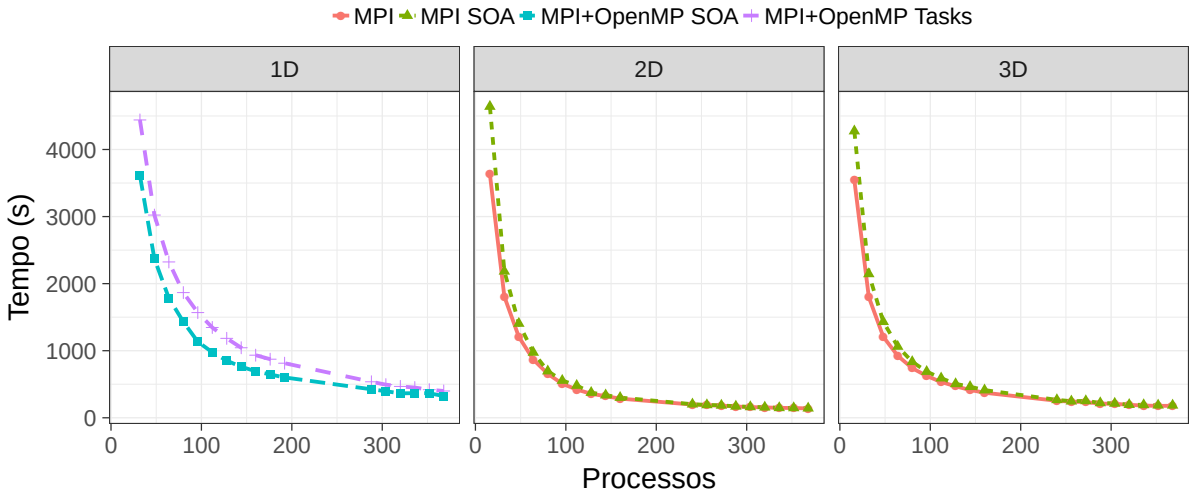
ANEXO B – Implementação MPI+OpenMP Tarefas

Neste anexo apresenta-se os tempos de execução e a percentagem de utilização do tempo de execução de cada função em implementações MPI puras, MPI+OpenMP baseado no paralelismo de iterações de laços de repetição e MPI+OpenMP baseado em tarefas com dependências. Os experimentos foram realizados em um sistema de memória distribuída denominado Nova. Esse sistema é composto por 23 nós Dell PowerEdge R430 e cada nó possui 2 *sockets*, totalizando 46 *sockets*. Em cada *socket* há um chip de processador conectado cujo modelo é Xeon Broadwell E5-2620 v4 da fabricante de circuitos integrados Intel. Os processadores possuem 8 núcleos de processamento físico com tecnologia de virtualização de *threads* denominada *hyper-threading* na qual cada núcleo físico pode ter até 2 *threads*, totalizando 368 núcleos de processamento físico e 736 *threads*. A frequência base desses processadores é 2.10GHz e pode ser ampliada para até 3.00GHz por meio da tecnologia *Turbo Boost*. Cada nó possui 64GB de memória RAM.

Na Figura 29 são apresentados os tempos de execução de ambas as implementações MPI. Os experimentos consistiram no particionamento somente em uma dimensão nas implementações MPI+OpenMP Tarefas e MPI+OpenMP For e em duas e três dimensões nas implementações puramente MPI. Como é possível observar, no particionamento 1D a implementação MPI+OpenMP For obteve tempos de execução relativamente melhores que os tempos de execução da implementação MPI+OpenMP Tarefas nas combinações de processos e particionamentos entre tarefas que obtiveram os melhores resultados. Já entre as implementações puramente MPI o desempenho de ambas tanto no particionamento 2D quanto 3D foi praticamente o mesmo.

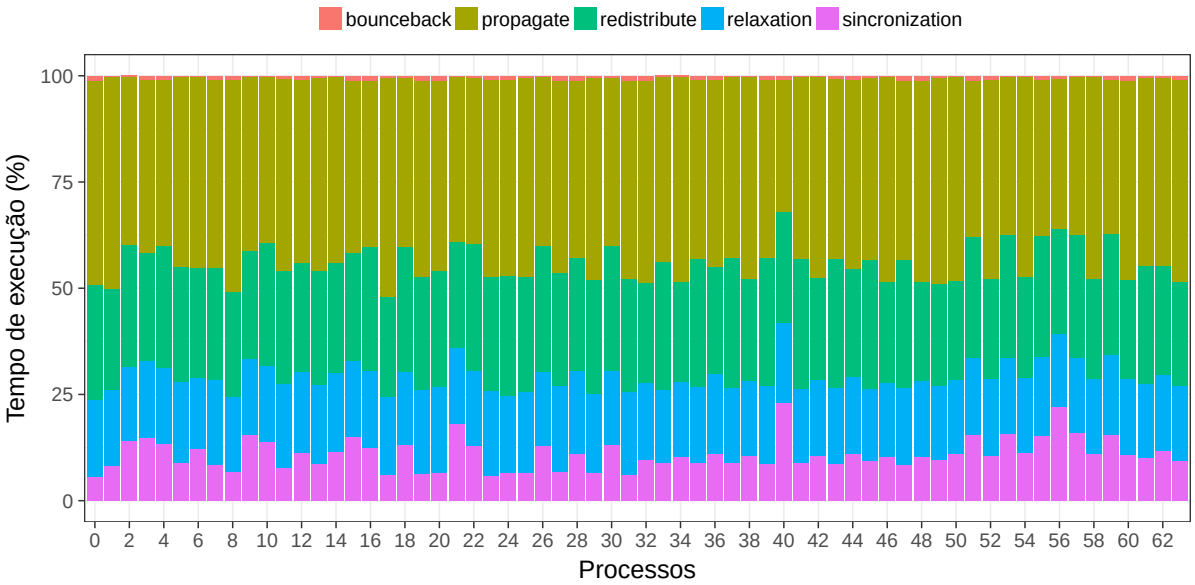
Já na Figura 30 é apresentada a proporção do tempo de execução correspondente à execução de cada função da implementação puramente MPI em 64 processos. Como é possível observar, grande parte do tempo de execução é dedicado ao processamento do método. Dentre as funções que mais demandam tempo de processamento estão propagação, redistribuição, relaxação e sincronização. Apesar de ser a função dentre essas 4 que utiliza a menor proporção do tempo de execução, a proporção dedicada à sincronização é considerável.

Figura 29 – Tempo de execução.



Fonte: O autor.

Figura 30 – Proporção do tempo de execução.



Fonte: O autor.