UNIVERSIDADE FEDERAL DO PAMPA

Hígor Uélinton da Silva

Paralelismo Aplicado à Simulação do Processo de Secagem de Grãos

Alegrete
2021

Hígor Uélinton da Silva

Paralelismo Aplicado à Simulação do Processo de Secagem de Grãos

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

Orientador: Prof. Dr. Claudio Schepke

Alegrete
2021

HÍGOR UÉLINTON DA SILVA

PARALELISMO APLICADO À SIMULAÇÃO DO PROCESSO DE SECAGEM DE GRÃOS

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

Trabalho de Conclusão de Curso defendido e aprovado em: 10 de março de 2022.

Banca examinadora:

____

Prof. Dr. Claudio Schepke

Orientador

UNIPAMPA

____

Profa. Dra. Aline Vieira de Mello

UNIPAMPA

____

Prof. Dr. Vinícius Garcia Pinto

FURG

Assinado eletronicamente por **CLAUDIO SCHEPKE**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 10/03/2022, às 17:46, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **ALINE VIEIRA DE MELLO**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 10/03/2022, às 17:46, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **Vinicius Garcia Pinto**, **Usuário Externo**, em 10/03/2022, às 17:46, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

A autenticidade deste documento pode ser conferida no site https://sei.unipampa.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **0750217** e o código CRC **9F47DFFA**.

Ficha catalográfica elaborada automaticamente com os dados fornecidos
pelo(a) autor(a) através do Módulo de Biblioteca do
Sistema GURI (Gestão Unificada de Recursos Institucionais) .

S586p Silva, Hígor Uélinton da
Paralelismo Aplicado à Simulação do Processo de Secagem de Grãos / Hígor Uélinton da Silva.
105 p.

Trabalho de Conclusão de Curso(Graduação)-- Universidade Federal do Pampa, CIÊNCIA DA COMPUTAÇÃO, 2022.
"Orientação: Claudio Schepke".

1. StarPU. 2. OpenMP. 3. OpenACC. 4. Aplicações científicas. 5. Secagem de grãos. I. Título.

AGRADECIMENTOS

Agradeço ao apoio e paciência do meu orientador, e também ao Prof. Dr. Vinícus Garcia pela ajuda com a interface StarPU. Agradeço também ao suporte da FAPERGS com as bolsas PROBIC (2020/2021) e PROBITI (2021/2022).

RESUMO

Estima-se que entre 10 % a 25 % da safra de grãos é perdida na pós-colheita. A secagem é uma das etapas mais críticas na sequência do processamento do grão para sua correta conservação após a colheita. Considerando que a massa do grão é uma quantidade de espaços sólidos e vazios (orifícios) através dos quais um fluido pode passar, a secagem do grão pode ser considerada um problema de meio aberto-poroso acoplado. Um modelo matemático e de simulação computacional foi previamente proposto para descrever a convecção em um fluxo livre com um obstáculo poroso aplicado à secagem de grãos. A partir dele, um esquema de dinâmica de fluidos computacional foi implementado em FORTRAN usando o método dos volumes finitos para simular e calcular as soluções numéricas. Já neste trabalho de conclusão de curso, o foco foi reduzir o tempo de execução da aplicação. Para determinar um modelo de paralelização eficiente para este problema, decidiu-se abordar as interfaces OpenMP, OpenACC e StarPU. O tempo total de simulação foi 8 vezes menor para uma arquitetura multicore (16 núcleos físicos) usando OpenMP e 17,3 vezes menor com uma única GPU (Quadro M5000) usando OpenACC. Ao contrário das outras duas, StarPU não proporcionou redução no tempo de execução da aplicação. Ainda, verificou-se que aumentar o número de tarefas/cores não é eficiente, uma vez que o tempo de execução piora. Dados os resultados, e uma análise mais aprofundada com a ferramenta StarVZ, pode-se concluir que a quantidade de computação em cada rotina paralelizada não foi suficiente para justificar os custos de criação das tarefas e gestão delas pelo *runtime*.

Palavras-chave: StarPU. OpenMP. OpenACC. Aplicações Científicas. Paralelismo de Tarefas. Secagem de Grãos.

ABSTRACT

It is estimated that between 10 % to 25 % of the grain crop is lost post-harvest. Drying is one of the most critical steps in the sequence of grain processing for its correct conservation after harvest. Considering that the grain mass is an amount of solid and empty spaces (holes) through which a fluid can pass, grain drying can be considered an coupled open-porous medium problem. A mathematical and computer simulation model was previously proposed to describe convection in a free flow with a porous obstacle applied to grain drying. From it, a computational fluid dynamics scheme was implemented in FORTRAN using the finite volume method to simulate and calculate numerical solutions. In this course conclusion work, the focus was to reduce the execution time of the application. To determine an efficient parallelization model for this problem, it was decided to address the OpenMP, OpenACC and StarPU interfaces. The total simulation time was 8 times lower for a multicore architecture (16 physical cores) using OpenMP and 17.3 times lower with a single GPU (Quadro M5000) using OpenACC. Unlike the other two, StarPU did not provide a reduction in application execution time. Also, it was found that increasing the number of tasks/cores is not efficient, since the execution time gets worse. Given the results, and a more in-depth analysis with the StarVZ tool, it can be concluded that the amount of computation in each parallelized routine was not enough to justify the costs of creating the tasks and managing them by runtime.

Key-words: StarPU. OpenMP. OpenACC. Scientific Applications. Tasks Parallelism. Grain Dryer.

LISTA DE FIGURAS

Figura 1 – Silo de secagem . . . 23
Figura 2 – Fluido Poroso Livre . . . 24
Figura 3 – Sem diretiva OpenMP x Com diretiva OpenMP . . . 31
Figura 4 – Sem diretiva OpenACC x Com diretiva OpenACC . . . 32
Figura 5 – Grade numérica esquemática . . . 34
Figura 6 – Volumes de controle discreto dentro do domínio . . . 35
Figura 7 – Figura esquemática de volume de controle único e pontos vizinhos. . . 35
Figura 8 – Interpolação de esquema rápido para propriedades avaliadas na face da célula . . . 36
Figura 9 – Grade escalonada: os pontos azuis são a grade original, os pontos vermelhos e amarelos representam o local de avaliação dos componentes das velocidades u e v, respectivamente. . . . 36
Figura 10 – Passos do algoritmo. . . . 37
Figura 11 – Etapas do TCC . . . 39
Figura 12 – Conversão dos 3 passos do algoritmo(Figura 10) para rotinas na aplicação. 41
Figura 13 – Antiga abordagem x Nova abordagem . . . 48
Figura 14 – Regiões de contorno nas chamadas `RESU` . . . 49
Figura 15 – Tempo de evolução de *streamlines* para Re = 100, $\varepsilon = 0.7$ e $Da = 5.10^{-3}$. 53
Figura 16 – *Speedup* relacionado à execução sequencial. . . . 56
Figura 17 – Disposição das tarefas dos *codelets* nos cores: **com** x **sem** barreira entre as rotinas. . . . 59
Figura 18 – Disposição das tarefas do *runtime* nos cores: **com** x **sem** barreira entre as rotinas. . . . 60
Figura 19 – Tarefas submetidas: **com** x **sem** barreira entre as rotinas. . . . 61
Figura 20 – Tarefas prontas: **com** x **sem** barreira entre as rotinas. . . . 61
Figura 21 – Tempo de execução de multiplicação de matrizes utilizando 1 tarefa . . 75
Figura 22 – Tempo de execução de multiplicação de matrizes utilizando 16 *slices* . . 77
Figura 23 – Particionamento dos dados com 4 *slices* . . . 78
Figura 24 – *Speed Up* de multiplicação de matrizes para diferentes dimensões e múltiplas implementações . . . 83

LISTA DE TABELAS

Tabela 1 – Interfaces/bibliotecas/*frameworks* utilizadas(os) nos trabalhos 27
Tabela 2 – Recursos da CPU 50
Tabela 3 – Recursos da GPU 50
Tabela 4 – Análise de desempenho com a ferramenta Perf 54
Tabela 5 – Performance: Tempo, variância e desvio padrão da execução 55
Tabela 6 – Performance: Tempo, variância e desvio padrão das rotinas paralelizadas com StarPU 56
Tabela 7 – Funções da API StarPU utilizadas 72
Tabela 8 – Estruturas da API StarPU utilizadas 73
Tabela 9 – Tempo de multiplicação de matrizes para diferentes dimensões e múltiplas implementações 82

LISTA DE CÓDIGOS FONTE

Código 1 – Bloco iterativo da aplicação. 40
Código 2 – Uso da diretiva `!$omp parallel do`. 42
Código 3 – Uso das diretivas OpenACC 42
Código 4 – Trecho do `codelet_U` associado a rotina `solve_u`. 44
Código 5 – Trecho de `solve_u`. 46
Código 6 – Bloco iterativo paralelizado da aplicação. 48
Código 7 – Esqueleto da aplicação Hello_World.c em StarPU 73
Código 8 – Codelet de uma multiplicação simples em CPU de matrizes 74
Código 9 – Definição e submissão de uma *task* 74
Código 10 – Criação, configuração e submissão de uma tarefa dividida em partes 74
Código 11 – Criação, configuração e submissão de uma tarefa em uma chamada de função 74
Código 12 – Estrutura do particionamento dos dados 78
Código 13 – Multiplicação de matrizes utilizando uma tarefa em CPU 85
Código 14 – Multiplicação de matrizes utilizando uma tarefa em GPU 85
Código 15 – Multiplicação de matrizes utilizando número de tarefas variável em CPU 86
Código 16 – Multiplicação de matrizes utilizando número de tarefas variável tarefas em CPU e GPU 87
Código 17 – Funções Externas 88
Código 18 – codelet_U associado com a rotina solve_u. 93
Código 19 – codelet_V associado com a rotina solve_v. 96
Código 20 – codelet_P associado com a rotina solve_p. 99
Código 21 – codelet_Z associado com a rotina solve_z. 101

LISTA DE SIGLAS

API *Application Programming Interface*

CPU *Central Processing Unit*

CUDA *Compute Unified Device Architecture*

GPU *Graphics Processing Unit*

MPI *Message Passing Interface*

OpenACC *Open Accelerator*

OpenCAL *Open Computing Abstraction Layer*

OpenCL *Open Computing Language*

OpenMP *Open Multi-Processing*

PETSc *Portable, Extensible Toolkit for Scientific Computation*

SUMÁRIO

Lista de Códigos Fonte 17

1 INTRODUÇÃO 23
1.1 Objetivos 24
1.2 Justificativa 25
1.3 Organização deste trabalho 25

2 TRABALHOS RELACIONADOS 27

3 INTERFACES DE PROGRAMAÇÃO PARALELA 31
3.1 OpenMP 31
3.2 OpenACC 31
3.3 StarPU 32
3.4 Considerações Finais do Capítulo 32

4 APLICAÇÃO 33
4.1 Formulação do Problema 33
4.2 Método dos Volumes Finitos 34
4.3 *Quadratic Upstream Interpolation* 34
4.4 Algoritmo 37
4.5 Considerações Finais do Capítulo 37

5 ESTRATÉGIAS METODOLÓGICAS 39
5.1 Implementação OpenMP 41
5.2 Implementação OpenACC 42
5.3 Implementação StarPU 42
5.3.1 1° abordagem 43
5.3.2 2° abordagem 43
5.3.3 3° abordagem – abordagem final 44
5.4 Configuração Experimental 50
5.5 Considerações Finais do Capítulo 51

6 ANÁLISE EXPERIMENTAL 53
6.1 Resultados Numéricos 53
6.2 Análise Preliminar da Performance 54
6.3 Avaliação de Desempenho 54

7 CONSIDERAÇÕES FINAIS 63

REFERÊNCIAS 65

APÊNDICES 69

APÊNDICE A – STARPU 71
A.1 Paradigma de Programação Orientado a Tarefas 71
A.2 Introduzindo StarPU em CPU 71
A.3 Particionamento de Dados 75
A.3.1 Aplicando Filtros nos Dados 78
A.4 Escalonadores 79
A.4.1 Políticas de Escalonamento sem Modelo de Performance . . . 79
A.4.2 Políticas de Escalonamento com Modelo de Performance . . . 79
A.4.3 Escalonadores Modularizados 80
A.5 Modelo de Performance 81
A.6 Alguns Resultados da API 82

APÊNDICE B – MULTIPLICAÇÃO DE MATRIZES 85

APÊNDICE C – CODELETS STARPU 93

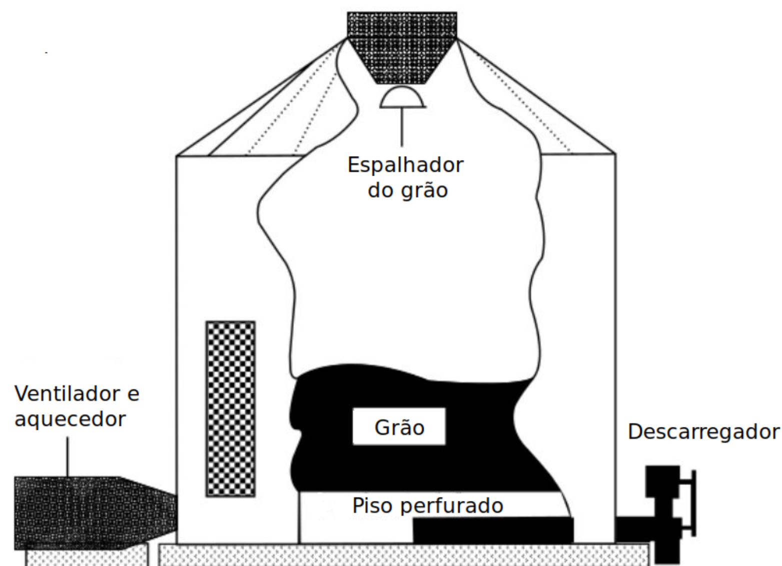
Índice 105

1 INTRODUÇÃO

Estima-se que entre 10% a 25% da safra de grãos seja perdida na pós-colheita (FAO, 2010). A secagem é um processo feito para retirar a umidade de um material, comumente utilizado na produção de alimentos, sendo uma das principais etapas para conservação do grão na pós colheita. Ela consiste na remoção da umidade do núcleo até que o teor de umidade seguro seja geralmente 12-14% em base úmida (BALA, 2017).

Existem dois tipos de secadores artificiais que são usados na produção de grãos: Silos e Secadores Portáteis. Os silos de secagem, como um secador de fluxo contínuo, recirculam o fluxo de secagem. Os secadores de fluxo contínuo geralmente funcionam com velocidades de fluxo de ar mais baixas do que outros tipos. Consequentemente, eles são mais eficientes em termos de energia. No entanto, é mais lento do que a maioria dos outros tipos de secadores. O silo de secagem é uma estrutura cilíndrica com piso perfurado, preenchido por um espalhador de grãos, uma unidade de ventilação quente e, recursos para varrer e descarregar os grãos sob o piso, conforme mostrado na Figura 1.

Figura 1 – Silo de secagem



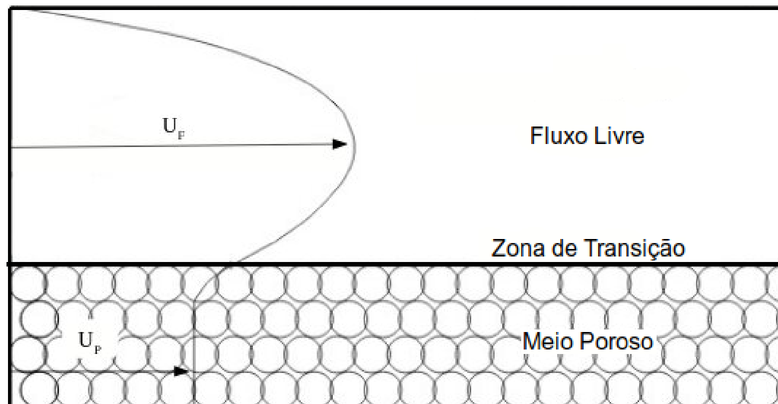
Fonte: (OLIVEIRA, 2020)

A ventoinha do aquecedor inicia quando os grãos são colocados dentro do silo e, desde que não atinja o menor teor de umidade médio dos grãos, o processo não termina (KROKIDA; MARINOS-KOURIS; MUJUMDAR, 2006). Todo este processo de secagem em leito fixo apresenta diferentes zonas de secagem: camada seca, camada de secagem e camada úmida, que devem ser consideradas para determinar o período de secagem. A zona seca está localizada no fundo da caixa e é a primeira camada a ser seca. O processo termina quando a zona úmida seca.

A modelagem matemática e simulação computacional são amplamente utilizadas para descrever a convecção em um fluxo livre com um obstáculo poroso. A previsão da

taxa de fluxo que passa através e ao redor de um meio poroso pode ser encontrada em muitos estudos na literatura. Usa-se a formulação da lei de Darcy e suas modificações atuais na parte porosa e a formulação de Navier-Stokes na parte aberta (CORNELISSEN, 2016). No entanto, deve-se levar em consideração a mudança abrupta no fluxo livre e meio poroso, criando uma zona de transição, conforme mostrado na Figura 2.

Figura 2 – Fluido Poroso Livre



Fonte: (OLIVEIRA, 2020)

Um esquema Dinâmico de Fluidos Computacional foi implementado em FORTRAN usando a técnica de volumes finitos para simular e computar as soluções numéricas (OLIVEIRA, 2020). Porém, esse tipo de aplicação demanda muito tempo de execução para qualquer simulação simples. Para reduzir esse tempo para um tempo aceitável, exploramos a simultaneidade das instruções em arquiteturas multicore e *Graphics Processing Unit* (GPU), que podem ser adotadas para aplicações numéricas.

1.1 Objetivos

O presente trabalho tem como objetivo obter um melhor desempenho da aplicação de secagem de grãos, contribuindo para a redução do seu tempo de execução. Além disso, uma solução melhor permite o cálculo de malhas particionadas mais finas, que representam uma carga de trabalho computacional mais alta, mas fornecem uma simulação mais precisa. Para atingir esse objetivo, adotamos estratégias de paralelização para criar uma versão da aplicação que roda em multicore e em GPU e explora seus recursos computacionais. Os objetivos específicos são:

- apresentar as etapas para acelerar uma aplicação não-paralela em arquiteturas multicore e GPU.
- reduzir o tempo de execução da aplicação à um tempo aceitável utilizando *Open Multi-Processing* (OpenMP), *Open Accelerator* (OpenACC) e StarPU.

- ver o impacto de uma interface de programação orientada a tarefas paralelas em uma aplicação de simulação.

1.2 Justificativa

No meio científico as aplicações científicas tendem a ser mais custosas do que aplicações do dia-a-dia. Dependendo da aplicação, horas ou dias podem ser gastos para se obter resultados mais interessantes. Hoje em dia, com computadores mais robustos, pode-se diminuir o tempo de espera por esses resultados. Ao invés de subutilizar um computador com programas sequenciais, tenta-se explorar sua capacidade máxima por meio da divisão desse programa em "partes menores". Essas divisões ficam a cargo, internamente, das *Application Programming Interface*s (APIs) de programação paralela.

Cada API possui um objetivo e é indicada para um domínio específico. StarPU é uma API ainda pouco conhecida e, por esse motivo, é explorada nesse trabalho com a intenção de fomentar seu conhecimento e mostrar sua capacidade.

Aplicações, como a simulação de secagem de grãos, são importantes e podem ser aplicadas na economia local. Como a região tem por tradição a produção de grãos em larga escala, armazenar e preservar são problemas enfrentados por ela e a pesquisa de novas soluções pode trazer benefícios.

1.3 Organização deste trabalho

Este trabalho está organizado em 7 capítulos. O Capítulo 2 destaca os trabalhos relacionados. O Capítulo 3 apresenta as interfaces de programação que são abordadas. O Capítulo 4 descreve a formulação matemática para esse problema e apresenta o algoritmo por trás da aplicação. O Capítulo 5 discute como o problema é tratado computacionalmente na forma paralela. O Capítulo 6 mostra os resultados numéricos e de performance para diferentes implementações paralelas. Por fim, o Capítulo 7 disserta as considerações finais sobre o trabalho e as perspectivas futuras.

2 TRABALHOS RELACIONADOS

Este Capítulo apresenta alguns trabalhos relacionados encontrados na literatura. Para a pesquisa dos trabalhos, foi utilizada principalmente a base IEEE, com as chaves de busca OpenMP, OpenACC e StarPU. Notou-se uma dificuldade na busca por artigos relacionados que utilizassem StarPU, especialmente aplicações que trabalhem com domínio multidimensional. Os trabalhos selecionados foram os que mais se assemelhavam com este trabalho, sendo por interfaces utilizadas, métodos ou domínio. Todos os trabalhos utilizam alguma interface de programação paralela para obter ganho de desempenho.

A Tabela 1 apresenta as interfaces, bibliotecas e/ou *frameworks* utilizadas(os) e também o *speedup* obtido, quando relatado, nos trabalhos relacionados apresentados neste capítulo. Os trabalhos que não apresentam, na tabela, *speedup*, ou apresentam os resultados com métricas diferentes ou não deixam claro no texto qual foi o maior *speedup* obtido. Dentre os trabalhos, 7 utilizam OpenMP e 5 OpenACC, onde 2 utilizam os dois simultaneamente, e 2 utilizam StarPU.

Tabela 1 – Interfaces/bibliotecas/*frameworks* utilizadas(os) nos trabalhos

Trabalho	**OpenMP**	**OpenACC**	**StarPU**	**Outras**	***Speedup***
1	Sim	Sim	Não	CUDA, OpenCL	–
2	Sim	Sim	Não	MPI, OpenMPI	6,50
3	Sim	Não	Não	–	2,86
4	Sim	Não	Não	–	6,05
5	Sim	Não	Não	–	2,00
6	Não	Sim	Não	–	21,00
7	Não	Sim	Não	–	31,43
8	Sim	Não	Não	–	7,00
9	Não	Sim	Não	–	–
10	Não	Não	Não	PETSc, cuSPARSE	16,00
11	Não	Não	Não	OpenCAL	447,00
12	Sim	Não	Não	MPI	–
13	Não	Não	Sim	MASA	–
14	Não	Não	Sim	–	–

Fonte: Autor

1. Em (GIMENES; PISANI; BORIN, 2018) é apresentada uma comparação entre diferentes interfaces de programação paralela, comparando itens como eficiência energética e desempenho computacional, para os métodos numéricos Ponto Médio Comum e Reflexão Comum. Como resultado dos estudos, usando as interfaces *Compute Unified Device Architecture* (CUDA), *Open Computing Language* (OpenCL), OpenACC e OpenMP, obteve-se um desempenho inferior em GPU usando OpenACC (comparado ao CUDA e OpenCL) e CPU (comparado ao OpenMP e OpenCL). Segundo seus testes, OpenCL mostrou-se mais eficiente em termos de desempenho ao utilizar

vetorização. Ainda, apontam que o desempenho da interface OpenCL é semelhante ao OpenMP quando a vetorização não é usada.

2. (GOYAL; LI; KIMM, 2017) aplicam a paralelização em três algoritmos de detecção de bordas em imagens de satélite. Os algoritmos utilizados para o estudo são Sobel, Prewitt e Canny, e as interfaces são OpenACC, OpenMP, *Message Passing Interface* (MPI) e a híbrida de OpenMP e MPI. OpenACC mostrou-se mais eficiente, obtendo uma aceleração de aproximadamente 6,5 vezes para os algoritmos.

3. Os autores em (SIMANJUNTAK; GUNAWAN, 2017) elaboram as simulações numéricas das equações de águas rasas de duas camadas (SWE) para avalanches submarinas usando, como em nosso trabalho, o esquema de volumes finitos. É feita uma comparação entre os resultados numéricos obtidos e o modelo de equações SWE-Exner. A simulação de avalanches submarinas próximas à topografia inclinada também é elaborada no artigo. A aceleração obtida foi em torno de 2,86 vezes em relação ao sequencial usando OpenMP com 4 cores.

4. Como em (SIMANJUNTAK; GUNAWAN, 2017), (JULIATI; GUNAWAN, 2017) discute a implementação usando a interface de programação paralela OpenMP em equações bidimensionais em águas rasas usando o método de volume finito usado para aproximar as equações, com a diferença de não aplicar o efeito de avalanches submarinas. O desempenho da implementação numérica é mostrado para grades maiores que 16×16. O maior ganho de desempenho veio com 8 *threads*, sendo 6,05 vezes comparado ao processamento serial. A eficiência máxima nesta simulação, para 8 *threads*, atingiu cerca de 75,6% no número de grades (Nx, Ny) = (256, 256).

5. O autor em (GUNAWAN, 2016) realiza um estudo sobre a implementação paralela do problema de difusão de calor unidimensional. A equação do calor contínuo é discretizada usando diferenças finitas explícitas. OpenMP é usado para a abordagem paralela, obtendo um ganho de desempenho de 2 vezes e uma eficiência de 50%.

6. (RAJ et al., 2018) trabalha com o método de métodos de fronteira imersa (IBM), usado para tratar fenômenos multifísicos. Este método usa uma malha cartesiana de estrutura fixa, gerando altas cargas de trabalho computacionais. O artigo apresenta um solucionador de limites embutido que usa diferenças finitas. Este solucionador é paralelizado usando a interface OpenACC, gerando uma aceleração de 21 vezes usando uma GPU NVIDIA Tesla P100 e 3,3 vezes usando uma CPU de soquete duplo Intel Xeon Gold 6148 de 20 núcleos.

7. O trabalho dos autores em (MORISHITA et al., 2021) traz uma implementação paralela do algoritmo Vlasov, código Vlasov girocinético (GKV). GKV é um código de simulação Vlasov baseado em equações girocinéticas delta-f em uma geometria de

tubo de fluxo local para analisar a turbulência do plasma em plasmas magnetizados. Para implementação paralela, OpenACC é utilizado. Como resultado, obteve-se uma aceleração de até 31,43 vezes em relação a execução sequencial.

8. O artigo (CARVALHO et al., 2020) utiliza paralelização via OpenMP para melhorar o desempenho de um simulador numérico de vazões bidimensionais em reservatórios de gás natural. Três métodos diferentes são utilizados para resolver o sistema de equações gerado pela formulação matemática na modelagem de escoamentos, Jacobi, Gauss-Seidel e SOR, e é feita uma comparação entre eles. Como resultado, obteve-se uma aceleração em torno de 7 vezes em relação à aplicação serial.

9. O autor em (ALMEIDA, 2021) implementa uma versão paralelizada, usando OpenACC, de um simulador numérico para escoamento não isotérmico em um reservatório de óleo pesado. O domínio de aplicação é bidimensional e discretizado em geometria cartesiana. Ele usa o método da diferença finita para discretizar as equações de fluxo governantes e o método do gradiente conjugado para determinar as variáveis de pressão e temperatura. Esse estudo relata uma aceleração da versão paralela em relação à versão sequencial.

10. O trabalho (CRUZ; MONSIVAIS, 2014) fornece um modelo de escoamento bifásico (água e óleo) em um meio poroso homogêneo. Assim como em nosso trabalho, o Método dos Volumes Finitos (FVM) é utilizado para resolver o modelo matemático apresentado. Eles comparam quatro esquemas numéricos para a aproximação de fluxos nas faces do volume discreto. Duas estratégias de paralelização são implementadas para reduzir o tempo de execução: usando CPUs, utilizando a biblioteca *Portable, Extensible Toolkit for Scientific Computation* (PETSc), e GPUs em cluster, utilizando a biblioteca cuSPARSE, baseada em CUDA. A aceleração foi de aproximadamente 8 usando 12 núcleos de CPU e cerca de 16 usando a GPU.

11. Em (RANGO et al., 2020), uma implementação de um modelo de fluxo insaturado tridimensional baseado em uma formulação direta discreta da equação de Richards foi apresentado. Como meio para paralelizar a aplicação, *Open Computing Abstraction Layer* (OpenCAL) foi utilizado. OpenCAL é uma biblioteca de software científico desenvolvida especificamente para a simulação de sistemas dinâmicos complexos 2D e 3D em dispositivos de computação paralela. Os desempenhos de computação foram avaliados para duas arquiteturas: um soquete Intel Xeon duplo e três GPUs Nvidia e um cluster de 16 nós com uma rede de interconexão rápida. A maior aceleração alcançada foi de 88 vezes nos soquetes da CPU Xeon e 447 vezes na GPU Titan XP.

12. Os autores em (HO et al., 2019) fornecem um solucionador de cinética de gás usando paralelização multinível desenvolvida para permitir simulações em escala de poros de

fluxos rarefeitos em meios porosos. A equação do modelo Bhatnagar–Gross–Krook é resolvida pelo método da velocidade discreta com um esquema iterativo. A paralelização multinível MPI/OpenMP é implementada, obtendo uma eficiência paralela de 94% em simulações 2D e 81% em simulações 3D.

13. O artigo (LOPES; THIBAULT; MELO, 2020) utiliza duas ferramentas para acelerar a comparação de sequencias baseadas no algoritmo de Smith-Waterman (SW). Ele propõe MASA-StarPU, um alinhador de sequências que integra o *framework* de domínio específico MASA a interface StarPU, criando uma ferramenta que possui os benefícios de ambos, como alinhamento rápido de sequências e políticas de agendamento de múltiplas tarefas. Como a poda de sequências leva a variados cenários, o trabalho defende que é impossível definir a melhor política de escalonamento, mas também apresenta o impacto de se escolher uma política de escalonamento erroneamente. Para 24 núcleos, comparando 5M x 5M, a aplicação obteve um tempo de execução de 1484 segundos com a política `dmda` e 3601 segundos com a política `lws`. Ainda para a comparação de 5M x 5M, a política de escalonamento de tarefas que mais gerou GCUPS (*Giga Connection Updates Per Second*) foi `dmdas`, com 18,41.

14. (KASMI et al., 2017) atacou aplicações lineares esparsas, tendo foco no Gradiente Conjugado Pré-condicionado (PCG), em arquiteturas heterogêneas. Avaliou-se o desempenho de escalonadores dinâmicos da interface StarPU e analisou-se a escalabilidade do algoritmo PCG. Diferentes combinações de recursos foram apresentadas, entre CPU e GPU, com o intuito de se obter a combinação que melhor utilizasse os recursos, e assim, obter um melhor desempenho. Como parte da conclusão, o aumento do número de cores da CPU não é eficiente devido ao custo de comunicação interno, porém, o aumento de GPUs proporciona uma aceleração significativa.

Em nosso trabalho, adotamos as interfaces de programação paralela OpenMP, OpenACC e StarPU para acelerar uma aplicação de meio aberto-poroso acoplado. A aplicação simula a secagem de grãos e usa interpolação quadrática para cinemática convectiva para esquema de diferenciação.

3 INTERFACES DE PROGRAMAÇÃO PARALELA

Nesse Capítulo, as APIs de programação paralela que foram utilizadas no trabalho são apresentadas superficialmente. APIs dão suporte a paralelização de aplicações, permitindo que, mesmo sem conhecimentos avançados em programação paralela, se consiga paralelizar aplicações e obter um melhor aproveitamento do hardware, consequentemente, um melhor uso do hardware pode significar uma redução no tempo de execução.

3.1 OpenMP

OpenMP é uma API para programação paralela de memória compartilhada e multiplataforma disponível em C/C++ e FORTRAN (OPENMP, 2021). Esta API é baseada no modelo de execução *fork-join*, onde uma *thread* mestre começa a ser executada e gera *threads* de trabalho para executar em paralelo conforme necessário ou especificado (CHAPMAN; MEHROTRA; ZIMA, 1998).

Essa interface utiliza diretivas para especificar os trechos de código onde se busca uma execução paralela. Desse modo consegue-se preservar a estrutura do código original, fazendo leves alterações tais como acréscimo de linhas ou alterações de estruturas. A Figura 3 apresenta a diferença entre um código sem e com diretiva OpenMP.

Figura 3 – Sem diretiva OpenMP x Com diretiva OpenMP

```
1
2 for(int i = inicio; i < limite; i ++){
3     //comandos
4 }
```

```
1 #pragma omp parallel for
2 for(int i = inicio; i < limite; i ++){
3     //comandos
4 }
```

Fonte: Autor

A quantidade de *threads* pode ser definida por uma variável de ambiente (`OMP_NUM_THREADS`) ou por uma função dentro do código (`omp_set_num_threads()`). Essa função recebe como parâmetro a quantidade de *threads* desejada na forma de um número natural maior do que 0.

3.2 OpenACC

OpenACC é uma API baseada em diretivas, assim como OpenMP, para o desenvolvimento de aplicações paralelas em arquiteturas heterogêneas, disponível para C / C++ e FORTRAN (CHANDRASEKARAN; JUCKELAND, 2017). Essas diretivas especificam *loops* e blocos de código que podem ser descarregados da *Central Processing Unit* (CPU) para um acelerador conectado (OPENACC, 2021).

Como em OpenMP, essa API utiliza de diretivas para especificar os trechos de código onde se busca uma execução paralela. A Figura 4 apresenta um código sem e com

diretiva OpenACC para o tratamento de dois *loops* aninhados. Nesse caso, é usada a diretiva `collapse`, que recebe como parâmetro o número de laços aninhados que estão por vir.

Figura 4 – Sem diretiva OpenACC x Com diretiva OpenACC

```
1
2 for(int i = inicio_i; i < limite_i; i++){
3     for(int j = inicio_j; j < limite_j; j++){
4         //comandos
5     }
6 }
```

```
1 #pragma acc parallel loop collapse(2)
2 for(int i = inicio_i; i < limite_i; i++){
3     for(int j = inicio_j; j < limite_j; j++){
4         //comandos
5     }
6 }
```

Fonte: Autor

3.3 StarPU

StarPU é uma API baseada no paradigma de programação paralela orientado a tarefas para arquiteturas híbridas. "Ao invés de lidar com problemas de mais baixo-nível, programadores podem se concentrar nos problemas do algoritmo!" (STARPU, 2018). A interface oferece tanto implementações em CPU como em GPU. Para programação em GPU, ela fornece suporte as interfaces CUDA e OpenCL. Está disponível em C/C++, mas possui algum suporte a FORTRAN, sendo esse nativo ou usando *marshalling wrappers* C. Maiores detalhes podem ser encontrados no Apêndice A.

O *runtime* da interface lida com: dependência de tarefas; *scheduling* heterogêneo otimizado; transferência de dados e replicação otimizada entre memória principal e memórias específicas; e comunicação em *cluster* otimizado.

3.4 Considerações Finais do Capítulo

Neste Capítulo foram apresentadas as interfaces que são abordadas no trabalho. Essas interfaces estão relacionadas com a tentativa de se encontrar a forma mais eficiente para paralelizar o problema apresentado.

4 APLICAÇÃO

A formulação matemática é apresentada neste Capítulo, onde o problema físico ajusta a equação governante de um modelo geral de fluxo de fluído para através e entorno de um meio poroso. A abordagem usa a modificação de Brinkman-Forchheimer para a equação de Darcy para meios porosos, sendo tratada como um problema de domínio único. Apesar de aplicar um conjunto de equações para cada região(domínio aberto e poroso), a abordagem de Domínio Único valida um conjunto de equações para todos os pontos de todo o domínio (BRETON; CALTAGIRONE; ARQUIS, 1991), (CORNELISSEN, 2016).

No presente trabalho são feitas as seguintes suposições:

- o fluxo laminar bidimensional, instável, de um fluxo incompressível e viscoso foi considerado;
- o meio poroso é isotrópico, uniforme e homogêneo;
- o cilindro poroso está completamente saturado de água;
- a abordagem de um domínio é usada para governar as equações;
- o diâmetro do cilindro poroso é suficientemente maior do que o raio de características dos poros do cilindro.

4.1 Formulação do Problema

A formulação de problemas de meio porosa usando variáveis normalizadas pode fornecer muitas vantagens. A ordem de magnitude das variáveis normalizadas é a mesma e, portanto, os erros de arredondamento numéricos resultantes de cálculos com valores de ordens de magnitude diferentes são evitados (VERSTEEG; MALALASEKERA, 2007). Figura 5 mostra a grade numérica esquemática. O problema é formulado em coordenadas cartesianas.

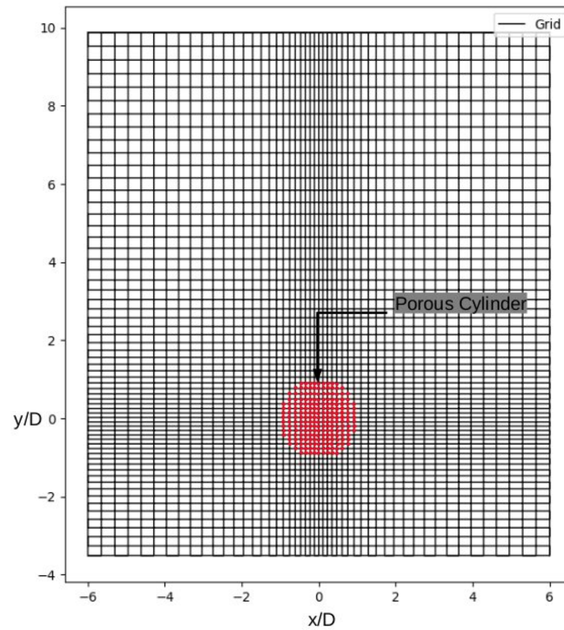
As variáveis não-dimensionais são definidas da seguinte forma:

$$x = \frac{\overline{x}}{d}, y = \frac{\overline{y}}{d}, u = \frac{\overline{u}}{u_0}, v = \frac{\overline{v}}{v_0}, p = \frac{\overline{p}}{\rho u_0^2}, T = \frac{\overline{T}}{T_c}, t = \frac{\overline{tu_0}}{d}, c = \frac{\overline{c}}{c_0}$$

onde $\overline{x}$ e $\overline{y}$ são coordenadas espaciais, e seus respectivos componentes de velocidade $\overline{u}$ e $\overline{v}$, u_0 é a velocidade de injeção, d é o diâmetro do cilindro, $\overline{p}$ é a pressão, t é o tempo e c é a concentração adimensional. A velocidade do fluido no meio poroso v está relacionada à velocidade do fluido livre pela equação de Dupuit-Forchheimer $v = \varepsilon V$, onde ε é a porosidade e V é a velocidade na região de fluido livre.

O centro do cilindro está localizado na origem do sistema cartesiano como mostrado na Figura 5. Assim, o domínio tem comprimento $2L_x$ e altura medindo $L_y = |2,5(-L_y)|$.

Figura 5 – Grade numérica esquemática



Fonte: (OLIVEIRA, 2020)

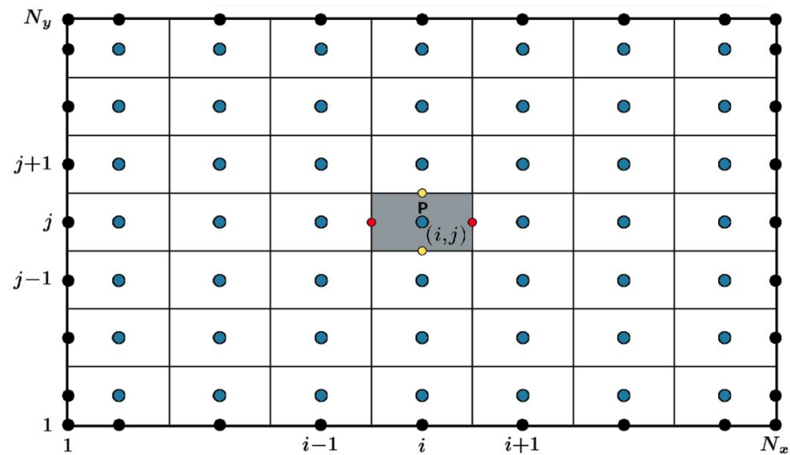
4.2 Método dos Volumes Finitos

O Método de Volumes Finitos (MVF) é uma técnica numérica amplamente usada. Neste método, o domínio é dividido em volumes de controle discretos, onde cada nó $P(i,j)$ é circundado por um volume de controle. Os limites dos volumes de controle são posicionados no meio do caminho entre os nós adjacentes. Cada volume de controle tem largura Δx e altura Δy. Como consequência, os limites físicos coincidem com os limites do volume de controle, como mostrado no Figura 6. Cada ponto $P(i,j)$ é delimitado por pontos vizinhos colocados na interface entre os dois volumes adjacentes denominados $n(i,j+\frac{1}{2})$, $s(i,j-\frac{1}{2})$ para norte e sul, $w(i-\frac{1}{2},j)$, $e(i+\frac{1}{2},j)$ para leste e oeste. Da mesma forma, os pontos colocados no centro dos volumes de controle vizinhos são definidos como $N(i,j+1)$, $S(i,j-1)$ para norte e sul, $W(i-1,j)$, $E(i+1,j)$ para leste e oeste, veja Figura 7. Além disso, as distâncias entre os pontos W e P, e entre os pontos P e E, são identificadas por δx_w e δx_e, respectivamente. Da mesma forma, as distâncias entre os pontos N e P, e entre os pontos P e S, são identificadas por δy_n e δy_s, respectivamente.

4.3 *Quadratic Upstream Interpolation*

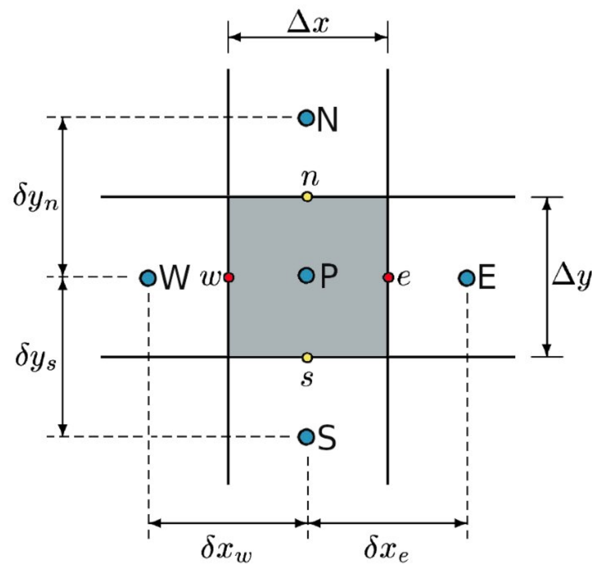
A *Quadratic Upstream Interpolation* para Cinemática Convectiva, chamada de esquema QUICK, é um dos métodos altamente utilizados para resolver problemas de convecção, baseado em uma formulação conservadora de controle de volume integral. Esta segurança é garantida pela boa precisão devido ao erro de truncamento espacial de terceira ordem e precisão de primeira ordem no tempo. Este método considera a

Figura 6 – Volumes de controle discreto dentro do domínio



Fonte: (OLIVEIRA, 2020)

Figura 7 – Figura esquemática de volume de controle único e pontos vizinhos.



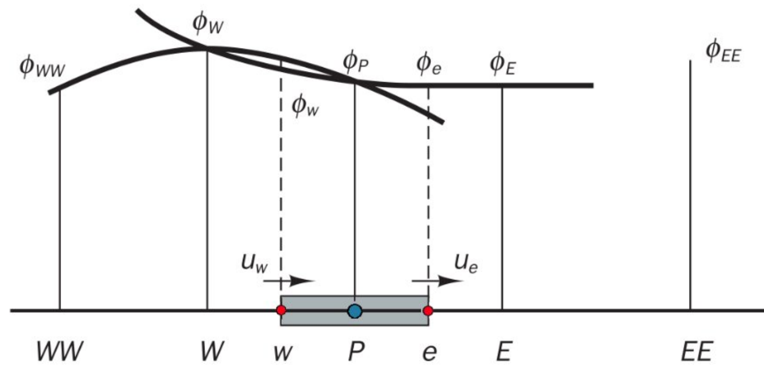
Fonte: (OLIVEIRA, 2020)

interpolação quadrática ponderada de três pontos a montante para os valores da face da célula (VERSTEEG; MALALASEKERA, 2007).

A escolha de armazenar as propriedades (u, v, p e T) no centro geométrico do volume de controle normalmente leva a oscilações não físicas e dificuldades na obtenção de uma solução convergente. Portanto, as velocidades devem ser estimadas em uma face de célula longa (w e e). Usando algum perfil de interpolação assumido, u_w e u_e podem ser formulados por um relacionamento da forma $u_f = f(u_{NB})$, em que NB denota o u valorizado nos nós vizinhos. Para garantir o acoplamento entre o campo de pressão e velocidade, uma segunda e terceira grades, que são escalonadas na direção x e y em relação à grade original, são usadas para os componentes de velocidade u e v, com a pressão

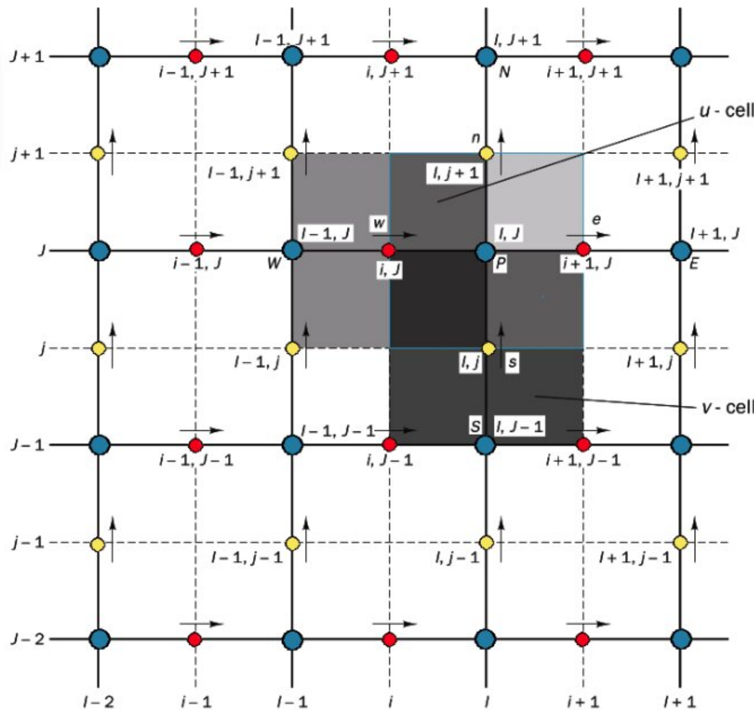
sendo calculado na grade original, como mostrado em Figura 9. A temperatura também é avaliada na grade original. Observe que as áreas dos componentes de velocidade u e v são avaliadas, respectivamente, pelos pontos vermelhos e amarelos, conforme mostrado na Figura 9. Este procedimento é conhecido como grade escalonada. Assim, o esquema Quick é usado e as velocidades nas interfaces de volume são calculadas por uma interpolação quadrática em esquema upwind.

Figura 8 – Interpolação de esquema rápido para propriedades avaliadas na face da célula



Fonte: (OLIVEIRA, 2020)

Figura 9 – Grade escalonada: os pontos azuis são a grade original, os pontos vermelhos e amarelos representam o local de avaliação dos componentes das velocidades u e v, respectivamente.

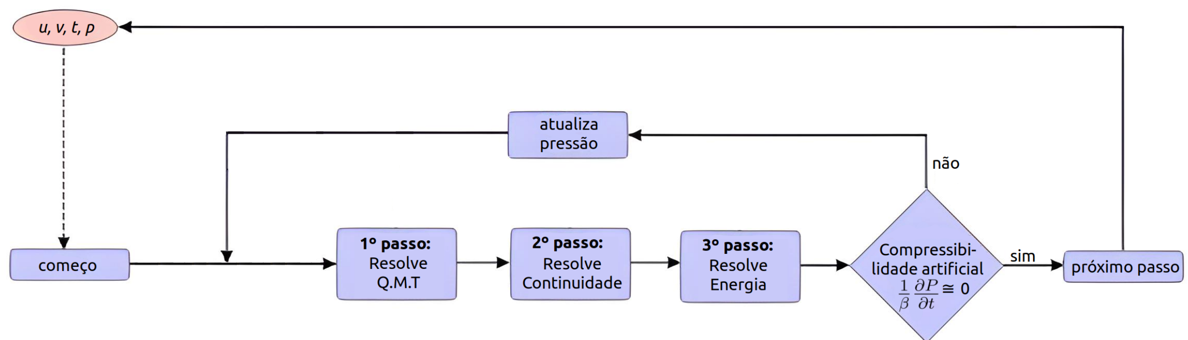


Fonte: (OLIVEIRA, 2020)

4.4 Algoritmo

O algoritmo da aplicação é apresentado na Figura 10. O algoritmo é composto essencialmente por um grande *loop* onde para cada passo de tempo são calculadas as propriedades físicas. Neste *loop*, o primeiro passo é resolver a quantidade de movimento. Em seguida, é necessário resolver a continuidade. Por último, resolve-se a energia. Se necessário, a pressão precisa ser recalculada.

Figura 10 – Passos do algoritmo.



Fonte: (OLIVEIRA, 2020)

4.5 Considerações Finais do Capítulo

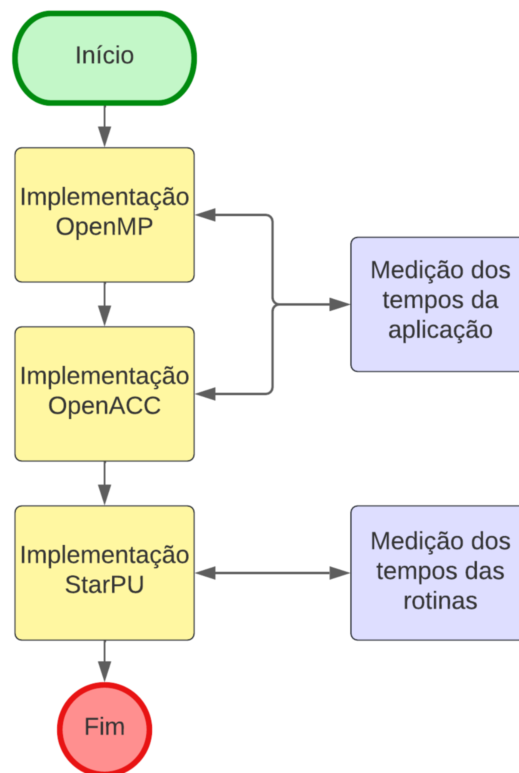
Neste Capítulo apresentou-se a formulação matemática por trás do problema de forma superficial. A formulação matemática completa pode ser encontrada em (SILVA et al., 2022). Também foi apresentado o algoritmo da aplicação.

5 ESTRATÉGIAS METODOLÓGICAS

Neste Capítulo, é descrita a implementação paralela da aplicação do problema de meio aberto-poroso acoplado. São listadas as interfaces de programação paralela utilizadas e descritos os procedimentos adotados para implementar o algoritmo de simulação numérica discreta. Também, os parâmetros utilizados na execução, como *flags* de compilação, são apresentados neste Capítulo.

A Figura 11 apresenta as etapas deste trabalho. Basicamente, este trabalho consiste nas implementações paralelas da versão sequencial da aplicação de simulação de secagem de grãos.

Figura 11 – Etapas do TCC



Fonte: Autor

O cilindro poroso é colocado em um ambiente de fluxo forçado. A temperatura inicial do cilindro é igual à temperatura ambiente $T\infty = T_p = 6$. Na parte inferior do domínio, o fluido é injetado com velocidade constante $= 1,0$ em temperatura fria $T = 1$. O presente campo de fluxo é governado por quatro parâmetros a saber: número de Reynolds (Re), número de Péclet (Pe), porosidade (ε) e número de Darcy (Da). Esses parâmetros foram variados de forma a analisar o comportamento de um fluxo laminar em torno de um cilindro sólido e ao redor e através de um cilindro circular poroso.

O código da aplicação consiste em etapas de pré-processamento, iteração e pós-

processamento. Todas essas etapas são chamadas explicitamente na rotina principal. Na etapa de pré-processamento, os arquivos de configuração são lidos, os dados são alocados e as variáveis são inicializadas. A etapa iterativa, apresentada no Código 1, consiste em um *loop* para o cálculo do pseudo-tempo. Neste *loop* são chamadas de rotinas `solve_u` e `solve_v` para resolver a equação de momento com QUICK, `solve_p` para resolver a equação de continuidade, e `solve_z` para resolver a equação de energia. A Figura 12 apresenta a conversão dos 3 principais passos do algoritmo apresentado na Figura 10 para rotinas na aplicação. No final do *loop*, a convergência é calculada e algumas variáveis são atualizadas. O pós-processamento consiste em gravar os resultados físicos em arquivos separados. Esses resultados são usados depois para plotar os resultados de saída usando *scripts* Python.

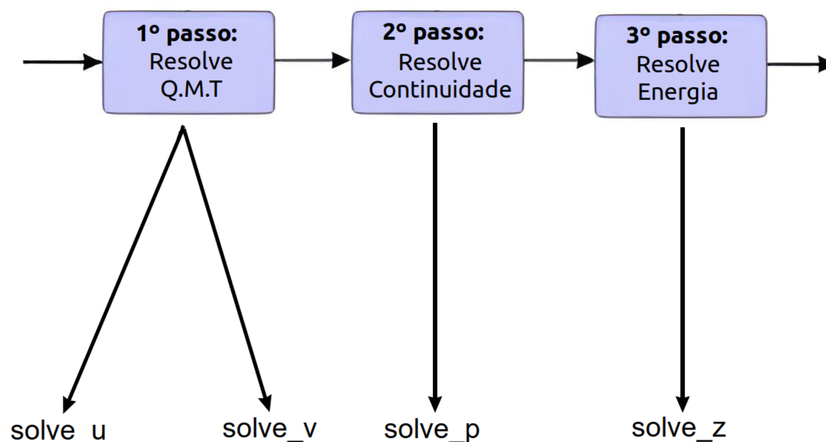
Código 1 – Bloco iterativo da aplicação.

```
1  DO WHILE (time .LT. final_time)
2      time = time + dt
3
4      ! Comeco do calculo do pseudo-tempo
5      DO WHILE(itc.LT.itc_max)
6          ! Resolucao da equacao do momento com QUICK
7          CALL solve_U(um,vm,um_n,um_tau,vm_tau,um_n_tau,pn,residual_u)
8          CALL solve_V(um,vm,vm_n,um_tau,vm_tau,vm_n_tau,pn,T,InvFr2,
       residual_v)
9
10         !  Resolucao da equacao da continuidade
11         CALL solve_P(c2,p,um_n_tau,vm_n_tau,pn,residual_p)
12
13         ! Resolucao da equacao da energia
14         CALL solve_Z(um_n_tau,vm_n_tau,T,T_n_tau,T_tau)
15
16         ! Verifica convergencia
17         CALL convergence(itc,c2,error,residual_p,residual_u,residual_v)
18
19         um_tau = um_n_tau
20         vm_tau = vm_n_tau
21         p = pn
22         T_tau = T_n_tau
23
24         ! Criterio de convergencia
25         IF (itc .NE. 1 .AND. error.LT.eps) then
26             EXIT
27         ENDIF
28     ENDDO
29     ! Fim do calculo do pseudo-tempo
30
31     um = um_n_tau
```

```
32      vm = vm_n_tau
33      T = T_n_tau
34
35      itc = 0
36      error = 100.d0
37
38      ! Resultados preliminares
39      IF (MOD(tr,n_tr).EQ.0) THEN
40          CALL comp_mean(u,v,um,vm)
41          CALL transient(u,v,p,T,Z,tr)
42
43          ! Grava num arquivo a saida
44          CALL output(um,vm,u,v,p,Z,T,H,itc)
45      ENDIF
46
47      tr = tr + 1
48  ENDDO
```

Fonte: (OLIVEIRA, 2020)

Figura 12 – Conversão dos 3 passos do algoritmo(Figura 10) para rotinas na aplicação.



Fonte: Autor

5.1 Implementação OpenMP

Threads paralelas foram criadas para dividir o cálculo dos *loops* das rotinas chamadas na etapa iterativa do código. Esses *loops* são responsáveis por percorrer os elementos da malha.

Para executar esses *loops* em paralelo – quando não há dependência de dados entre as iterações do *loop* – foi usada a diretiva `!$omp parallel do`. Quando os dados são restritos para cada *thread*, variáveis privadas são definidas. O Código 2 mostra o uso dessa diretiva em um trecho da aplicação.

Código 2 – Uso da diretiva `!$omp parallel do`.

```
!$omp parallel do private(i,j)
DO i=3,imax-1
    DO j=2,jmax-1
        res_u(i,j) = ((um(i,j) - um_tau(i,j)) + RU(i,j) * dt) * dtau
        ui(i,j) = (um_tau(i,j) + res_u(i,j))
    ENDDO
ENDDO
!$omp end parallel do
```

Fonte: Autor

5.2 Implementação OpenACC

Em nossa aplicação, os dados são copiados da CPU para GPU antes do início da etapa iterativa utilizando a diretiva `!$acc enter data copyin`. Esses dados são mantidos na GPU durante a execução das rotinas. Ao final da etapa iterativa, os resultados são descarregados da GPU para CPU para o pós-processamento utilizando `!$acc exit data copyout`. Para cada rotina paralela, a diretiva `!$acc data present` é definida, indicando as variáveis utilizadas previamente copiadas e já presentes na GPU.

A execução paralela é definida pela diretiva `!$acc parallel`. Como a maioria dos *loops* das rotinas são aninhados, porque a malha é bidimensional, adotamos a diretiva `!$acc loop collapse (2)`. O Código 3 apresenta um trecho do código com algumas das diretivas mencionadas.

Código 3 – Uso das diretivas OpenACC

```
!$acc data present (um(1:imax+1,1:jmax), ui(1:imax+1,1:jmax), &
!& RU(1:imax+1,1:jmax),res_u(1:imax+1,1:jmax),um_tau(1:imax+1,1:jmax))

!$acc parallel loop collapse(2)
DO i=3,imax-1
    DO j=2,jmax-1
        res_u(i,j) = ((um(i,j) - um_tau(i,j)) + RU(i,j) * dt) * dtau
        ui(i,j) = (um_tau(i,j) + res_u(i,j))
    ENDDO
ENDDO
!$acc end parallel
```

Fonte: Autor

5.3 Implementação StarPU

Diferentes metodologias foram abordadas para implementação da aplicação com StarPU. Variadas abordagens foram aplicadas para tentar tratar problemas advindos da

estrutura dos dados da aplicação e também do domínio dos dados. Todas as abordagens detalhadas são apresentadas nessa seção.

5.3.1 1° abordagem

Para cada passo apresentado na Figura 10, foram criados *codelets*. Na aplicação, o primeiro passo, resolução da equação do momento, é dividido em duas partes. Sendo assim, foram criados 4 *codelets* no total, 2 para resolução do momento, 1 para resolução da continuidade e 1 para resolução da energia. As rotinas que resolviam esses passos, `solve_u`, `solve_v`, `solve_p` e `solve_z`, foram modificadas, simultaneamente, para serem associadas a um *codelet*, com implementação em CPU.

Cada variável a ser utilizada nas rotinas da aplicação teve que ser registrada, ou seja, um *handle* foi associado a ela. Como existiam muitas variáveis, não foi possível utilizar um *buffer* simples como meio de passar os parâmetros de entrada para as rotinas. Ao invés disso, teve-se que empacotar os dados criando um descritor de dados. Um descritor de dados é um vetor onde cada posição está associada a um *handle*, como os *buffers* simples, porém com a vantagem de seu tamanho poder ser o quão grande se precisa.

A interface não oferece tratamento dos dados de entrada das tarefas de forma automática. Sendo assim, as matrizes de entrada foram particionadas, divididas horizontalmente, para evitar conflitos de escrita e possibilitar a execução paralela.

Feito isso, após a primeira execução, verificou-se que os resultados não estavam de acordo com os esperados. Depois de poucas iterações, os valores dos residuais U, V e P estouravam. Como eram muitas variáveis de entrada, na sua maioria matrizes, e o domínio era muito grande, o erro era quase impossível de ser encontrado. Haviam suposições de onde o erro poderia estar, como particionamento das matrizes, pois o domínio das matrizes não era perfeito, tendo matrizes com dimensão (2:imax-1)x(3:jmax-1) e outras com (imax+1)x(jmax).

5.3.2 2° abordagem

Paralelizar *loops* específicos das rotinas, onde o controle das variáveis é mais fácil, uma vez que existem menos variáveis envolvidas. Porém, novamente, decidiu-se por aplicar em todas as rotinas ao mesmo tempo.

Os *loops* das rotinas foram transformados em rotinas própias, para que assim pudessem ser associadas a *codelets*. Foi possível generalizar uma rotina para todos os *loops* das rotinas `solve_u`, `solve_v` e `solve_z`, adicionando parâmetros que controlam as operações. `solve_p` foi isolada porque continha alguma certa diferença das demais.

Contudo, nessa abordagem, começou-se a obter erros de memória, gerando falha de segmentação. Esse erro estava sendo gerado na declaração das variáveis de entrada, onde em Fortran90 quando se recebe uma matriz deve-se declará-la com sua dimensão, porém, com isso, não se pode declará-la como ponteiro, e para registrá-la como um *handle*

precisa-se que seja um ponteiro. Para achar a solução, descobriu-se que se pode declarar qualquer variável como *target* e que assim pode-se pegar seu ponteiro. Mesmo com essa descoberta, as falhas de segmentação estavam ocorrendo.

5.3.3 3° abordagem – abordagem final

Ao invés de trabalhar com as 4 funções simultaneamente, decidiu-se por abordar a que tinha menos complexidade entre as 4, `solve_p`. Somente as matrizes que estavam envolvidas nas operações de `solve_p` foram particionadas. Além disso, uma variável foi criada para se obter o controle da região onde a tarefa estava operando, sendo 0 para *top*, 1 para *middle* e 2 para *bottom*. Isso porque, dependendo da região das matrizes em que se estava, os cálculos eram diferentes.

Funções de *boundary* mexiam nas matrizes. Então, dependendo de qual parte estávamos, podíamos ou não fazer operações. Porém, mesmo com esse controle, o residual P continuava estourando. Viu-se que o particionamento não estava funcionando corretamente devido ao modo de alocação das matrizes, que estava sendo feito em duas dimensões. Para que pudesse ser implementada a versão com StarPU, teve-se que transformar essas matrizes 2D para 1D, ou seja, alocá-las contiguamente na forma de vetores.

Nem todas as matrizes utilizadas nas rotinas precisavam ser particionadas, uma vez que eram utilizadas somente para leitura. Com isso, as regiões onde cada tarefa trabalha precisam ser definidas. O controle dessas regiões é manual: linhas iniciais e finais são previamente calculadas para cada tarefa. O Código 4 apresenta um trecho do *codelet* associado a rotina `solve_u`, onde esse cálculo das regiões é feito. Todos os *codelets* completos estão no Apêndice C.

Código 4 – Trecho do `codelet_U` associado a rotina `solve_u`.

```
1  ! Calculo das linhas inicial e final das matrizes particionadas
2  ! lu = quantidade de linhas da matriz particionada um_n_tau
3  IF (pos .EQ. 1) THEN            !top
4      ini_i = 3
5      fim_i = lu
6  ELSE IF (pos .EQ. slices) THEN !bottom
7      ini_i = 1
8      fim_i = lu-2
9  ELSE                            !middle
10     ini_i = 1
11     fim_i = lu
12 ENDIF
13
14 ! Calculo auxiliar das linhas inicial e final das matrizes nao
       particionadas
15 ! imaxpu1 = (imax+1)/slices
16 ! resto1 = (imax+1)-slices*imaxpu1
```

```
17 IF (pos .EQ. 1) THEN              !top
18     ini_ui = 1
19 ELSE              !middle/bottom
20     fim_ui = pos*imaxpu1
21     IF (pos .LE. resto1) THEN
22         fim_ui = fim_ui + pos
23         ini_ui = fim_ui - imaxpu1
24     ELSE
25         fim_ui = fim_ui + resto1
26         ini_ui = fim_ui - imaxpu1 + 1
27     ENDIF
28 ENDIF
29
30 ! cu = quantidade de colunas da matriz particionada um_n_tau
31 res_u = 0.d0
32 DO i=ini_i,fim_i
33     DO j=2,jmax-1
34         ij = cu*(i-1)+j
35         ii = cu*(ini_ui-1)
36         res_u(i+(ini_ui-1),j) = ((um(ii+ij)-um_tau(ii+ij)) + RU(i+(ini_ui
       -1),j)*dt) * dtau
37         um_n_tau(ij) = 1.0d0 / 3.0d0 * um_tau(ii+ij) + 2.0d0 / 3.0d0 * ( ui
       (ii+ij) + res_u(i+(ini_ui-1),j) )
38     ENDDO
39 ENDDO
```

Fonte: Autor

Analisou-se `RESP`, que a principio estava correta, onde ela começava na linha 2 e ia até a linha imax-1. Assim como em *boundary*, os cálculos deveriam tomar o cuidado, quando *top*, irem da linha 2 até imax/(*slices*). Quando chegasse na última linha – não seria a última linha real da matriz, mas a última linha da matriz particionada –, deveria pegar a próxima linha para fazer os cálculos. Para acessar a próxima linha da matriz particionada era fácil, não estando na última parte da matriz, o particionamento não te impedia. Porém, para acessar a linha de cima de uma parte era impossível. Isso porque as matrizes começam em 1, sendo impossível acessar a posição 0. Para solucionar isso, estudou-se a necessidade do particionamento de todas as matrizes de `solve_p`. Chegando a conclusão negativa, optou-se por particionar somente as matrizes que estavam sendo modificadas na rotina. Assim, acessar as linhas anteriores não seria mais um problema. Como as matrizes não estavam mais sendo particionadas, o controle das regiões utilizando *top* (0), *middle* (1) e *bottom* (1) teve que ser modificado. Agora essa variável que controlava as regiões variaria de 1 a quantidade de *slices* das matrizes. Assim, com cálculos auxiliares, conseguimos acessar as corretas posições das matrizes que não foram particionadas.

Conseguiu-se dividir em um número n de tarefas a rotina `solve_p`. Verificou-se os

resultados até a 10ª iteração. Aparentemente eles estavam condizentes com os resultados esperados. Com isso, tendo os problemas sido abordados e enfrentados com uma das rotinas menos complexa, decidiu-se por abordar uma das rotinas mais complexas, onde `solve_u` foi escolhida.

`solve_u` possuí um nível de computação superior a `solve_p`, por conter mais chamadas de funções e uma região de contorno maior e com mais parâmetros envolvidos. Como em `solve_p`, verificou-se os resultados até a 10ª iteração. Percebeu-se uma leve alteração na 6ª casa decimal dos residuais, mas pensou-se que poderia ser algo relacionado ao *runtime*. Realizando mais testes, que iam até a 100ª iteração, percebeu-se que o erro, antes estando na 6ª casa decimal, estava mais agravado, mudando os residuais quase que completamente.

Devido a grande discrepância entre os valores obtidos e esperados, decidiu-se analisar novamente as regiões do domínio. O Código 5 mostra um trecho da função *solve_u* para que se possa entender melhor o problema encontrado. Vamos supor que `pos`, variável que controla as regiões das matrizes, seja igual a 1 (valor que representa o topo das matrizes), e que as matrizes foram particionadas em duas, horizontalmente. Na linha 1, temos a chamada da função `RESU`, que calcula `RU` da linha inicial até a linha do meio. Os cálculos em uma linha `RU` envolvem suas vizinhanças. Essas vizinhanças incluem duas linhas acima e duas linhas abaixo. Com isso, originalmente, os cálculos começam na 3ª linha e vão até a antepenúltima linha. Sendo assim, agora que se está trabalhando somente com a primeira parte da matriz, deve-se começar na 3ª linha e ir até a última linha, onde os cálculos da última linha da matriz vão precisar de duas linhas a frente. Na 1ª chamada da função `RESU`, os valores de `RU` estarão corretos, pois as duas linhas a frente de `um_tau` estão com os valores esperados. Com isso, os cálculos feitos da linha 3 a 9 também estarão corretos, onde `ui` terá seus valores atualizados do início até seu meio. O problema surge a partir da linha 11. Para calcular o novo `RU`, novamente iremos calcular até a última linha dele e, para calcular a última, pegamos as duas linhas a frente, dessa vez de `ui`. O problema é que essas duas linhas a frente de `ui` não foram calculadas anteriormente. Esses erros não apresentam grande relevância para pequenas iterações, porém com o passar do tempo interferem muito nos resultados.

Para corrigir esse problema, deve-se utilizar 6 linhas adiante de `um_tau`. A Figura 13 mostra a diferença da região pega na antiga e nova abordagem. Assim, no Código 5, a linha 1 utilizará 6 linhas adiante de `um_tau` para calcular 4 linhas adiante de `RU`. Com isso, o calculo de `ui`, na linha 7, poderá ser feito 4 linhas adiante. Com essas 4 linhas de `ui`, pode-se obter 2 linhas adiante de `RU`, na linha 11. Assim, na linha 17, `ui` poderá receber o calculo de duas linhas adiante, permitindo assim, o cálculo das duas últimas linhas `RU`, na linha 22. Com isso, teremos o calculo correto de `um_n_tau`, na linha 30.

Código 5 – Trecho de `solve_u`.

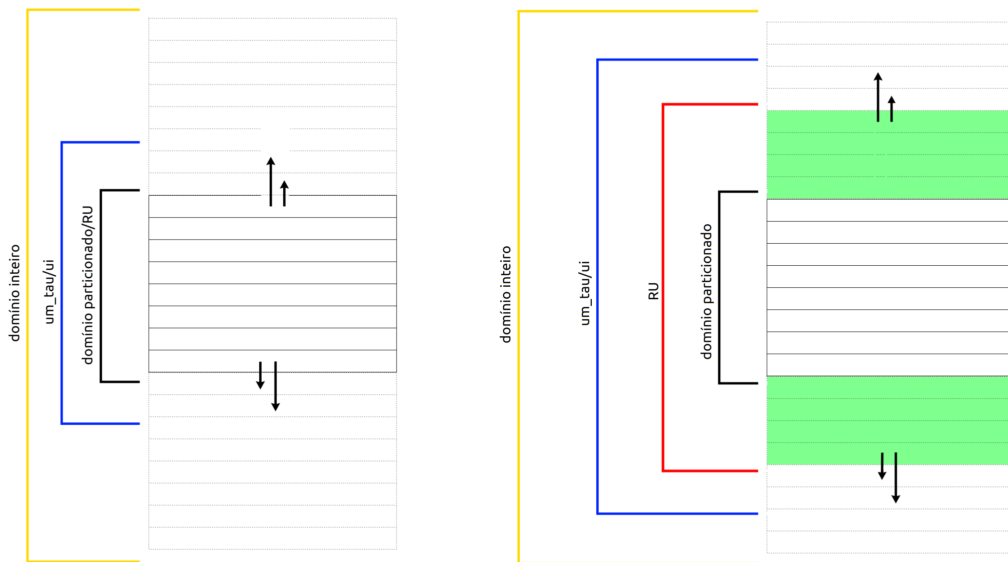
```
CALL RESU(um_tau,vm_tau,p,RU,pos)

DO i=ini_i,fim_i
        DO j=2,jmax-1
                ij = jmax*(i-1)+j
                res_u(i,j) = ((um(ij)-um_tau(ij)) + RU(i,j)*dt) * dtau
                ui(ij) = um_tau(ij) + res_u(i,j)
        ENDDO
ENDDO

CALL RESU(ui,vm_tau,p,RU,pos)

DO i=ini_i,fim_i
        DO j=2,jmax-1
                ij = jmax*(i-1)+j
                res_u(i,j) = ((um(ij)-um_tau(ij)) +  RU(i,j)*dt) * dtau
                ui(ij) = 0.75d0 * um_tau(ij) + 0.25d0 * &
                    (ui(ij) + res_u(i,j))
        ENDDO
ENDDO

CALL RESU(ui,vm_tau,p,RU,pos)

DO i=ini_i,fim_i
        DO j=2,jmax-1
                ij = cu*(i-1)+j
                ii = cu*(ini_ui-1)
                res_u(i+(ini_ui-1),j) = ((um(ii+ij)-um_tau(ii+ij)) + &
                    RU(i+(ini_ui-1),j)*dt) * dtau
                um_n_tau(ij) = 1.0d0 / 3.0d0 * um_tau(ii+ij) + &
                    2.0d0 / 3.0d0 * ( ui(ii+ij) + res_u(i+(ini_ui-1),j) )
        ENDDO
ENDDO
```

Fonte: Autor

A Figura 14 apresenta o domínio das matrizes necessário durante as chamadas de `RESU` para o correto cálculo dos valores. Esse domínio vai diminuindo a medida que a rotina é chamada.

O Código 6 apresenta a parte da etapa iterativa modificada com as rotinas paralelizadas. Cada rotina foi dividida em n tarefas, onde o número de tarefas é controlado pela variável *slices*. As tarefas de cada rotina são lançadas na mesma ordem da aplicação sequencial, ou seja, primeiro todas as tarefas da rotina `solve_u`, depois todas as tarefas de `solve_v`, e assim por diante. E, para manter a integridade dos dados manipulados, uma barreira é adicionada entre o lançamento das tarefas dessas rotinas.

Figura 13 – Antiga abordagem x Nova abordagem

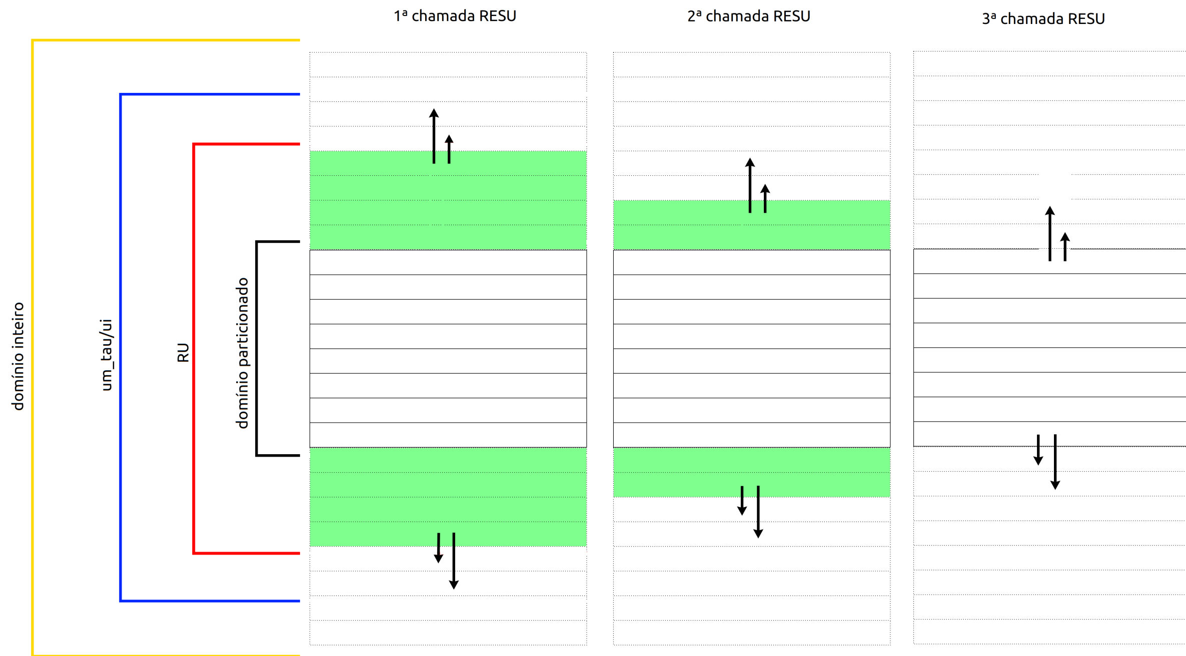


Fonte: Autor

Código 6 – Bloco iterativo paralelizado da aplicação.

```
1  DO i=1, slices
2      CALL fstarpu_insert_task((/ codelet_U , FSTARPU_DATA_MODE_ARRAY,
       descrit_U(i), C_LOC(buffers_U), C_NULL_PTR /))
3  ENDDO
4
5  CALL fstarpu_task_wait_for_all()
6
7  DO i=1, slices
8      CALL fstarpu_insert_task((/ codelet_V , FSTARPU_DATA_MODE_ARRAY,
       descrit_V(i), C_LOC(buffers_V), C_NULL_PTR /))
9  ENDDO
10
11 CALL fstarpu_task_wait_for_all()
12
13 DO i=1, slices
14     CALL fstarpu_insert_task((/ codelet_P , FSTARPU_DATA_MODE_ARRAY,
       descrit_P(i), C_LOC(buffers_P), C_NULL_PTR /))
15 ENDDO
16
17 CALL fstarpu_task_wait_for_all()
18
19 DO i=1, slices
20     CALL fstarpu_insert_task((/ codelet_Z , FSTARPU_DATA_MODE_ARRAY,
       descrit_Z(i), C_LOC(buffers_Z), C_NULL_PTR /))
21 ENDDO
```

Figura 14 – Regiões de contorno nas chamadas RESU



Fonte: Autor

```
22
23 CALL fstarpu_task_wait_for_all()
24
25 residual_u = MAXVAL(ABS(residual_us))
26 residual_v = MAXVAL(ABS(residual_vs))
27 residual_p = MAXVAL(ABS(residual_ps))
```

Fonte: Autor

Após o término da etapa iterativa, os *handles* foram desparticionados, quando necessário, e desalocados. Além disso, todas as estruturas, como *codelets* e descritores, foram desalocadas, e a função *shutdown* foi chamada para finalizar o *runtime*.

A diferença entre o tamanho do código gerado com OpenACC e OpenMP é pequena, pois ambos usam pragmas para definir os trechos de código a serem paralelizados. OpenACC acaba gerando mais código porque, para cada trecho de código a ser paralelizado, ele precisa informar quais dados devem ser transferidos para a GPU ou, se já estiverem presentes nela, indicar os dados a serem manipulados. Diferentemente delas, StarPU gerou muito código adicional devido a grande quantidade de parâmetros das rotinas paralelizadas. Isso torna o uso da interface menos atrativo em comparação com as outras duas, uma vez que o trabalho é muito maior, exigindo mais esforço e tempo para conseguir gerar código paralelo eficiente.

Tabela 2 – Recursos da CPU

Recursos	**Xeon E5-2650 (×2)**
Frequência	2.00 GHz
Núcleos / *Threads*	8 (×2) / 16 (×2)
Cache L1	32 KB
Cache L2	256 KB
Cache L3	20 MB
Memória RAM	128 GB

Tabela 3 – Recursos da GPU

Recursos	**Quadro M5000**
Frequência	1.04 GHz
Núcleos CUDA	2048
Cache L1	64 KB
Cache L2	2 MB
Memória Global	8 GB

Fonte: Autor

5.4 Configuração Experimental

O ambiente computacional utilizado neste trabalho para execução dos testes é composto por dois processadores Intel Xeon E5-2650 e uma GPU Nvidia Quadro M5000. Os detalhes são descritos na Tabela 2 e Tabela 3. Ubuntu focal 20.04 é o sistema operacional usado com versão GNU/Linux 5.11.0-43-generic e gcc 9.3.0. A versão CUDA é a 10.1.243. O kit de ferramentas do compilador usado foi o hpc_sdk 21.2. da NVidia. O compilador pgf90 foi usado para ambos os casos, com o adicional da *flag* `-O3`. *Tags* `-fopenmp` para OpenMP e `-fast`, `-acc`, e `Minfo=all` para OpenACC foram usadas.

Para uma melhor amostragem do tempo de execução da aplicação, sequencial, com OpenMP e OpenACC, foram realizadas 32 execuções para cada uma das malhas. Dessas 32 execuções, o maior e o menor tempo foram descartados. Foi utilizado o comando `time`, nativo do *command line interface*, para obter o tempo de execução total da aplicação. Três tamanhos de malha foram escolhidos: 51x63, 100x124, 200x249, utilizando 20.000 iterações para cada simulação. Esse número de iterações permite a secagem do grão. Os tamanhos de malha foram escolhidos para representar adequadamente o problema.

Para medir o tempo das rotinas, sequencial e com StarPU, foi utilizada a rotina nativa do Fortran90 `cpu_time()`. Foram realizadas 2.000 medições, com 4 tamanhos de malhas diferentes, sendo os mesmos três utilizados nos testes com OpenMP e OpenACC e a adição de um quarto tamanho: 400x499. Essa adição de uma malha maior tem como objetivo verificar se tamanhos maiores proporcionariam *speedup* com a interface StarPU. Para medir o tempo das rotinas foi utilizado somente 20 iterações.

Dos valores obtidos nas medições, para os casos de teste, obteve-se a média aritmética, a variância e o desvio padrão. Esses valores permitem realizar análises a respeito da qualidade das implementações e das medições, bem como calcular o *speedup*.

Para poder tirar melhores conclusões a respeito da paralelização com StarPU, verificando aspectos das tarefas, como a submissão delas e disposição entre os cores, utilizou-se StarVZ[1], uma ferramenta que permite a visualização gráfica do tempo de processamento e comunicação das tarefas, baseado na coleta de traços durante a execução.

[1] Disponível em: https://CRAN.R-project.org/package=starvz

Para isso, setou-se a variável de ambiente `STARPU_GENERATE_TRACE`, atribuindo o valor 1 a ela.

5.5 Considerações Finais do Capítulo

Nesse Capítulo foram apresentados os aspectos metodológicos para o desenvolvimento do trabalho. Foram descritos os métodos adotados para a implementação e para a realização dos testes, as ferramentas utilizadas, a máquina e a forma com que os resultados são expressos para permitir a reprodutibilidade dos experimentos.

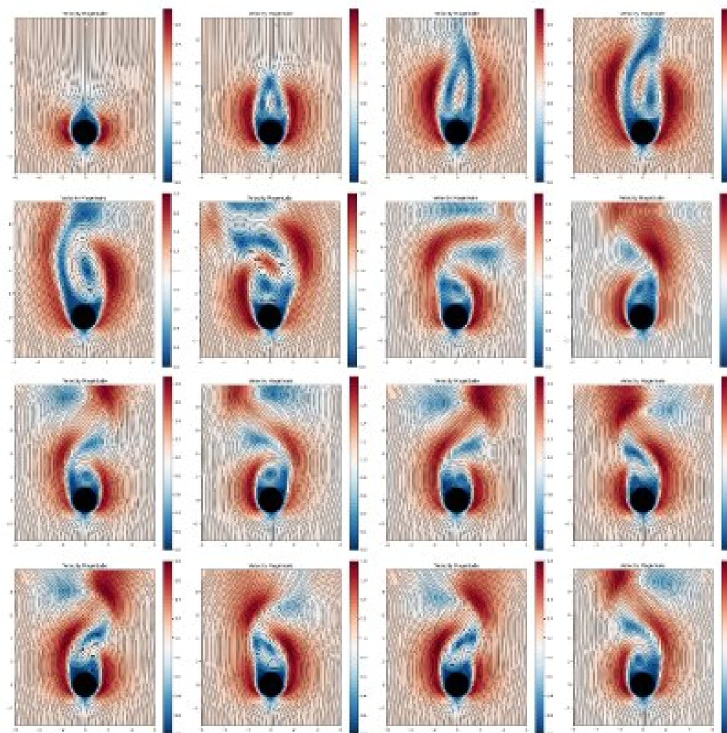
6 ANÁLISE EXPERIMENTAL

Este Capítulo apresenta alguns experimentos realizados para avaliar os resultados numéricos da aplicação e as diretivas paralelas adotadas para CPU usando OpenMP e StarPU, e para GPU usando OpenACC. Para cada estudo de caso, tanto os resultados que representam o desempenho da aplicação quanto a qualidade de seus resultados numéricos foram destacados quando comparados aos resultados gerados pela versão original (sequencial).

6.1 Resultados Numéricos

A Figura 15 mostra o perfil de velocidade para diferentes intervalos de tempo para uma execução da aplicação. Inicialmente, o fluxo está em baixa velocidade e a linha do fluxo é simétrica ao redor do cilindro. Observe que as linhas de fluxo penetram na região do cilindro devido à permeabilidade porosa. O fluido frio é injetado na região de entrada do domínio. Como resultado, o fluido frio se aproxima do cilindro. Um par de vórtices se forma e começa a aumentar com o tempo. O efeito do vórtice é a mistura da região fria e aquecida do escoamento do fluido.

Figura 15 – Tempo de evolução de *streamlines* para Re = 100, $\varepsilon = 0.7$ e $Da = 5.10^{-3}$.



Fonte: (OLIVEIRA, 2020)

6.2 Análise Preliminar da Performance

Um estudo preliminar de performance foi conduzido usando a ferramenta Perf. A Tabela 4 apresenta as 9 rotinas que mais demandam tempo na execução da aplicação. `solve_u`, `solve_v`, `solve_p`, e `solve_z` são essencialmentes usadas para chamar outras rotinas, incluindo as que mais gastam tempo de execução como `resv`, `resu`, `resz`, `upwind_v` e `upwind_u`.

Tabela 4 – Análise de desempenho com a ferramenta Perf

Rotina	% de tempo
resv	42,53
resu	41,28
resz	7,21
upwind_v	2,60
upwind_u	1,80
solve_p	1,25
solve_u	1,10
solve_v	1,10
solve_z	0,45

Fonte: Autor

6.3 Avaliação de Desempenho

A Tabela 5 apresenta o tempo médio de execução, a variância e o desvio padrão para três tamanhos de malha. A média da soma do tempo de pré e pós processamento para cada uma das malhas foi de $0,122$s, $0,349$s e $1,271$s, isto é ele cresce a medida que aumenta o tamanho da malha. Assim, pode-se concluir que a maior parte do tempo de execução é gasta na etapa iterativa.

Pode-se observar na Tabela 5 que a variância obtida para 8 *threads* OpenMP foi, para a maioria das malhas, maior do que todas as outras *threads*, não sendo maior somente que a variância obtida para 4 *threads* na malha 200x249. Pode-se observar também, que a medida em que o tamanho da malha cresce, a variância também cresce.

Conforme pode ser observado na Tabela 5, a implementação paralela do OpenMP fornece redução do tempo de execução. Para todos os experimentos, o uso de mais *threads* reduz continuamente o tempo de execução, incluindo quando o *hyperthreading* está ativo (32 *threads* OMP).

O uso de OpenACC na GPU também proporciona redução do tempo. A diferença entre os melhores resultados de OpenMP e OpenACC aumenta à medida que tamanhos de malha maiores são usados. Para a malha 200x249, o *speedup* obtido com OpenACC é o dobro do obtido com OpenMP.

Tabela 5 – Performance: Tempo, variância e desvio padrão da execução

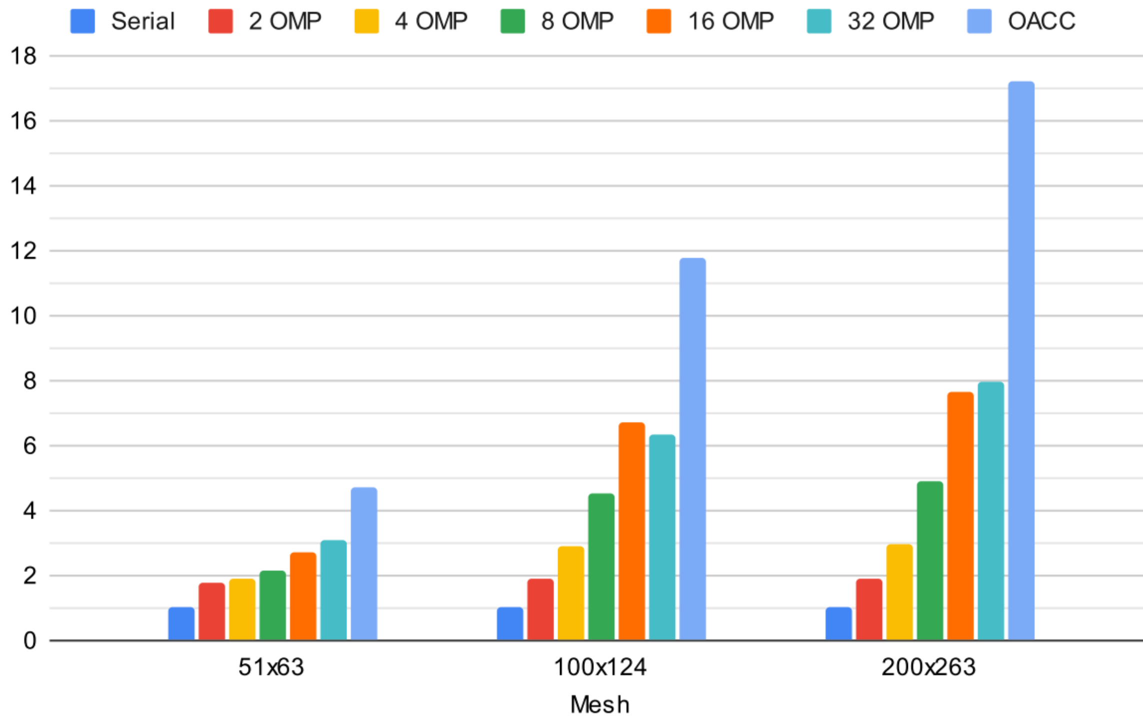
Malha	Configuração		Tempo(s)	σ^2	σ
51x63	Sequencial		301,79	0,19	0,44
	OpenMP	2 threads	167,52	0,13	0,36
		4 threads	156,05	3,79	1,95
		8 threads	138,33	35,39	5,95
		16 threads	110,22	13,72	3,70
		32 threads	97,61	2,23	1,49
	OpenACC		64,14	0,50	0,71
100x124	Sequencial		1231,71	122,34	11,06
	OpenMP	2 threads	655,81	13,07	3,62
		4 threads	421,72	11,39	3,37
		8 threads	271,25	29,58	5,44
		16 threads	182,96	13,17	3,63
		32 threads	194,87	3,06	1,75
	OpenACC		104,41	0,11	0,32
200x249	Sequencial		4830,54	458,67	21,42
	OpenMP	2 threads	2532,63	49,13	7,01
		4 threads	1632,77	323,11	17,98
		8 threads	979,65	138,81	11,78
		16 threads	628,77	83,82	9,16
		32 threads	605,18	66,71	8,17
	OpenACC		280,89	0,12	0,35

Fonte: Autor

A Figura 16 apresenta o *speedup* relacionado à execução sequencial para as interfaces OpenMP e OpenACC. StarPU não está presente na figura devido à não apresentar *speedup*. Os resultados mostram que quanto maior a malha, melhor é o *speedup*. Para OpenMP, o *speedup* ficou em torno de 8, usando 32 *threads* OpenMP, e 17,3 para o OpenACC.

O uso da interface StarPU não proporcionou redução no tempo de execução. A Tabela 6 mostra os tempos, a variância e o desvio padrão obtidos das rotinas, onde a execução da aplicação na forma sequencial é representada por 1 core–tarefa. A coluna cores–tarefas indica o número de cores e o número máximo de tarefas simultâneas, onde tenta-se manter um balanceamento de 1 para 1, uma tarefa por core. A última coluna, $\times$ serial, aponta a quantidade de vezes que determinada configuração de teste ficou mais lenta em relação a versão serial da aplicação. A rotina `solve_u` paralelizada com StarPU ficou, em média, 2,6 vezes mais lenta utilizando 4 cores, sendo o melhor aproveitamento obtido. Utilizando 16 cores, a rotina paralelizada ficou, em média, 12,4 vezes mais lenta. As demais rotinas, `solve_v`, `solve_p` e `solve_z`, ficaram, em média, 2,4, 12,5, e 6,2 vezes mais lentas utilizando 4 e 2 cores, sendo o melhor aproveitamento obtido. Utilizando 16 cores, as rotinas paralelizadas ficaram, em média, 8,2, 49,3 e 50,5 vezes mais lentas.

Figura 16 – *Speedup* relacionado à execução sequencial.



Fonte: Autor

Observou-se que, a medida que a malha aumenta, o tempo da aplicação paralelizada com StarPU fica menor, ou seja, a aplicação fica mais rápida. Existem algumas exceções, como por exemplo, utilizando 16 tarefas e malha 400x499 para as rotinas `solve_u` e `solve_v`. Também pode-se observar que a variância obtida se manteve baixa, sendo o maior valor obtido $6,774.10^{-03}$.

Tabela 6 – Performance: Tempo, variância e desvio padrão das rotinas paralelizadas com StarPU

Rotina	**Malha**	**Cores** **Tarefas**	**Tempo (ms)**	σ^2	σ	$\times$ **serial**
	51x63	1	0,673	$4{,}660.10^{-10}$	$2{,}159.10^{-05}$	-
		2	1,851	$1{,}077.10^{-05}$	$3{,}282.10^{-03}$	2,75
		4	2,719	$1{,}246.10^{-05}$	$3{,}530.10^{-03}$	4,04
		8	4,245	$2{,}187.10^{-05}$	$4{,}676.10^{-03}$	6,31
		16	17,911	$1{,}168.10^{-03}$	$3{,}417.10^{-02}$	26,61
	100x124	1	2,634	$9{,}140.10^{-08}$	$3{,}023.10^{-04}$	-
		2	6,459	$4{,}010.10^{-05}$	$6{,}332.10^{-03}$	2,45
		4	5,821	$3{,}143.10^{-05}$	$5{,}606.10^{-03}$	2,21
		8	8,192	$3{,}710.10^{-05}$	$6{,}091.10^{-03}$	3,11

solve_u		16	23,540	$1{,}021.10^{-03}$	$3{,}196.10^{-02}$	8,94
	200x249	1	10,791	$7{,}884.10^{-07}$	$8{,}879.10^{-04}$	-
		2	27,621	$5{,}014.10^{-05}$	$7{,}081.10^{-03}$	2,56
		4	24,123	$2{,}053.10^{-05}$	$4{,}531.10^{-03}$	2,24
		8	28,102	$1{,}150.10^{-04}$	$1{,}072.10^{-02}$	2,60
		16	71,454	$1{,}467.10^{-03}$	$3{,}831.10^{-02}$	6,62
	400x499	1	51,912	$4{,}254.10^{-06}$	$2{,}062.10^{-03}$	-
		2	126,180	$3{,}417.10^{-05}$	$5{,}846.10^{-03}$	2,43
		4	114,082	$1{,}164.10^{-04}$	$1{,}079.10^{-02}$	2,20
		8	120,565	$2{,}589.10^{-05}$	$5{,}088.10^{-03}$	2,32
		16	383,432	$4{,}488.10^{-03}$	$6{,}699.10^{-02}$	7,39
solve_v	51x63	1	0,893	$3{,}266.10^{-09}$	$5{,}715.10^{-05}$	-
		2	2,325	$1{,}509.10^{-05}$	$3{,}884.10^{-03}$	2,60
		4	2,836	$1{,}209.10^{-05}$	$3{,}477.10^{-03}$	3,18
		8	4,122	$8{,}884.10^{-06}$	$2{,}981.10^{-03}$	4,62
		16	14,153	$4{,}338.10^{-04}$	$2{,}083.10^{-02}$	15,85
	100x124	1	3,528	$8{,}315.10^{-08}$	$2{,}884.10^{-04}$	-
		2	8,866	$4{,}794.10^{-05}$	$6{,}924.10^{-03}$	2,51
		4	9,110	$4{,}661.10^{-05}$	$6{,}827.10^{-03}$	2,58
		8	10,207	$3{,}046.10^{-05}$	$5{,}519.10^{-03}$	2,89
		16	22,274	$3{,}309.10^{-04}$	$1{,}819.10^{-02}$	6,31
	200x249	1	14,368	$9{,}030.10^{-07}$	$9{,}503.10^{-04}$	-
		2	34,898	$3{,}708.10^{-05}$	$6{,}090.10^{-03}$	2,43
		4	30,488	$1{,}119.10^{-04}$	$1{,}058.10^{-02}$	2,12
		8	32,967	$1{,}093.10^{-04}$	$1{,}046.10^{-02}$	2,29
		16	59,967	$5{,}620.10^{-04}$	$2{,}371.10^{-02}$	4,17
	400x499	1	64,721	$5{,}369.10^{-06}$	$2{,}317.10^{-03}$	-
		2	158,519	$3{,}225.10^{-05}$	$5{,}679.10^{-03}$	2,45
		4	136,174	$1{,}332.10^{-04}$	$1{,}154.10^{-02}$	2,10
		8	143,099	$3{,}880.10^{-04}$	$1{,}970.10^{-02}$	2,21
		16	408,815	$4{,}916.10^{-03}$	$7{,}011.10^{-02}$	6,32
	51x63	1	0,033	$6{,}477.10^{-12}$	$2{,}545.10^{-06}$	-
		2	0,772	$5{,}845.10^{-06}$	$2{,}418.10^{-03}$	23,39
		4	0,764	$6{,}892.10^{-06}$	$2{,}625.10^{-03}$	23,15
		8	1,235	$1{,}056.10^{-05}$	$3{,}250.10^{-03}$	37,42
		16	3,330	$9{,}363.10^{-05}$	$9{,}676.10^{-03}$	100,91
	100x124	1	0,169	$1{,}315.10^{-10}$	$1{,}147.10^{-05}$	-
		2	1,858	$2{,}053.10^{-05}$	$4{,}531.10^{-03}$	10,99
		4	2,304	$2{,}908.10^{-05}$	$5{,}393.10^{-03}$	13,63

solve_p		8	3,057	$3{,}691.10^{-05}$	$6{,}076.10^{-03}$	18,09
		16	5,180	$8{,}125.10^{-05}$	$9{,}014.10^{-03}$	30,65
	200x249	1	0,723	$3{,}185.10^{-08}$	$1{,}785.10^{-04}$	-
		2	7,274	$5{,}921.10^{-05}$	$7{,}695.10^{-03}$	10,06
		4	10,479	$1{,}381.10^{-04}$	$1{,}175.10^{-02}$	14,49
		8	17,794	$3{,}434.10^{-04}$	$1{,}853.10^{-02}$	24,61
		16	30,577	$8{,}158.10^{-04}$	$2{,}856.10^{-02}$	42,29
	400x499	1	6,222	$7{,}247.10^{-07}$	$8{,}513.10^{-04}$	-
		2	30,970	$1{,}682.10^{-05}$	$4{,}101.10^{-03}$	4,98
		4	47,981	$2{,}706.10^{-05}$	$5{,}202.10^{-03}$	7,71
		8	65,099	$3{,}733.10^{-04}$	$1{,}932.10^{-02}$	10,46
		16	134,551	$6{,}774.10^{-03}$	$8{,}230.10^{-02}$	21,63
solve_z	51x63	1	0,176	$6{,}333.10^{-11}$	$7{,}958.10^{-06}$	-
		2	2,396	$6{,}303.10^{-06}$	$2{,}511.10^{-03}$	13,61
		4	3,670	$6{,}732.10^{-06}$	$2{,}595.10^{-03}$	20,85
		8	6,468	$6{,}938.10^{-06}$	$2{,}634.10^{-03}$	36,75
		16	20,899	$2{,}113.10^{-04}$	$1{,}454.10^{-02}$	118,74
	100x124	1	0,746	$1{,}544.10^{-08}$	$1{,}243.10^{-04}$	-
		2	3,946	$2{,}410.10^{-05}$	$4{,}909.10^{-03}$	5,29
		4	6,081	$1{,}756.10^{-05}$	$4{,}191.10^{-03}$	8,15
		8	12,205	$2{,}766.10^{-05}$	$5{,}259.10^{-03}$	16,36
		16	33,120	$4{,}714.10^{-04}$	$2{,}171.10^{-02}$	44,40
	200x249	1	3,130	$4{,}636.10^{-08}$	$2{,}153.10^{-04}$	-
		2	9,414	$5{,}161.10^{-05}$	$7{,}184.10^{-03}$	3,01
		4	12,809	$8{,}971.10^{-05}$	$9{,}472.10^{-03}$	4,09
		8	22,188	$1{,}791.10^{-04}$	$1{,}338.10^{-02}$	7,09
		16	61,513	$8{,}195.10^{-04}$	$2{,}863.10^{-02}$	19,65
	400x499	1	19,645	$1{,}382.10^{-06}$	$1{,}176.10^{-03}$	-
		2	48,827	$1{,}129.10^{-05}$	$3{,}359.10^{-03}$	2,49
		4	47,878	$6{,}409.10^{-05}$	$8{,}005.10^{-03}$	2,44
		8	67,157	$3{,}454.10^{-04}$	$1{,}859.10^{-02}$	3,42
		16	293,224	$6{,}446.10^{-03}$	$8{,}029.10^{-02}$	14,93

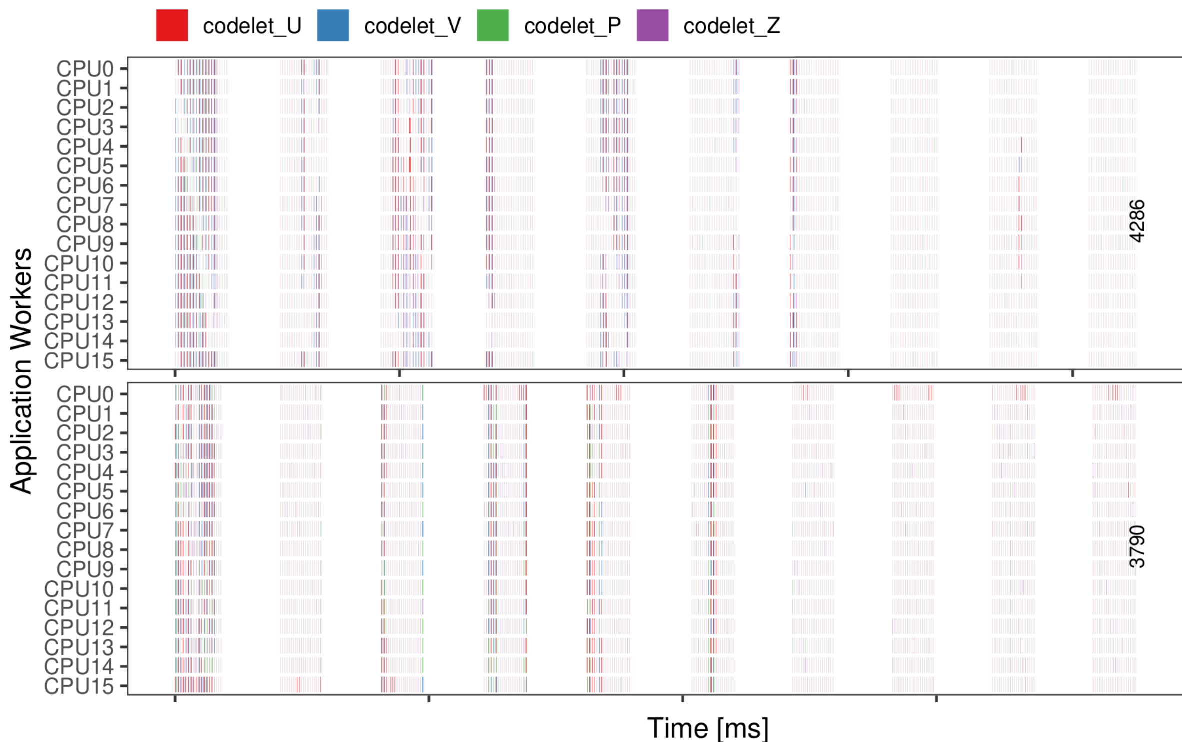
Fonte: Autor

A razão dos resultados não estarem bons pode estar associado à baixa carga de trabalho da aplicação. Figura 17, Figura 18, Figura 19 e Figura 20 apresentam alguns dados gerados com StarVZ a partir da execução paralela com StarPU. Essas figuras apre-

sentam resultados para dois casos: utilizando barreira entre as rotinas e não utilizando. A Figura 17 apresenta a disposição das tarefas de cada rotina, paralelizadas com 16 tarefas cada, onde cada rotina, `solve_u`, `solve_v`, `solve_p` e `solve_z`, possui um codelet associado, `codelet_U`, `codelet_V`, `codelet_P` e `codelet_Z`. Pode-se notar que as tarefas são representadas por linhas muito finas, o que permite concluir que não há volume suficiente de computação nas rotinas para que sejam divididas em muitas tarefas. A Figura 18 apresenta também a disposição das tarefas, porém as tarefas do *runtime*, desconsiderando os *codelets* de origem. Além das tarefas possuírem pouca carga, há um vão entre entre elas. Isso significa que existe computação entre as rotinas paralelizadas que demanda algum tempo e que não foi paralelizada. Figura 19 mostra as tarefas submetidas e Figura 20 as tarefas prontas.

Não utilizar barreiras entre as rotinas deixa a aplicação mais rápida. Pode-se concluir isso com base na Figura 19 e Figura 20, onde temos mais tarefas submetidas e tarefas prontas. Porém, retirar as barreiras entre as rotinas nos trouxe perda de precisão da aplicação, onde os resultados gerados não eram mais condizentes com o esperado.

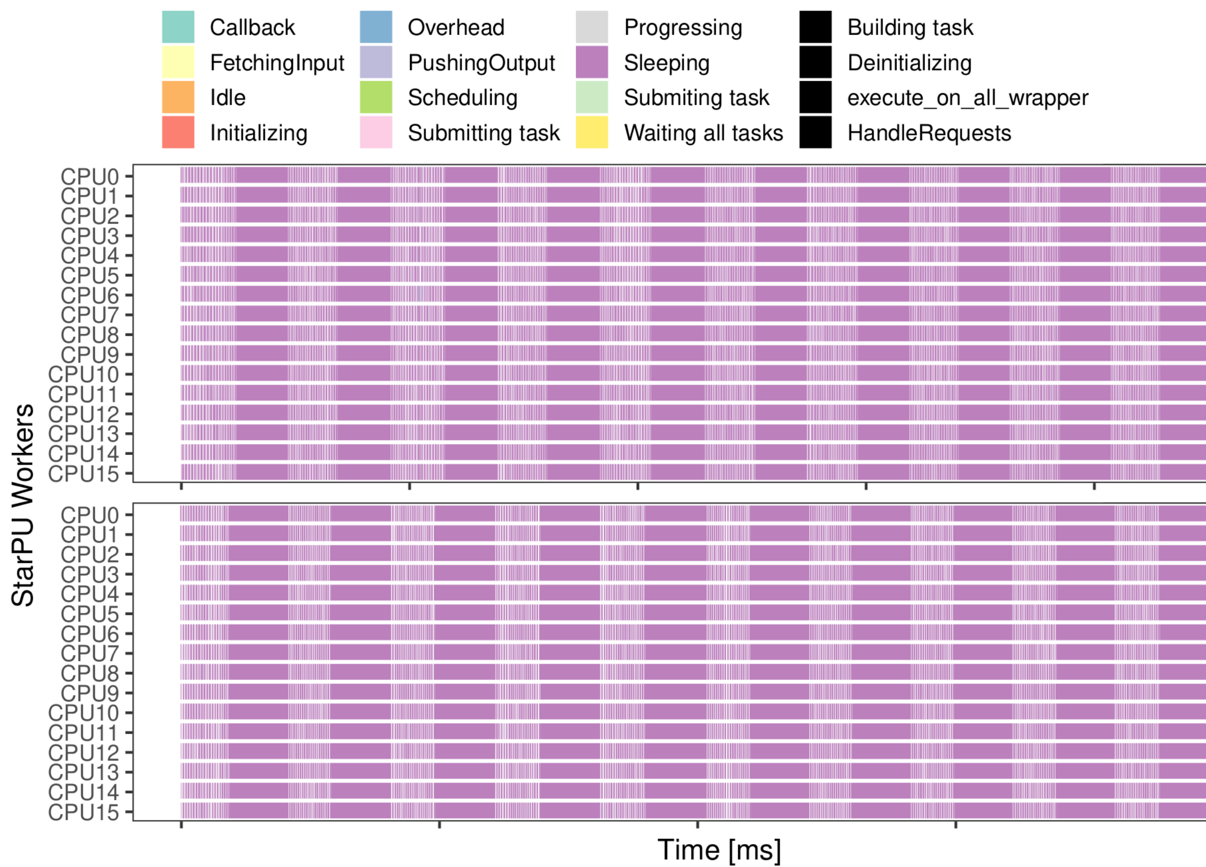
Figura 17 – Disposição das tarefas dos *codelets* nos cores: **com** x **sem** barreira entre as rotinas.



Fonte: Autor

Além de não obter diminuição no tempo da aplicação, obteve-se perda na precisão dos resultados. Essa perda surgiu já na passagem da alocação das matrizes de 2D para 1D, e com a paralelização com StarPU ela aumentou. Ela surge na 1ª iteração em casas

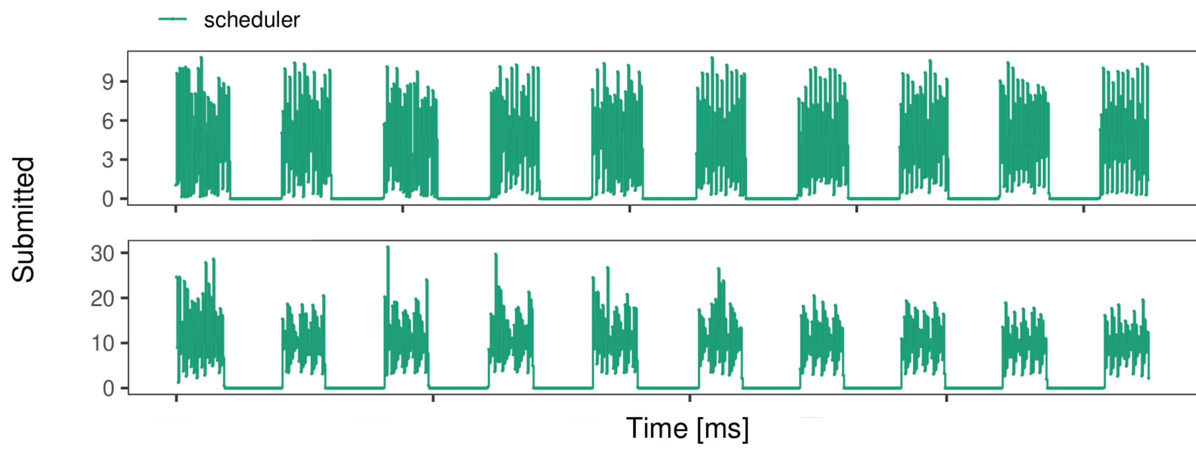
Figura 18 – Disposição das tarefas do *runtime* nos cores: **com** x **sem** barreira entre as rotinas.



Fonte: Autor

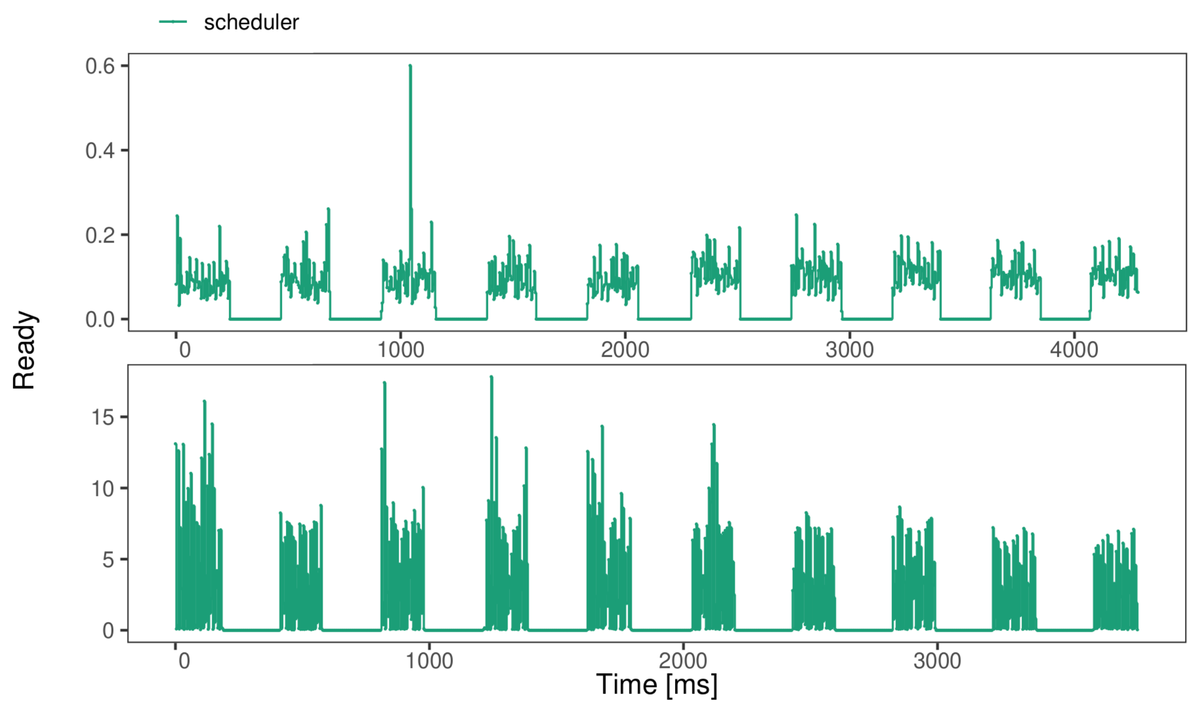
decimais bem pequenas, porém, a medida que as iterações da aplicação decorrem, esse erro cresce. Algumas suposições são feitas, como a disposição das matrizes, para o erro na passagem de 2D para 1D, e o domínio da aplicação, para o erro na Paralelização com StarPU.

Figura 19 – Tarefas submetidas: **com** x **sem** barreira entre as rotinas.



Fonte: Autor

Figura 20 – Tarefas prontas: **com** x **sem** barreira entre as rotinas.



Fonte: Autor

7 CONSIDERAÇÕES FINAIS

Neste trabalho de conclusão de curso utilizou-se um modelo de aplicação para o problema de meio aberto-poroso acoplado. O modelo pode ser aplicado à secagem de grãos, problema esse presente na região devido a produção de culturas como arroz, milho e soja.

Alguns testes foram realizados para comprovar o desempenho da aplicação. Se percebeu que a aplicação pode ser executada mais eficientemente se forem exploradas implementações paralelas. Para tanto, foram projetadas versões paralelas eficientes usando OpenMP, para ambientes multicore, e OpenACC, para ambientes GPU. Além dessas duas interfaces amplamente utilizadas, StarPU foi adotado para ambientes multicore para avaliar o desempenho do paradigma orientado à tarefas numa aplicação desse tipo.

Foram fornecidos testes de desempenho que mostram a escalabilidade da implementação para diferentes tamanhos de *threads* e aumento do *speedup* para malhas maiores com OpenMP e OpenACC. A implementação da GPU proporcionou melhores resultados em relação à CPU. O tempo total de simulação foi 8 vezes menor para uma arquitetura multicore usando OpenMP e 17,3 vezes menor com uma única GPU usando OpenACC.

Os testes com a interface StarPU não apontaram redução no tempo de execução das rotinas. Feita uma análise minuciosa, pode-se concluir que a quantidade de computação em cada rotina paralelizada não foi suficiente para justificar os custos de criação das tarefas e gestão delas pelo *runtime*.

Esse trabalho obteve aprovação em dois grandes eventos internacionais de computação paralela: *Latin America High Performance Computing* (CARLA 2021) e *Euromicro International Conference on Parallel, Distributed and Network-based Processing* (PDP 2022). Também foram aceitos resumos no Salão Internacional de Ensino Pesquisa e Extensão da Unipampa (SIEPE 2021) e Escola Regional de Alto Desempenho/Região Sul (ERAD/RS 2022).

As mesmas interfaces de programação paralela podem continuar sendo exploradas em trabalhos futuros. Por exemplo, OpenMP pode ser usado para GPUs de acordo com as especificações mais recentes da interface. Também, pode-se estudar meios para que o desempenho com StarPU se torne aceitável, como avaliar melhor questões de domínio dos dados e/ou abordar somente as rotinas mais custosas, tais como `RESU`, bem como utilizar outras políticas de escalonamento, como escalonadores de tarefas que utilizem modelos de performance, por exemplo, `dmda`. Além disso, a causa do erro numérico nas saídas com StarPU deve ser melhor investigada.

REFERÊNCIAS

ALMEIDA, R. A. B. da S. **Solução numérica do escoamento não-isotérmico em reservatórios de óleo pesado empregando computação paralela**. Tese (Doutorado) — Universidade do Estado do Rio de Janeiro - Instituto Politécnico, 2021. Citado na página 29.

BALA, B. K. **Drying and Storage of Cereal Grains**. [S.l.]: Wesley Blackwell, 2017. Citado na página 23.

BRETON, P. L.; CALTAGIRONE, J. P.; ARQUIS, E. Natural convection in a square cavity with thin porous layers on its vertical walls. **Journal of Heat Transfer**, v. 113, n. 4, p. 892 – 898, 1991. Citado na página 33.

CARVALHO, L. de et al. Uma implementação paralelizada via a API OpenMP para a simulação numérica de reservatórios de gás natural. **Revista Brasileira de Computação Aplicada**, v. 12, n. 2, p. 103–121, jun. 2020. Disponível em: <http://seer.upf.br/index.php/rbca/article/view/10158>. Citado na página 29.

CHANDRASEKARAN, S.; JUCKELAND, G. **OpenACC for Programmers: Concepts and Strategies**. 1st. ed. [S.l.]: Addison-Wesley Professional, 2017. ISBN 0134694287. Citado na página 31.

CHAPMAN, B.; MEHROTRA, P.; ZIMA, H. Enhancing openmp with features for locality control. In: CITESEER. **Proc. ECWMF Workshop" Towards Teracomputing-The Use of Parallel Processors in Meteorology**. Austrian: PSU, 1998. Citado na página 31.

CORNELISSEN, P. **Coupled free-flow and porous media flow: a numerical and experimental investigation**. Tese (Doutorado) — Faculty of Geosciences - Utrecht University, 2016. Citado 2 vezes nas páginas 24 e 33.

CRUZ, L. M. de la; MONSIVAIS, D. Parallel numerical simulation of two-phase flow model in porous media using distributed and shared memory architectures. **GeofÃsica internacional**, scielomx, v. 53, p. 59 – 75, 03 2014. ISSN 0016-7169. Disponível em: <http://www.scielo.org.mx/scielo.php?script=sci_arttext&pid=S0016-71692014000100005&nrm=iso>. Citado na página 29.

FAO. **Grain crop drying, handling and storage.** 2010. [*Online*; acessado em 30 de novembro de 2019]. Disponível em: <http://www.fao.org/3/i2433e/i2433e10.pdf>. Citado na página 23.

GIMENES, T. L.; PISANI, F.; BORIN, E. Evaluating the Performance and Cost of Accelerating Seismic Processing with CUDA, OpenCL, OpenACC, and OpenMP. In: **2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2018. p. 399–408. ISSN 1530-2075. Citado na página 27.

GOYAL, A.; LI, Z.; KIMM, H. Comparative Study on Edge Detection Algorithms Using OpenACC and OpenMPI on Multicore Systems. In: **2017 IEEE 11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)**. [S.l.: s.n.], 2017. p. 67–74. Citado na página 28.

GUNAWAN, P. H. Scientific parallel computing for 1D heat diffusion problem based on OpenMP. In: **2016 4th International Conference on Information and Communication Technology (ICoICT)**. [S.l.: s.n.], 2016. p. 1–5. Citado na página 28.

HO, M. T. et al. A multi-level parallel solver for rarefied gas flows in porous media. **Computer Physics Communications**, v. 234, p. 14–25, 01 2019. Citado na página 29.

JULIATI, S.; GUNAWAN, P. H. OpenMP architecture to simulate 2D water oscillation on paraboloid. In: **2017 5th International Conference on Information and Communication Technology (ICoIC7)**. [S.l.: s.n.], 2017. p. 1–5. Citado na página 28.

KASMI, N. et al. Performance evaluation of starpu schedulers with preconditioned conjugate gradient solver on heterogeneous (multi-cpus/multi-gpus) architecture. In: **2017 3rd International Conference of Cloud Computing Technologies and Applications (CloudTech)**. [S.l.: s.n.], 2017. p. 1–6. Citado na página 30.

KROKIDA, M.; MARINOS-KOURIS, D.; MUJUMDAR, A. S. **Handbook of industrial drying. Rotary Drying**. [S.l.]: Taylor & Francis Group, 2006. 151–172 p. Citado na página 23.

LOPES, R. A.; THIBAULT, S.; MELO, A. C. M. A. Masa-starpu: Parallel sequence comparison with multiple scheduling policies and pruning. In: **2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. [S.l.: s.n.], 2020. p. 225–232. Citado na página 30.

MORISHITA, M. et al. Parallelization of GKV benchmark using OpenACC. In: **2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. [S.l.: s.n.], 2021. p. 723–729. Citado na página 28.

NESI, L. L. et al. Introdução ao desenvolvimento de aplicações paralelas com o paradigma orientado a tarefas e o runtime starpu. **Minicursos da XX Escola Regional de Alto Desempenho da Região Sul**, p. 70 – 88, 2020. Citado na página 71.

OLIVEIRA, D. P. de. **Fluid Flow Through Porous Media with the One Domain Approach: A Simple Model for Grains Drying.** Dissertação (Dissertação de Mestrado) — Universidade Federal do Pampa, 2020. Citado 8 vezes nas páginas 23, 24, 34, 35, 36, 37, 41 e 53.

OPENACC. **What is OpenACC?** 2021. [Online; acessed july, 20 2021]. Disponível em: <https://www.openacc.org/>. Citado na página 31.

OPENMP. **The OpenMP API specification for parallel programming**. 2021. Available at: https://www.openmp.org. Disponível em: <https://www.openmp.org/>. Citado na página 31.

RAJ, A. et al. Acceleration of a 3D Immersed Boundary Solver Using OpenACC. In: **2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)**. [S.l.: s.n.], 2018. p. 65–73. Citado na página 28.

RANGO, A. D. et al. Opencal system extension and application to the three-dimensional richards equation for unsaturated flow. **Computers & Mathematics with Applications**, v. 81, 05 2020. Citado na página 29.

SILVA, H. U. et al. Parallel OpenMP and OpenACC Mixing Layer Simulation. In: **Proceedings of Euromicro International Conference on Parallel, Distributed and Network-Based Processing**. [S.l.]: IEEE, 2022. Citado na página 37.

SIMANJUNTAK, C. A.; GUNAWAN, P. H. Computing two-layer SWE for simulating submarine avalanches on OpenMP. In: **2017 International Conference on Control, Electronics, Renewable Energy and Communications (ICCREC)**. [S.l.: s.n.], 2017. p. 190–195. Citado na página 28.

STARPU. **A Unified Runtime System for Heterogeneous Multicore Architectures**. 2018. [Online; acesso 9-Setembro-2020]. Disponível em: <https://starpu.gitlabpages.inria.fr/>. Citado na página 32.

Université de Bordeaux; CNRS (LaBRI UMR 5800); Inria. **StarPU Handbook**. [S.l.], 2021. Disponível em: <https://files.inria.fr/starpu/doc/starpu.pdf>. Citado 3 vezes nas páginas 71, 79 e 81.

VERSTEEG, H. K.; MALALASEKERA, W. **An introduction to computational fluid dynamics: the finite volume method**. [S.l.]: Pearson Education, 2007. Citado 2 vezes nas páginas 33 e 35.

Apêndices

APÊNDICE A – STARPU

A.1 Paradigma de Programação Orientado a Tarefas

O paradigma de programação orientado a tarefas consiste na divisão de blocos de processamento em tarefas que, para gerar otimização, são executadas em paralelo (NESI et al., 2020). Para que essa execução possa ocorrer paralelamente, todas e quaisquer dependências de dados devem ser eliminadas. Caso contrário, as tarefas serão executadas sequencialmente, podendo até piorar o tempo de execução. O gerenciamento dessas tarefas fica a cargo do *runtime*, bastando apenas que o programador as submeta sequencialmente.

Cada unidade de processamento (*core*), seja da CPU ou GPU, é considerado como um trabalhador pelo StarPU. Para cada trabalhador é atribuída uma tarefa, sem preempção. As tarefas podem ter múltiplas implementações, em CPU e GPU, e o *runtime* decide qual será a melhor implementação para determinada tarefa. Cada uma das tarefas obrigatoriamente deve estar associada a um *codelet*. *Codelets* são estruturas de StarPU (`struct starpu_codelet`) que descrevem como aquela tarefa será executada.

A.2 Introduzindo StarPU em CPU

Existem funções que devem, obrigatoriamente, ser chamadas no corpo de uma aplicação StarPU (Université de Bordeaux; CNRS (LaBRI UMR 5800); Inria, 2021). No Código 7 pode-se ver o esqueleto de uma aplicação simples já conhecida onde se imprime na tela a frase "Hello_World!". Esse código mostra a estrutura básica que deve ser utilizada por uma aplicação. No corpo do código (*main*) estão presentes, em ordem, todas as funções que devem ser chamadas. Basicamente, deve-se inicializar o *runtime*, submeter as tarefas e esperar até que elas fiquem prontas e, por último, encerrar o *runtime*. A Tabela 7 apresenta as principais funções utilizadas nesse trabalho e faz uma breve descrição sobre cada uma. Já a Tabela 8 apresenta as estuturas auxiliares que são utilizadas pelas funções da API.

As funções que serão atribuídas a uma tarefa devem receber, por *default*, dois parâmetros, nessa ordem: *array* de ponteiros do tipo *void*, que é responsável por guardar os ponteiros dos *handles*; ponteiro do tipo *void*, que carrega os argumentos. Como meio para se passar mais de um parâmetro, pode-se ciar uma estrutura auxiliar, como também utilizar vetores ou matrizes. Os nomes dessas variáveis, como o nome da função, é irrelevante, ou seja, pode-se nomear como bem entende-se.

A estrutura `starpu_codelet` possui variáveis que definem informações a respeito do *codelet*. Entre as linhas 11 e 13 do Código 7 são apresentadas algumas delas. São elas: `cpu_funcs`, que guarda as funções destinadas a executar na CPU; `nbuffers`, que guarda o número de *buffers* que serão utilizados; e `name`, que guarda o nome daquele *codelet*.

Para explicar melhor alguns conceitos e formas de se trabalhar com essa API, trazse um exemplo de aplicação simples e muito conhecida: multiplicação de matrizes. Nessa

Tabela 7 – Funções da API StarPU utilizadas

Função	Descrição
`starpu_init()`	Inicializa o *runtime*. Pode receber `NULL` como parâmetro e usar a configuração *default* ou receber um ponteiro da estrutura `starpu_conf` com as configurações desejadas.
`starpu_shutdown()`	Encerra o *runtime*.
`STARPU_MATRIX_GET_PTR()`	Recupera o ponteiro do *handle* definido como matriz.
`STARPU_MATRIX_GET_NX()`	Retorna a quantidade de colunas de uma matriz.
`STARPU_MATRIX_GET_NY()`	Retorna a quantidade de linhas de uma matriz.
`starpu_matrix_data_register()`	Registra uma matriz na memória.
`starpu_task_insert()`	Cria, configura e submete uma tarefa. Necessita que seja passado como parâmetros as configurações da tarefa, como o *codelet* associado e os *data handles*.
`starpu_task_create()`	Cria uma tarefa com as configurações *default* e retorna um ponteiro do tipo `starpu_task`.
`starpu_task_submit()`	Submete uma tarefa. Recebe como parâmetro um ponteiro do tipo `starpu_task`.
`starpu_data_unregister()`	Libera os dados registrados na memória.
`starpu_data_partition()`	Particiona um *data handle* de acordo com um filtro. Esse filtro é do tipo `starpu_data_filter`.
`starpu_data_map_filters()`	Mapeia filtros do tipo `starpu_data_filter` em um *data handle*. É equivalente a aplicação de mais de um filtro num *handle*.
`starpu_data_get_sub_data()`	Retorna um ponteiro para uma parte específica de um *data handle* que foi particionado.
`starpu_data_unpartition()`	Remove particionamento de um *data handle*.

Fonte: Autor

aplicação precisa-se de três *buffers*, sendo dois com acesso de leitura e um com acesso de escrita. Pode-se definir os modos de acesso aos *buffers* com o atributo `modes` na estrutura `starpu_codelet`. O trecho apresentado no Código 8 mostra o *codelet* referente a essa aplicação.

Para acessar esses *buffers* na função definida no *codelet* usa-se funções fornecidas pela API. Ela fornece funções específicas para cada tipo de dado. Como o exemplo apresentado trabalha com matrizes, deve-se utilizar a função `STARPU_MATRIX_GET_PTR()`. Ela fornece ainda outras funções auxiliares, como funções que te permitem saber a dimensão de uma matriz.

Antes de se pensar em acessar esses *buffers*, eles precisam ser registrados. O trecho presente no Código 9 apresenta o registro das matrizes que serão utilizadas na aplicação. Já os códigos 10 e 11 apresentam duas formas equivalentes de se criar, configurar e sub-

Tabela 8 – Estruturas da API StarPU utilizadas

Estrutura	Descrição
`starpu_codelet`	Permite definir as configurações de um *codeldet*.
`starpu_data_handle_t`	Tipo de dado que a interface trabalha. Todos os dados devem – a não ser parâmetros simples como um número, uma letra, etc – devem ter um *handle* associado.
`starpu_task`	Permite definir alterar as configurações de uma task antes de submetê-la.
`starpu_data_filter`	Permite criar filtros para aplicação em *data handles* grandes. São utilizados dois tipos: `starpu_matrix_filter_vertical_block`(divide a matriz verticalmente) e `starpu_matrix_filter_block`(divide a matriz horizontalmente).
`starpu_conf`	Permite definir as configurações do *runtime*.

Fonte: Autor

Código 7 – Esqueleto da aplicação Hello_World.c em StarPU

```
1  #include <stdlib.h>
2  #include <limits.h>
3  #include <starpu.h>
4
5  void func_cpu(void *buffers[], void *args){
6      printf("Hello_World!");
7  }
8
9  struct starpu_codelet codelet_world =
10 {
11     .cpu_funcs = { func_cpu },
12     .nbuffers = 0,
13     .name = "hello_world"
14 };
15
16 int main(void){
17     starpu_init(NULL);
18     starpu_task_insert(&codelet_world, 0);
19     starpu_task_wait_for_all();
20     starpu_shutdown();
21 }
```

Fonte: Autor

meter uma tarefa.

As tarefas podem ser configuradas de duas formas, como podemos ver nos códigos 10 e 11. No Código 10 cria-se uma tarefa com a configuração padrão(linha 2), configura-se como desejado (linhas 4-8) e, por último, se submete (linha 11). Já no Código 11 a criação, configuração e submissão da tarefa são feitas todas em uma chamada de função.

Após a submissão de todas as tarefas precisa-se liberar a memória registrada. Diferentemente de outras interfaces, como CUDA, não é necessário carregar o dado antes

Código 8 – Codelet de uma multiplicação simples em CPU de matrizes

```
struct starpu_codelet codelet =
{
    .cpu_funcs = { func_cpu },
    .nbuffers = 3,
    .modes = { STARPU_R, STARPU_R, STARPU_W },
    .name = "mult"
};
```

Fonte: Autor

Código 9 – Definição e submissão de uma *task*

```
starpu_data_handle_t Ah, Bh, Ch;

starpu_matrix_data_register(&Ah, STARPU_MAIN_RAM,
    (uintptr_t)A, N, N, N, sizeof(int));
starpu_matrix_data_register(&Bh, STARPU_MAIN_RAM,
    (uintptr_t)B, N, N, N, sizeof(int));
starpu_matrix_data_register(&Ch, STARPU_MAIN_RAM,
    (uintptr_t)C, N, N, N, sizeof(int));
```

Fonte: Autor

Código 10 – Criação, configuração e submissão de uma tarefa dividida em partes

```
struct starpu_task* task =
    starpu_task_create();

task->cl = &codelet;

task->handles[0] = Ah;
task->handles[1] = Bh;
task->handles[2] = Ch;

int task_submit =
    starpu_task_submit(task);
```

Código 11 – Criação, configuração e submissão de uma tarefa em uma chamada de função

```
starpu_task_create(&codelet,
    STARPU_R, Ah,
    STARPU_R, Bh,
    STARPU_W, Ch,
    0);
```

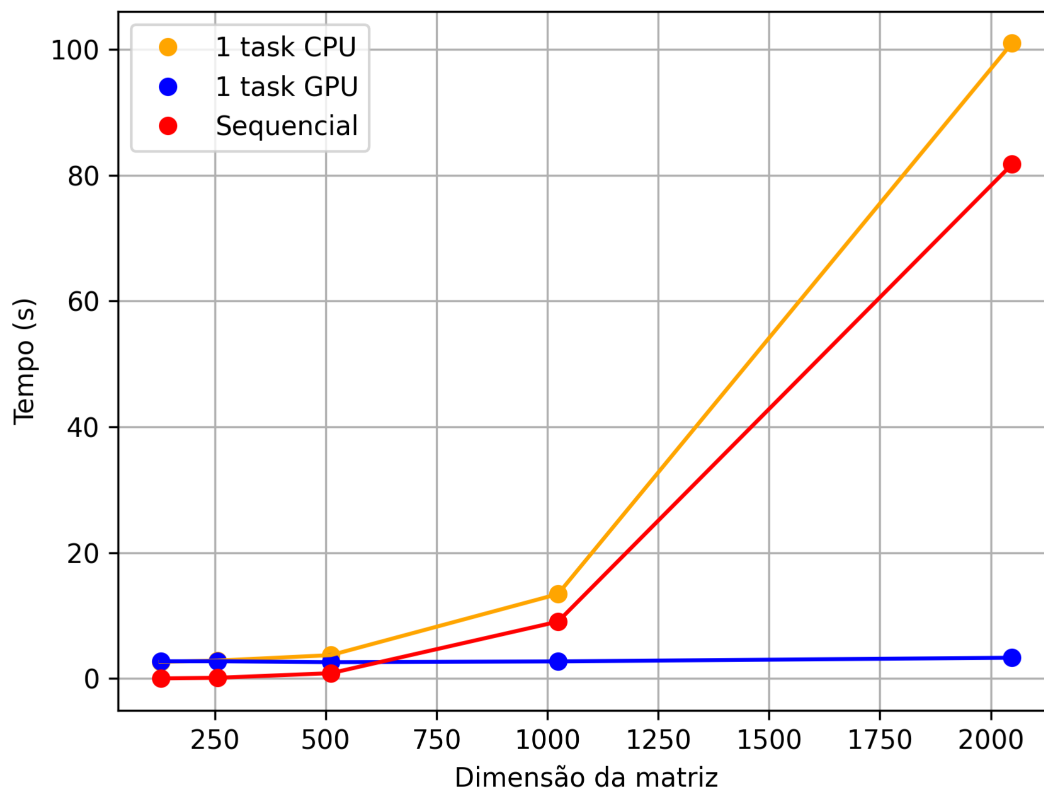
Fonte: Autor

registrado na memória para a variável de origem.

Com as definições apresentadas até aqui já seria possível executar uma multiplicação de matrizes utilizando o *runtime* com uma tarefa. A Figura 21 apresenta um gráfico onde se consegue comparar o tempo da aplicação sequencial e da aplicação gerenciada pelo *runtime*. Esse resultado pode parecer estranho visto que a aplicação sequencial se mostra mais eficiente do que a aplicação gerenciada pelo *runtime*. A explicação para essa piora nos resultados se baseia no fato de que, mesmo com o uso do *runtime*, a aplicação ainda é sequencial, visto que temos a criação de somente uma tarefa. Assim, somado

ao tempo de execução da própria aplicação ainda temos o tempo de gerenciamento do *runtime*, como criação de tarefas, registro de *data handles*, etc.

Figura 21 – Tempo de execução de multiplicação de matrizes utilizando 1 tarefa



Fonte: Autor

Para resolver o problema observado na Figura 21 pode-se aumentar o número de tarefas, dividindo a carga da aplicação entre elas. Deve-se atentar ao fato de que não basta somente aumentar o número de tarefas. Quando tarefas trabalham em cima do mesmo *data handle* pode ocorrer dependência de dados, acarretando na execução sequencial das tarefas. Para solucionar isso pode-se aderir a uma técnica já conhecida: dividir os dados em partes, ou seja, particioná-los.

A.3 Particionamento de Dados

Essa técnica surge como uma solução ao problema da dependência de dados. Ela consegue isolar dados dentro de uma mesma estrutura. Com ela, pode-se dividir os dados e, para cada parte, uma tarefa é escalada para a manipular. Ou seja, tornar os dados independentes.

A Equação 1 apresenta duas matrizes de dimensão 4×4. Elas serão utilizadas para demostrar como o particionamento funciona na prática em uma simples multiplicação de matrizes – seguindo o exemplo apresentado na seção A.2.

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}_{4\times4}, B = \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix}_{4\times4} \tag{1}$$

Ao dividi-las em duas partes/*slices* obtém-se as matrizes mostradas na Equação 2. Pode-se dividir essas matrizes em no máximo 4 partes – número de linhas de A e de colunas em B. Elas não estão divididas em partes iguais, ou seja, a matriz A está dividida horizontalmente enquanto a matriz B está verticalmente. Com a Figura 23 pode-se fazer uma análise do porquê disso. Os blocos das matrizes particionadas são destinados para posições da matriz resultante. Vale ressaltar que a quantidade de partes em que A e B são divididas não precisa ser a mesma.

$$A = \begin{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \end{bmatrix}_{2\times4} \\ \begin{bmatrix} a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}_{2\times4} \end{bmatrix}_{2x1}, B = \begin{bmatrix} \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \\ b_{20} & b_{21} \\ b_{30} & b_{31} \end{bmatrix}_{4\times2} & \begin{bmatrix} b_{02} & b_{03} \\ b_{12} & b_{13} \\ b_{22} & b_{23} \\ b_{32} & b_{33} \end{bmatrix}_{4\times2} \end{bmatrix}_{1\times2} \tag{2}$$

Podemos reescrever essas matrizes como:

$$A = \begin{bmatrix} A_{00} \\ A_{10} \end{bmatrix}_{2\times1}, B = \begin{bmatrix} B_{00} & B_{01} \end{bmatrix}_{1\times2} \tag{3}$$

, onde as submatrizes de A possuem dimensão 2×4 e as submatrizes de B possuem dimensão 4×2.

Com isso, a matriz $C = A \times B$ torna-se:

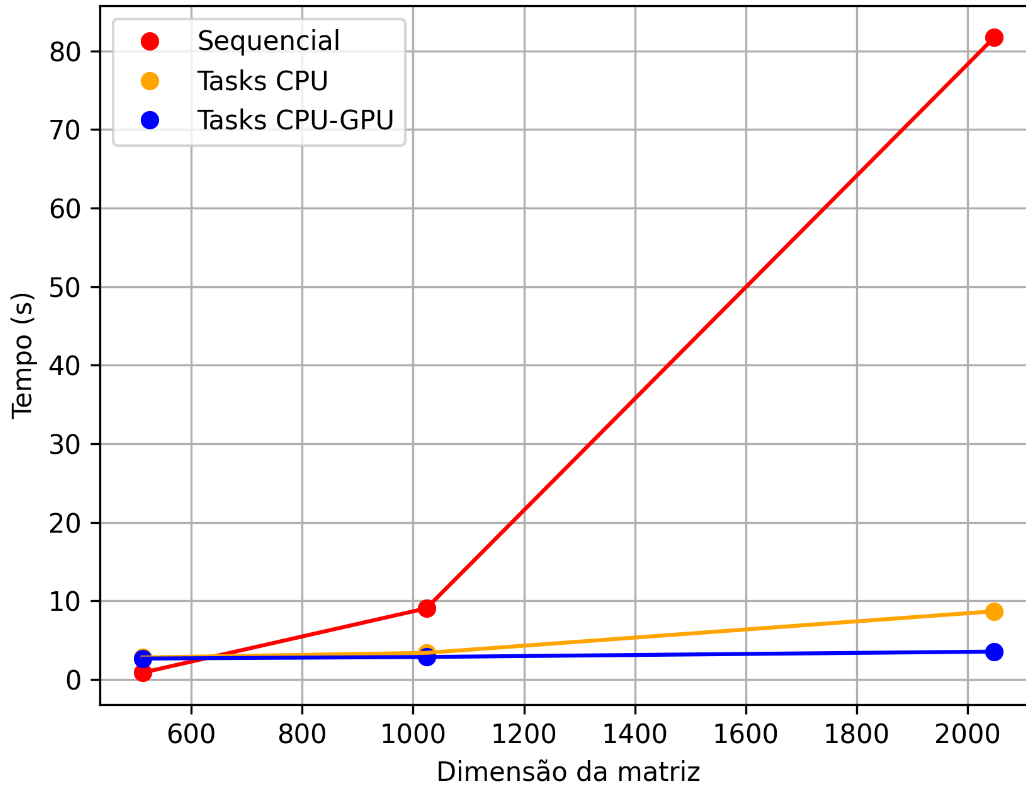
$$C = \begin{bmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{bmatrix}_{2\times2} \tag{4}$$

, onde cada par/submatriz AB possui dimensão 2×2.

Agora, basta criar 4 tarefas, uma para cada submatriz de C. Se as matrizes fossem dividas no maior número de partes possível, poderia se ter, no máximo, n^2 tarefas. Assim, para cada elemento da matriz resultante, existira uma tarefa para calcular seu valor.

Sabe-se que essa técnica, de dividir as matrizes em *slices*, por si só já reduziria o tempo da multiplicação de matrizes. Eliminada a dependência de dados, pode-se desfrutar do poder do *runtime*. A Figura 22 nos mostra a diferença entre os três tempos de execução: somente com particionamento, com particionamento e com o gerenciamento do *runtime*, e sequencial.

Figura 22 – Tempo de execução de multiplicação de matrizes utilizando 16 *slices*



Fonte: Autor

As equações acima mostram como o particionamento funciona para matrizes. Sabemos que também podemos representar matrizes como vetores unidimensionais. Podendo, de alguma forma, servir como uma otimização da aplicação descrita acima, as equações a seguir apresentam um esquema de particionamento para matrizes representadas como vetores.

A Equação 5 apresenta A e B na forma vetorial. Nela, w representa o números de linhas de A, y representa o números de colunas em A e linhas em B e z o número de colunas em B. Pode-se dividir esses vetores em s *slices*, como mostra a Equação 6. Pode-se, ainda, representar essas partes na forma matricial, como mostra a Equação 7.

$$A = [w \times y], B = [y \times z] \tag{5}$$

$$\begin{gathered} A_1 = [\tfrac{w}{s} \times y], A_2 = [\tfrac{w}{s} \times y], ..., A_s = [\tfrac{w}{s} \times y] \\ B_1 = [y \times \tfrac{z}{s}], B_2 = [y \times \tfrac{z}{s}], ..., B_s = [y \times \tfrac{z}{s}] \end{gathered} \tag{6}$$

$$A = [s][\frac{w}{s} \times y], B = [s][y \times \frac{z}{s}] \tag{7}$$

Até aqui apresentou-se como o particionamento de dados funciona na prática. Em StarPU esse particionamento é realizado através da aplicação de filtros diretamente nos *data handles*.

A.3.1 Aplicando Filtros nos Dados

Pode-se definir filtros usando a estrutura `starpu_data_filter`, onde defini-se a função que fará a filtragem e o número de filhos que se quer. Depois de definir os filtros, usa-se a função `starpu_data_partition()` para particionar os dados, onde se recebe o *handle* e o ponteiro do filtro definido. Para os casos onde se deseja aplicar mais de um filtro, pode-se usar a função `starpu_data_map_filters()`, que recebe como parâmetro o *handle*, o número de filtros e os ponteiros deles. O Código 12 apresenta a aplicação de filtros nos *handles* Ah, Bh e Ch. Quando se usa partições não é possível liberar os dados registrados diretamente. Antes de liberá-los é preciso desparticioná-los.

Código 12 – Estrutura do particionamento dos dados

```
struct starpu_data_filter vert =
{
        .filter_func = starpu_matrix_filter_vertical_block,
        .nchildren   = 2
};
struct starpu_data_filter horiz =
{
        .filter_func = starpu_matrix_filter_block,
        .nchildren   = 2
};
starpu_data_partition(Ah, &horiz);
starpu_data_partition(Bh, &vert);
starpu_data_map_filters(Ch, 2, &horiz, &vert);
```

Fonte: Autor

Figura 23 – Particionamento dos dados com 4 *slices*

Fonte: Autor

A.4 Escalonadores

Os escalonadores mantém no mínimo uma fila de tarefas prontas para execução (Université de Bordeaux; CNRS (LaBRI UMR 5800); Inria, 2021). As tarefas são classificadas prontas para execução quando não possuem dependência de dados, dependência de tarefas, etc. Existem diferentes tipos de políticas de escalonamento em StarPU.

A.4.1 Políticas de Escalonamento sem Modelo de Performance

Conjuntos de escalonadores indicados para aplicações que não utilizam um modelo de performance. Por padrão, o escalonador `lws` é utilizado nesses casos. A razão disso é que ele fornece equilíbrio de carga e localidade, mesmo quando os *codelets* da aplicação não possuem modelos de performance.

As políticas de escalonamento que não necessitam de um modelo de performance são:

- `eager`: mantém uma única fila de tarefas. Quando os trabalhadores estão livre, eles pegam as tarefas do topo;

- `random`: mantém uma fila de tarefas para cada trabalhador, distribuindo-as aleatoriamente de modo a deixar todos os trabalhadores ocupados;

- `ws`(*work stealing*): mantém uma fila de tarefas para cada trabalhador. Quando um trabalhador fica ocioso, ele rouba a tarefa do trabalhador mais ocupado;

- `lws`(*locality work stealing*): mantém uma fila de tarefas para cada trabalhador. Quando um trabalhador fica ocioso, ele rouba a tarefa do trabalhador vizinho, levando em conta a prioridade da tarefa;

- `prio`: mantém uma única fila de tarefas. Ele ordena as tarefas pela ordem de prioridade definida pelo programador;

- `heteroprio`: semelhante ao `prio`, porém utiliza de diferentes prioridades para diferentes unidades de processamento.

A.4.2 Políticas de Escalonamento com Modelo de Performance

Conjunto de escalonadores indicados somente para aplicações que utilizam um modelo de performance. Um desses modelos, conhecido mas já depreciado, chamado `heft`(*heterogeneous earliest finish time*), deu origem a diversos outros escalonadores, que o adaptaram de alguma forma. Esse escalonador ordenava as tarefas de acordo com seu tempo de execução.

Os escalonadores disponíveis atualmente são:

- `dm`(*deque model*): similar ao `heft`. Ele programa tarefas onde seu tempo de término será mínimo. Ele agenda as tarefas assim que estão disponíveis e, portanto, na ordem em que são disponibilizadas, sem levar em consideração as prioridades.

- `dmda`(*deque model data aware*): é semelhante ao `dm`, mas também leva em consideração o tempo de transferência de dados.

- `dmdap`(*deque model data aware prio*): é semelhante ao `dmda`, com a diferença de ordenar as tarefas por ordem de prioridade.

- `dmdar`(*deque model data aware ready*): é semelhante ao `dmda`, mas também privilegia tarefas cujos *buffers* de dados já estão disponíveis no dispositivo de destino.

- `dmdas`: combina `dmdap` e `dmdas`. Ele classifica as tarefas por ordem de prioridade, mas para uma determinada prioridade, ele irá privilegiar tarefas cujos *buffers* de dados já estão disponíveis no dispositivo de destino.

- `dmdasd`(*deque model data aware sorted decision*): é semelhante ao `dmdas`, exceto que ao agendar uma tarefa, ele leva em consideração sua prioridade ao calcular o tempo mínimo de conclusão, uma vez que esta tarefa pode ser executada antes de outras, e portanto, o último deve ser ignorado.

- `pheft`(*parallel heft*): é semelhante ao `dmda`, mas também suporta tarefas paralelas.

- `peager`(*parallel eager*): é semelhante ao `eager`, mas também suporta tarefas paralelas.

A.4.3 Escalonadores Modularizados

Além dos escalonadores padrão, a interface oferece a possibilidade de criação de novos escalonadores. StarPU já conta com alguns escalonadores extras, que nada mais são do que a combinação de um ou mais tipos predefinidos. São eles:

- `modular-eager`, `modular-eager-prefetching`: baseados em `eager` (sem e com pré-busca). Tentam mapear uma tarefa no primeiro recurso disponível que encontram. A variante de pré-busca enfileira várias tarefas com antecedência para poder fazer a pré-busca de dados. Isso pode acabar degradando um pouco o balanceamento de carga.

- `modular-prio`, `modular-prio-prefetching`, `modular-eager-prio`: baseados em `prio` (sem / com pré-busca), semelhantes aos escalonadores baseados em `eager`. Podem lidar com tarefas que tenham uma prioridade definida e programá-las de acordo. A variante `modular-eager-prio` integra a fila `eager` e `prio` em um único componente. Isso permite que ele faça um trabalho melhor ao empurrar tarefas.

- `modular-random`, `modular-random-prio`, `modular-random-prefetching`, `modular-random-prio-prefetching`: baseados em `random` (sem / com prefetching). Seleciona aleatoriamente um recurso a ser mapeado para cada tarefa.

- `modular-ws`: implementa `ws`. Mapeia tarefas para trabalhadores em *round robin*, mas permite que trabalhadores roubem trabalhos de outros trabalhadores.

- `modular-heft`, `modular-heft2`, `modular-heft-prio`: mapeiam tarefas para trabalhadores usando uma heurística muito próxima de `heft`. É aconselhável que cada tarefa tenha um modelo de performance definido para funcionar de forma eficiente. `modular-heft` apenas aceita tarefas por pedido. `modular-heft2` realiza no máximo 5 tarefas com a mesma prioridade e verifica qual delas se encaixa melhor. `modular-heft-prio` é semelhante a `modular-heft`, mas apenas decide o nó de memória, não o trabalhador exato. Apenas envia tarefas para uma fila central por nó de memória. Por padrão, eles classificam as tarefas por prioridades e privilégios, executando primeiro uma tarefa que já possui a maioria de seus dados disponíveis no destino. No entanto, eles podem ser alterados com as variáveis de ambiente `STARPU_SCHED_SORTED_ABOVE`, `STARPU_SCHED_SORTED_BELOW` e `STARPU_SCHED_READY`.

- `modular-heteroprio`: escalonador `heteroprio`. Mapeia tarefas para o trabalhador de forma semelhante ao `heft`, mas primeiro atribui tarefas aceleradas as GPUs, depois as tarefas não tão aceleradas as CPUs.

A.5 Modelo de Performance

A maioria dos escalonadores são baseados numa estimativa de duração de cada tipo de unidade de processamento (Université de Bordeaux; CNRS (LaBRI UMR 5800); Inria, 2021). Para que isso seja possível, o programador precisa configurar um modelo de performance para os *codelets* da aplicação.

A interface calibra automaticamente os *codelets* da aplicação que ainda não foram calibrados e salva em $STARPU_HOME/.starpu/sampling/codelets, mesmo que não sejam usados futuramente. Para evitar ter que calibrar modelos de performance para cada nó de um *cluster* homogêneo, pode-se compartilhar entre os nós usando `export STARPU_HOSTNAME = some_global_name (STARPU_HOSTNAME)`, onde some_global_name é o nome do *cluster*. Da mesma forma, ao invés de se criar um modelo de performance para cada GPU pode-se compartilhar um modelo entre elas.

O StarPU não registra a primeira medição para um determinado *codelet* e um determinado tamanho. Isso porque na maioria das vezes o carregamento e/ou inicialização da biblioteca podem interferir na medição, gerando um valor incorreto. O StarPU também

descarta as medições se notar que, depois de calcular um tempo médio de execução, a maioria das tarefas subsequentes possuem um desvio padrão alto.

A.6 Alguns Resultados da API

Para aprimorar os conhecimentos de StarPU, em um primeiro momento foram feitas implementações e testes em um código distinto da aplicação alvo. A Tabela 9 mostra os tempos de execução para multiplicação de matrizes de diferentes dimensões quadradas. Os códigos podem ser encontrados no Apêndice B. Para o cálculo do tempo total foi utilizado o comando `time`. Devido a isso, o calculo do tempo total gasto englobou alguns fatores extras. Além do tempo da função que de fato realiza a operação de multiplicação, operações extras como alocação, inicialização e liberação das matrizes consumiram uma certa quantidade de tempo. Essas 3 operações consumiram, em média, $5,33.10^{-3}$ para a dimensão 512x512, $11,33.10^{-3}$ para 1024x1024 e $45,47.10^{-3}$ para 2048x2048. Uma média da soma do tempo total de 3 execuções para cada caso de teste foi realizada. O escalonador utilizado para as execuções foi `lws`.

Tabela 9 – Tempo de multiplicação de matrizes para diferentes dimensões e múltiplas implementações

Dimensão	Configuração	Dispositivo	Tempo(s)
512x512	Sequencial	CPU	0,847
	8 *slices*	CPU	2,795
		CPU+GPU	2,767
	16 *slices*	CPU	2,779
		CPU+GPU	2,646
	32 *slices*	CPU	2,682
		CPU+GPU	2,700
1024x1024	Sequencial	CPU	9,040
	8 *slices*	CPU	3,376
		CPU+GPU	2,627
	16 *slices*	CPU	3,370
		CPU+GPU	2,846
	32 *slices*	CPU	3,159
		CPU+GPU	2,869
2048x1024	Sequencial	CPU	81,729
	8 *slices*	CPU	9,635
		CPU+GPU	3,768
	16 *slices*	CPU	8,665
		CPU+GPU	3,532
	32 *slices*	CPU	8,813
		CPU+GPU	4,203

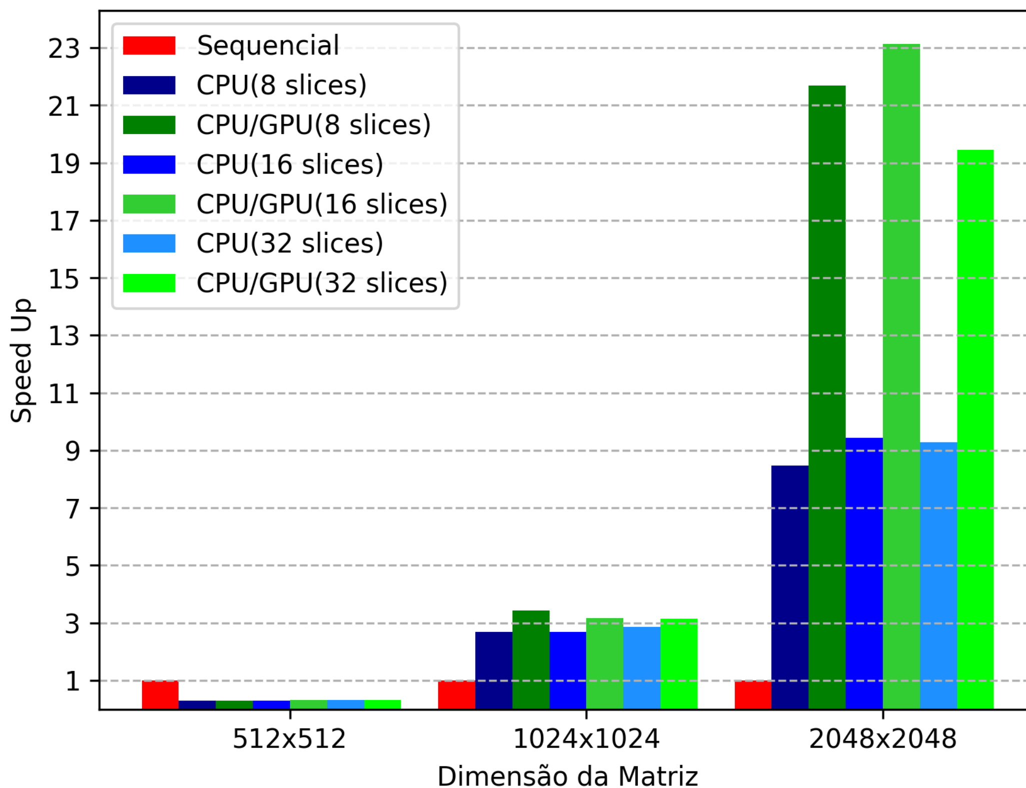
Fonte: Autor

A Figura 24 apresenta o *speed up* gerado com os valores obtidos presentes na Tabela 9. Para a dimensão 512x512 nota-se uma piora no tempo de execução com o uso

do *runtime*. Essa piora nos resultados é justificável. Como o tamanho do problema para essa aplicação é considerado pequeno e, como já mencionado nas seções anteriores, ao tempo da aplicação é acrescido o tempo de gerenciamento do *runtime*, o tempo total da aplicação tende a ser piorado. A partir do momento em que o tamanho da aplicação cresce, esse tempo de gerenciamento se torna irrelevante.

A aplicação com múltiplas implementações, sendo elas em CPU e GPU(CUDA), se mostrou mais eficiente do que uma única implementação em CPU. Pode-se observar na Figura 24, para dimensão 2048x2048, um *speed up* maior do que duas vezes o *speed up* da implementação em CPU para CPU+GPU(CUDA). Outra observação interessante a se fazer é com relação ao número de *slices*. Percebe-se que o melhor tempo de execução fica em torno do uso de 16 *slices* – para essa aplicação e essas dimensões. Não foi testado para número de *slices* entre 16 e 32 para saber qual realmente era a melhor quantidade a ser escolhida. Pode-se encontrar os códigos usados para gerar esses resultados nos anexos.

Figura 24 – *Speed Up* de multiplicação de matrizes para diferentes dimensões e múltiplas implementações



Fonte: Autor

APÊNDICE B – MULTIPLICAÇÃO DE MATRIZES

Código 13 – Multiplicação de matrizes utilizando uma tarefa em CPU

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <starpu.h>

int *A, *B, *C;
starpu_data_handle_t Ah, Bh, Ch;

int N = 32; //valor default

struct starpu_codelet codelet =
{
        .cpu_funcs = {func_cpu},
        .nbuffers  = 3,
        .modes     = {STARPU_R, STARPU_R, STARPU_W},
};

extern void func_cpu(void *buffers[], void *args);
extern void init_problem(void);
extern void end_problem(void);
extern void launch_task(void);

int main(int argc, char *argv[]){
        N = atoi(argv[1]);

        int ini = starpu_init(NULL);

        init_problem();
        launch_task();
        starpu_task_wait_for_all();
        end_problem();

        starpu_shutdown();

        return 0;
}
```

Código 14 – Multiplicação de matrizes utilizando uma tarefa em GPU

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <starpu.h>

int *A, *B, *C;
```

```
starpu_data_handle_t Ah, Bh, Ch;

int N = 32; //valor default

struct starpu_codelet codelet =
{
        .cuda_funcs = { func_gpu },
        .nbuffers   = 3,
        .modes      = { STARPU_R, STARPU_R, STARPU_W },
};

extern void func_gpu(void *buffers[], void *args);
extern void pin_memory(void);
extern void unpin_memory(void);
extern void init_problem(void);
extern void end_problem(void);
extern void launch_task(void);

int main(int argc, char *argv[]){
        N = atoi(argv[1]);

        int ini = starpu_init(NULL);

        init_problem();
        pin_memory();
        launch_task();
        starpu_task_wait_for_all();
        end_problem();
        unpin_memory();

        starpu_shutdown();

        return 0;
}
```

Código 15 – Multiplicação de matrizes utilizando número de tarefas variável em CPU

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <starpu.h>

int *A, *B, *C;
starpu_data_handle_t Ah, Bh, Ch;

int N = 32, s = 4; //valor default

struct starpu_codelet codelet =
```

```
12 {
13         .cpu_funcs = {cpu_mult},
14         .nbuffers  = 3,
15         .modes     = {STARPU_R, STARPU_R, STARPU_W},
16 };
17
18 extern void func_cpu(void *buffers[], void *args);
19 extern void init_problem(void);
20 extern void end_problem(void);
21 extern void partition(void);
22 extern void unpartition(void);
23 extern void launch_tasks(void);
24
25 int main(int argc, char *argv[]){
26         N = atoi(argv[1]);
27         s = atoi(argv[2]);
28
29         int ini = starpu_init(NULL);
30
31         init_problem();
32         partition();
33         launch_tasks();
34         starpu_task_wait_for_all();
35         unpartition();
36         end_problem();
37
38         starpu_shutdown();
39
40         return 0;
41 }
```

Código 16 – Multiplicação de matrizes utilizando número de tarefas variável tarefas em CPU e GPU

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <limits.h>
4  #include <starpu.h>
5
6  int *A, *B, *C;
7  starpu_data_handle_t Ah, Bh, Ch;
8
9  int N = 32, s = 4; //valor default
10
11 struct starpu_codelet codelet =
12 {
13         .where      = STARPU_CPU | STARPU_CUDA,
14         .cpu_funcs  = { func_cpu },
```

```
15          .cuda_funcs = { func_gpu },
16          .nbuffers   = 3,
17          .modes      = { STARPU_R, STARPU_R, STARPU_W },
18  };
19
20  extern void func_gpu(void *buffers[], void *args);
21  extern void func_cpu(void *buffers[], void *args);
22  extern void init_problem(void);
23  extern void end_problem(void);
24  extern void pin_memory(void);
25  extern void unpin_memory(void);
26  extern void partition(void);
27  extern void unpartition(void);
28  extern void launch_tasks(void);
29
30  int main(int argc, char *argv[]){
31          N = atoi(argv[1]);
32          s = atoi(argv[2]);
33
34          int ini = starpu_init(NULL);
35
36          init_problem();
37          pin_memory();
38          partition();
39          launch_tasks();
40          starpu_task_wait_for_all();
41          unpartition();
42          end_problem();
43          unpin_memory();
44
45          starpu_shutdown();
46
47          return 0;
48  }
```

Código 17 – Funções Externas

```
1   __global__ void multMatDevice(int nn, int NN, int *a, int *b, int *c){
2           int linha = blockIdx.x * blockDim.x + threadIdx.x;
3           int coluna = blockIdx.y * blockDim.y + threadIdx.y;
4
5           if(linha < nn && coluna < nn){
6                   int aux = 0;
7                   for(int k = 0; k < NN; k ++)
8                           aux += a[linha*NN+k] * b[k*nn+coluna];
9                   c[linha*nn+coluna] = aux;
10          }
11  }
```

```
12
13 void func_gpu(void *buffers[], void *args){
14         int NN = STARPU_MATRIX_GET_NY(buffers[0]);
15         int nn = STARPU_MATRIX_GET_NX(buffers[0]);
16
17         int *A = (int*)STARPU_MATRIX_GET_PTR(buffers[0]);
18         int *B = (int*)STARPU_MATRIX_GET_PTR(buffers[1]);
19         int *C = (int*)STARPU_MATRIX_GET_PTR(buffers[2]);
20
21         unsigned threads_per_block = 32;
22         unsigned nblocks = (nn + threads_per_block-1) / threads_per_block;
23
24         dim3 threadsPerBlock(threads_per_block, threads_per_block);
25         dim3 dimGrid(nblocks, nblocks);
26
27         multMatDevice<<<dimGrid, threadsPerBlock,
28                 0, starpu_cuda_get_local_stream()>>>(nn, NN, A, B, C);
29
30         cudaError_t status = cudaGetLastError();
31         if (status != cudaSuccess)
32                 STARPU_CUDA_REPORT_ERROR(status);
33
34         cudaStreamSynchronize(starpu_cuda_get_local_stream());
35 }
36
37 void func_cpu(void *buffers[], void *args){
38         int *A = (int*)STARPU_MATRIX_GET_PTR(buffers[0]);
39         int *B = (int*)STARPU_MATRIX_GET_PTR(buffers[1]);
40         int *C = (int*)STARPU_MATRIX_GET_PTR(buffers[2]);
41
42         int NN = STARPU_MATRIX_GET_NY(buffers[0]);
43         int nn = STARPU_MATRIX_GET_NX(buffers[0]);
44
45         int aux;
46         for(int i = 0; i < nn; i ++){
47                 for(int j = 0; j < nn; j ++){
48                         aux = 0;
49                         for(int k = 0; k < NN; k ++)
50                                 aux += A[i*NN+k] * B[k*nn+j];
51                         C[i*NN+j] = aux;
52                 }
53         }
54 }
55
56 void init_problem(void){
57         A = (int*)malloc(N*N*sizeof(int));
58         B = (int*)malloc(N*N*sizeof(int));
```

```
59          C = (int*)malloc(N*N*sizeof(int));
60
61          for(int i=0; i < N; i++){
62                  for(int j=0; j < N; j++){
63                          A[i*N+j] = 1;
64                          B[j*N+i] = 1;
65                  }
66          }
67
68          starpu_matrix_data_register(&Ah, STARPU_MAIN_RAM,
69                  (uintptr_t)A, N, N, N, sizeof(int));
70          starpu_matrix_data_register(&Bh, STARPU_MAIN_RAM,
71                  (uintptr_t)B, N, N, N, sizeof(int));
72          starpu_matrix_data_register(&Ch, STARPU_MAIN_RAM,
73                  (uintptr_t)C, N, N, N, sizeof(int));
74  }
75
76  void end_problem(void){
77          starpu_data_unregister(Ah);
78          starpu_data_unregister(Bh);
79          starpu_data_unregister(Ch);
80
81          free(A);
82          free(B);
83          free(C);
84  }
85
86  void pin_memory(void){
87          starpu_memory_pin((void*)A, N * N * sizeof(int));
88          starpu_memory_pin((void*)B, N * N * sizeof(int));
89          starpu_memory_pin((void*)C, N * N * sizeof(int));
90  }
91
92  void unpin_memory(void){
93          starpu_memory_unpin((void*)A, N * N * sizeof(int));
94          starpu_memory_unpin((void*)B, N * N * sizeof(int));
95          starpu_memory_unpin((void*)C, N * N * sizeof(int));
96  }
97
98  void partition(void){
99          struct starpu_data_filter vert =
100         {
101                 .filter_func = starpu_matrix_filter_vertical_block,
102                 .nchildren   = (unsigned int)s
103         };
104
105         struct starpu_data_filter horiz =
```

```
106         {
107                 .filter_func = starpu_matrix_filter_block,
108                 .nchildren   = (unsigned int)s
109         };
110 
111         starpu_data_partition(Ah, &horiz);
112         starpu_data_partition(Bh, &vert);
113 
114         starpu_data_map_filters(Ch, 2, &horiz, &vert);
115 }
116 
117 void unpartition(void){
118         starpu_data_unpartition(Ah, STARPU_MAIN_RAM);
119         starpu_data_unpartition(Bh, STARPU_MAIN_RAM);
120         starpu_data_unpartition(Ch, STARPU_MAIN_RAM);
121 }
122 
123 void launch_tasks(void){
124         for(int i = 0; i < s; i++) {
125                 for(int j = 0; j < s; j++){
126                         struct starpu_task *task = starpu_task_create();
127 
128                         task->cl = &codelet;
129 
130                         task->handles[0] =
131                                 starpu_data_get_sub_data(Ah, 1, i);
132                         task->handles[1] =
133                                 starpu_data_get_sub_data(Bh, 1, j);
134                         task->handles[2] =
135                                 starpu_data_get_sub_data(Ch, 2, i, j);
136 
137                         int ts = starpu_task_submit(task);
138                 }
139         }
140 }
141 
142 void launch_task(void){
143         struct starpu_task *task = starpu_task_create();
144 
145         task->cl = &codelet;
146 
147         task->handles[0] = Ah;
148         task->handles[1] = Bh;
149         task->handles[2] = Ch;
150 
151         int ts = starpu_task_submit(task);
152 }
```

APÊNDICE C – CODELETS STARPU

Código 18 – codelet_U associado com a rotina solve_u.

```
1  recursive subroutine solve_U (buffers, cl_args) bind(C)
2      use iso_c_binding        ! C interfacing module
3      use fstarpu_mod          ! StarPU interfacing module
4      use comum
5      implicit none
6
7      integer :: i, j, ij, ii,uij,ini_i,fim_i,lu,cu,lv,cv, ini_ui, fim_ui
8
9      TYPE(C_PTR), VALUE, INTENT(IN)                :: buffers, cl_args
10
11     REAL(8), DIMENSION(:,:), pointer              :: RU, res_u
12     REAL(8), DIMENSION(:), pointer                :: ui, um, um_n, um_tau,
       vm_tau, vm_taup, um_n_tau, p
13     REAL(8), POINTER                              :: residual_u
14     INTEGER, POINTER                              :: pos
15
16     lu=fstarpu_matrix_get_ny(buffers, 4)
17     cu=fstarpu_matrix_get_nx(buffers, 4)
18
19     lv=fstarpu_matrix_get_ny(buffers, 3)
20     cv=fstarpu_matrix_get_nx(buffers, 3)
21
22      call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 0), um,
       shape=[(imax+1)*jmax])
23      call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 1), um_tau,
       shape=[(imax+1)*jmax])
24      call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 2), vm_tau,
       shape=[imax*(jmax+1)])
25      call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 3), vm_taup,
       shape=[lv*cv])
26      call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 4), um_n_tau,
       shape=[lu*cu])
27      call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 5), p,
       shape=[imax*jmax])
28      call c_f_pointer(fstarpu_variable_get_ptr(buffers, 6), residual_u)
29      call c_f_pointer(fstarpu_variable_get_ptr(buffers, 7), pos)
30
31     IF (pos .EQ. 1) THEN        !top
32         ini_i = 3
33         fim_i = imaxpu1
34
35         IF (pos .LE. resto1) THEN
36             fim_i = fim_i + pos
37         ENDIF
```

```
38
39          fim_i = fim_i+2 !add
40      ELSE                          !middle/bottom
41          fim_i = pos*imaxpu1
42
43          IF (pos .LE. resto1) THEN
44              fim_i = fim_i + pos
45              ini_i = fim_i - imaxpu1
46          ELSE
47              fim_i = fim_i + resto1
48              ini_i = fim_i - imaxpu1 + 1
49          ENDIF
50
51          IF (pos .EQ. slices) THEN
52              fim_i = fim_i - 2
53          ENDIF
54
55          IF (pos .EQ. slices) THEN
56              ini_i = ini_i-2 !add
57          ELSE
58              ini_i = ini_i-2 !add
59              fim_i = fim_i+2 !add
60          ENDIF
61      ENDIF
62
63      ALLOCATE(ui((imax+1)*jmax))
64      ALLOCATE(RU(imax+1,jmax))
65      ALLOCATE(res_u(imax+1,jmax))
66
67      RU = 0.d0
68      res_u = 0.d0
69      ui = 0.0d0
70
71      CALL RESU(um_tau,vm_tau,p,RU,pos)
72
73      DO i=ini_i,fim_i
74          DO j=2,jmax-1
75              ij = jmax*(i-1)+j
76              res_u(i,j) = ((um(ij)-um_tau(ij)) + RU(i,j)*dt) * dtau
77              ui(ij) = um_tau(ij) + res_u(i,j)
78          ENDDO
79      ENDDO
80
81      call bcV_part(vm_taup,lv,cv,pos)
82      call bcU_npar(ui,pos)
83      CALL RESU(ui,vm_tau,p,RU,pos)
84
```

```
85      DO i=ini_i,fim_i
86          DO j=2,jmax-1
87              ij = jmax*(i-1)+j
88              res_u(i,j) = ((um(ij)-um_tau(ij)) + RU(i,j)*dt) * dtau
89              ui(ij) = 0.75d0 * um_tau(ij) + 0.25d0 * (ui(ij) + res_u(i,j))
90          ENDDO
91      ENDDO
92
93      call bcU_npar(ui,pos)
94      CALL RESU(ui,vm_tau,p,RU,pos)
95
96      IF (pos .EQ. 1) THEN           !top
97          ini_i = 3
98          fim_i = lu
99      ELSE IF (pos .EQ. slices) THEN !bottom
100         ini_i = 1
101         fim_i = lu-2
102     ELSE                           !middle
103         ini_i = 1
104         fim_i = lu
105     ENDIF
106
107     IF (pos .EQ. 1) THEN           !top
108         ini_ui = 1
109     ELSE              !middle/bottom
110         fim_ui = pos*imaxpu1
111         IF (pos .LE. resto1) THEN
112             fim_ui = fim_ui + pos
113             ini_ui = fim_ui - imaxpu1
114         ELSE
115             fim_ui = fim_ui + resto1
116             ini_ui = fim_ui - imaxpu1 + 1
117         ENDIF
118     ENDIF
119
120     res_u = 0.d0
121     DO i=ini_i,fim_i
122         DO j=2,jmax-1
123             ij = cu*(i-1)+j
124             ii = cu*(ini_ui-1)
125             res_u(i+(ini_ui-1),j) = ((um(ii+ij)-um_tau(ii+ij)) + RU(i+(
    ini_ui-1),j)*dt) * dtau
126             um_n_tau(ij) = 1.0d0 / 3.0d0 * um_tau(ii+ij) + 2.0d0 / 3.0d0 *
    ( ui(ii+ij) + res_u(i+(ini_ui-1),j) )
127         ENDDO
128     ENDDO
129
```

```
130      call bcU_part(um_n_tau,lu,cu,pos)
131
132      residual_u =  MAXVAL(ABS(res_u))
133
134      DEALLOCATE(ui)
135      DEALLOCATE(RU)
136      DEALLOCATE(res_u)
137
138      RETURN
139 END SUBROUTINE solve_U
```

Fonte: Autor

Código 19 – codelet_V associado com a rotina solve_v.

```
1  recursive subroutine solve_V (buffers, cl_args) bind(C)
2      use iso_c_binding        ! C interfacing module
3      use fstarpu_mod          ! StarPU interfacing module
4      use comum
5      implicit none
6
7      TYPE(C_PTR), VALUE, INTENT(IN)                    :: buffers, cl_args
8      INTEGER                                           :: i, j, ij, ii, uij,
       ini_i,fim_i,lu,cu,lv,cv, ini_vi, fim_vi
9      REAL(8), DIMENSION(:,:), pointer                  :: RV, res_v
10     REAL(8), DIMENSION(:), pointer                    :: vi, um_tau, um_taup,
       vm_tau, vm_n_tau, P, T, vm
11     REAL(8), POINTER                                  :: residual_u
12     INTEGER, POINTER                                  :: pos
13     REAL(8), pointer                                  :: residual_v, InvFr2
14
15     lu=fstarpu_matrix_get_ny(buffers, 2)
16     cu=fstarpu_matrix_get_nx(buffers, 2)
17
18     lv=fstarpu_matrix_get_ny(buffers, 4)
19     cv=fstarpu_matrix_get_nx(buffers, 4)
20
21     !vm,um_tau,vm_tau,vm_n_tau,p,T,InvFr2,residual_v
22     call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 0), vm,
       shape=[imax*(jmax+1)])
23     call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 1), um_tau,
       shape=[(imax+1)*jmax])
24     call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 2), um_taup,
       shape=[lu*cu])
25     call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 3), vm_tau,
       shape=[imax*(jmax+1)])
26     call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 4), vm_n_tau,
       shape=[lv*cv])
```

```
27      call c_f_pointer(  fstarpu_matrix_get_ptr(buffers, 5), p,
       shape=[imax*jmax])
28      call c_f_pointer(  fstarpu_matrix_get_ptr(buffers, 6), T,
       shape=[imax*jmax])
29      call c_f_pointer(fstarpu_variable_get_ptr(buffers, 7), InvFr2)
30      call c_f_pointer(fstarpu_variable_get_ptr(buffers, 8), residual_v)
31      call c_f_pointer(fstarpu_variable_get_ptr(buffers, 9), pos)
32
33      IF (pos .EQ. 1) THEN         !top
34          ini_i = 2
35          fim_i = imaxpu
36
37          IF (pos .LE. resto) THEN
38              fim_i = fim_i + pos
39          ENDIF
40
41          fim_i = fim_i+2 !add
42      ELSE                              !middle/bottom
43          fim_i = pos*imaxpu
44
45          IF (pos .LE. resto) THEN
46              fim_i = fim_i + pos
47              ini_i = fim_i - imaxpu
48          ELSE
49              fim_i = fim_i + resto
50              ini_i = fim_i - imaxpu + 1
51          ENDIF
52
53          IF (pos .EQ. slices) THEN
54              fim_i = fim_i - 1
55          ENDIF
56
57          IF (pos .EQ. slices) THEN
58              ini_i = ini_i-2 !add
59          ELSE
60              ini_i = ini_i-2 !add
61              fim_i = fim_i+2 !add
62          ENDIF
63      ENDIF
64
65      ALLOCATE(vi(imax*(jmax+1)))
66      ALLOCATE(RV(imax,jmax+1))
67      ALLOCATE(res_v(imax,jmax+1))
68
69      RV = 0.d0
70      res_v = 0.d0
71      vi = 0.d0
```

```
72
73      CALL RESV(um_tau,vm_tau,p,T,InvFr2,RV,pos)
74
75      DO i=ini_i,fim_i
76          DO j=3,jmax-1
77              ij = (jmax+1)*(i-1)+j
78              res_v(i,j) = ( (vm(ij)-vm_tau(ij)) + RV(i,j)*dt) * dtau
79              vi(ij) = ( vm_tau(ij) + res_v(i,j) )
80          ENDDO
81      ENDDO
82
83      call bcU_part(um_taup,lu,cu,pos)
84      call bcV_npar(vi,pos)
85      CALL RESV(um_tau,vi,p,T,InvFr2,RV,pos)
86
87      DO i=ini_i,fim_i
88          DO j=3,jmax-1
89              ij = (jmax+1)*(i-1)+j
90              res_v(i,j) = ( (vm(ij)-vm_tau(ij)) + RV(i,j)*dt) * dtau
91              vi(ij) =( 0.75d0 * vm_tau(ij) + 0.25d0 * ( vi(ij) + res_v(i,j))
        )
92          ENDDO
93      ENDDO
94
95      call bcV_npar(vi,pos)
96      CALL RESV(um_tau,vi,p,T,InvFr2,RV,pos)
97
98      IF (pos .EQ. 1) THEN            !top
99          ini_i = 2
100         fim_i = lv ! +2!para poder fazer RESU
101     ELSE IF (pos .EQ. slices) THEN !bottom
102         ini_i = 1
103         fim_i = lv-1
104     ELSE                            !middle
105         ini_i = 1
106         fim_i = lv
107     ENDIF
108
109     IF (pos .EQ. 1) THEN            !top
110         ini_vi = 1
111     ELSE            !middle/bottom
112         fim_vi = pos*imaxpu
113         IF (pos .LE. resto) THEN
114             fim_vi = fim_vi + pos
115             ini_vi = fim_vi - imaxpu
116         ELSE
117             fim_vi = fim_vi + resto
```

```
118                 ini_vi = fim_vi - imaxpu + 1
119             ENDIF
120         ENDIF
121 
122         DO i=ini_i,fim_i
123             DO j=3,jmax-1
124                 ij = (jmax+1)*(i-1)+j
125                 ii = cv*(ini_vi-1)
126                 res_v(i+(ini_vi-1),j) = ( (vm(ii+ij)-vm_tau(ii+ij)) + RV(i+(
        ini_vi-1),j)*dt) * dtau
127                 vm_n_tau(ij) = 1.0d0 / 3.0d0 * vm_tau(ii+ij) + 2.0d0 / 3.0d0 *
        ( vi(ii+ij) + res_v(i+(ini_vi-1),j))
128             ENDDO
129         ENDDO
130 
131         CALL bcV_part(vm_n_tau,lv,cv,pos)
132 
133         residual_v = MAXVAL(ABS(res_v))
134 
135         DEALLOCATE(vi)
136         DEALLOCATE(RV)
137         DEALLOCATE(res_v)
138 
139         RETURN
140 END SUBROUTINE solve_V
```

Fonte: Autor

Código 20 – codelet_P associado com a rotina solve_p.

```
1  recursive subroutine solve_P (buffers, cl_args) bind(C)
2      use iso_c_binding          ! C interfacing module
3      use fstarpu_mod            ! StarPU interfacing module
4      use comum
5      implicit none
6 
7      TYPE(C_PTR), VALUE, INTENT(IN)                  :: buffers, cl_args
8 
9      INTEGER                                         :: i, j,ini_i,fim_i,lp,cp
       ,lu,cu,lv,cv
10     REAL(8)                                         :: c, max_vel
11     REAL(8), DIMENSION(:,:), pointer                :: RP, res_p
12     REAL(8), DIMENSION(:), pointer                  :: Pi,p, um_n, vm_n, pn
13     REAL(8), POINTER                                :: c2, residual_p
14     INTEGER, POINTER                                :: pos
15 
16     lp=fstarpu_matrix_get_ny(buffers, 4)
17     cp=fstarpu_matrix_get_nx(buffers, 4)
```

```
18
19      lu=fstarpu_matrix_get_ny(buffers, 2)
20      cu=fstarpu_matrix_get_nx(buffers, 2)
21
22      lv=fstarpu_matrix_get_ny(buffers, 3)
23      cv=fstarpu_matrix_get_nx(buffers, 3)
24
25      call c_f_pointer(fstarpu_variable_get_ptr(buffers, 0), c2)
26      call c_f_pointer(  fstarpu_matrix_get_ptr(buffers, 1), P,
       shape=[lp*cp])
27      call c_f_pointer(  fstarpu_matrix_get_ptr(buffers, 2), um_n,
       shape=[lu*cu])
28      call c_f_pointer(  fstarpu_matrix_get_ptr(buffers, 3), vm_n,
       shape=[lv*cv])
29      call c_f_pointer(  fstarpu_matrix_get_ptr(buffers, 4), Pn,
       shape=[lp*cp])
30      call c_f_pointer(fstarpu_variable_get_ptr(buffers, 5), residual_p)
31      call c_f_pointer(fstarpu_variable_get_ptr(buffers, 6), pos)
32
33      IF (pos .EQ. 1) THEN              !top
34          ini_i = 2
35          fim_i = lp
36      ELSE IF (pos .EQ. slices) THEN !bottom
37          ini_i = 1
38          fim_i = lp-1
39      ELSE                              !middle
40          ini_i = 1
41          fim_i = lp
42      ENDIF
43
44      ALLOCATE(Pi(lp*cp))
45      ALLOCATE(RP(lp,cp))
46      ALLOCATE(res_p(fim_i,jmax))
47
48      max_vel = MAXVAL(um_n**2.d0) + MAXVAL(vm_n**2.d0)
49
50      c2 = beta !* MAXVAL(vm_n**2.d0)
51
52      RP = 0.0d0
53      pi = 0.0d0
54      res_p=0.0d0
55
56      CALL RESP(um_n,lu,cu,vm_n,lv,cv,RP,lp,cp,pos)
57
58      DO i=ini_i,fim_i
59          DO j=2,jmax-1
60              pi(cp*(i-1)+j) = P(cp*(i-1)+j) + dtau * RP(i,j) * c2
```

```
            ENDDO
        ENDDO

        CALL bcP(pi,lp,cp,pos)

        CALL RESP(um_n,lu,cu,vm_n,lv,cv,RP,lp,cp,pos)

        DO i=ini_i,fim_i
            DO j=2,jmax-1
                pi(cp*(i-1)+j) = 0.75d0 * P(cp*(i-1)+j) + 0.25d0 * (pi(cp*(i-1)
        +j) + dtau * RP(i,j) * c2)
            ENDDO
        ENDDO

        CALL bcP(pi,lp,cp,pos)

        CALL RESP(um_n,lu,cu,vm_n,lv,cv,RP,lp,cp,pos)

        DO i=ini_i,fim_i
            DO j=2,jmax-1
                res_p(i,j) = dtau * RP(i,j) * c2
                Pn(cp*(i-1)+j) = 1.0d0 / 3.0d0 * P(cp*(i-1)+j) + 2.0d0 / 3.0d0
        * (pi(cp*(i-1)+j) + res_p(i,j))
            ENDDO
        ENDDO

        CALL bcP(pn,lp,cp,pos)

        residual_p = MAXVAL(ABS(res_p))

        DEALLOCATE(Pi)
        DEALLOCATE(RP)
        DEALLOCATE(res_p)

        RETURN
END SUBROUTINE solve_P
```

Fonte: Autor

Código 21 – codelet_Z associado com a rotina solve_z.

```
recursive subroutine solve_Z (buffers, cl_args) bind(C)
    use iso_c_binding       ! C interfacing module
    use fstarpu_mod         ! StarPU interfacing module
    use comum
    implicit none
    TYPE(C_PTR), VALUE, INTENT(IN)                    :: buffers, cl_args
```

```
7      INTEGER                                          :: i, j, ij, ii, uij,
       ini_i,fim_i,lz,cz, ini_zi, fim_zi
8
9      REAL(8), DIMENSION(:), pointer :: um_n, vm_n, Z, Z_n_tau, Z_tau, zi
10     REAL(8), DIMENSION(:,:), pointer :: RZ,res_Z
11     INTEGER, pointer :: pos
12
13      lz=fstarpu_matrix_get_ny(buffers, 3)
14      cz=fstarpu_matrix_get_nx(buffers, 3)
15
16      !um_n,vm_n,Z,Z_n_tau,Z_tau
17      call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 0), um_n,      shape
       =[(imax+1)*jmax])
18      call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 1), vm_n,      shape
       =[imax*(jmax+1)])
19      call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 2), Z,        shape=[
       imax*jmax])
20      call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 3), Z_n_tau, shape=[
       lz*cz])
21      call c_f_pointer( fstarpu_matrix_get_ptr(buffers, 4), Z_tau,    shape=[
       imax*jmax])
22      call c_f_pointer(fstarpu_variable_get_ptr(buffers, 5), pos)
23
24     IF (pos .EQ. 1) THEN       !top
25         ini_i = 2
26         fim_i = imaxpu
27
28         IF (pos .LE. resto) THEN
29             fim_i = fim_i + pos
30         ENDIF
31
32         fim_i = fim_i+2 !add
33     ELSE                           !middle/bottom
34         fim_i = pos*imaxpu
35
36         IF (pos .LE. resto) THEN
37             fim_i = fim_i + pos
38             ini_i = fim_i - imaxpu
39         ELSE
40             fim_i = fim_i + resto
41             ini_i = fim_i - imaxpu + 1
42         ENDIF
43
44         IF (pos .EQ. slices) THEN
45             fim_i = fim_i - 1
46         ENDIF
47
```

```
48            IF (pos .EQ. slices) THEN
49                ini_i = ini_i-2 !add
50            ELSE
51                ini_i = ini_i-2 !add
52                fim_i = fim_i+2 !add
53            ENDIF
54        ENDIF
55
56        ALLOCATE(zi(imax*jmax))
57        ALLOCATE(RZ(imax,jmax))
58        ALLOCATE(res_z(imax,jmax))
59
60        zi = 0.d0
61        res_Z = 0.d0
62        RZ = 0.d0
63
64        CALL RESZ(um_n,vm_n,Z_tau,RZ,pos)
65
66        DO i=ini_i,fim_i
67            DO j=2,jmax-1
68                ij = jmax*(i-1)+j
69                res_Z(i,j) =( (Z(ij)-Z_tau(ij)) + RZ(i,j)*dt) * dtau
70                Zi(ij) = Z_tau(ij) + res_Z(i,j)
71            ENDDO
72        ENDDO
73
74        CALL bcZ_npar(Zi,pos)
75        CALL RESZ(um_n,vm_n,Zi,RZ,pos)
76
77        DO i=ini_i,fim_i
78            DO j=2,jmax-1
79                ij = jmax*(i-1)+j
80                res_Z(i,j) =( (Z(ij)-Z_tau(ij)) + RZ(i,j)*dt) * dtau
81                Zi(ij) = 0.75d0 * Z_tau(ij) + 0.25d0 * (Zi(ij) + res_Z(i,j))
82            ENDDO
83        ENDDO
84
85        CALL bcZ_npar(Zi,pos)
86        CALL RESZ(um_n,vm_n,Zi,RZ,pos)
87
88        IF (pos .EQ. 1) THEN              !top
89            ini_i = 2
90            fim_i = lz ! +2!para poder fazer RESU
91        ELSE IF (pos .EQ. slices) THEN !bottom
92            ini_i = 1
93            fim_i = lz-1
94        ELSE                              !middle
```

```
95          ini_i = 1
96          fim_i = lz
97      ENDIF
98
99      IF (pos .EQ. 1) THEN               !top
100         ini_zi = 1
101     ELSE                !middle/bottom
102         fim_zi = pos*imaxpu
103         IF (pos .LE. resto) THEN
104             fim_zi = fim_zi + pos
105             ini_zi = fim_zi - imaxpu
106         ELSE
107             fim_zi = fim_zi + resto
108             ini_zi = fim_zi - imaxpu + 1
109         ENDIF
110     ENDIF
111
112     DO i=ini_i,fim_i
113         DO j=2,jmax-1
114             ij = jmax*(i-1)+j
115             ii = cz*(ini_zi-1)
116             res_Z(i+(ini_zi-1),j) =( (Z(ii+ij)-Z_tau(ii+ij)) +  RZ(i+(
    ini_zi-1),j)*dt) * dtau
117             Z_n_tau(ij) = 1.0d0 / 3.0d0 * Z_tau(ii+ij) + 2.0d0 / 3.0d0 * (
    Zi(ii+ij) + res_Z(i+(ini_zi-1),j))
118         ENDDO
119     ENDDO
120
121     CALL bcZ_part(Z_n_tau,lz,cz,pos)
122
123     DEALLOCATE(zi)
124     DEALLOCATE(RZ)
125     DEALLOCATE(res_z)
126
127     RETURN
128 END SUBROUTINE solve_Z
```

Fonte: Autor

ÍNDICE

API, 25, 31, 32, 71, 72

CPU, 31, 32, 42, 43, 53, 63, 71, 81, 83
CUDA, 27, 29, 32, 73, 83

GPU, 24, 32, 42, 49, 50, 53, 54, 63, 71, 81, 83

MPI, 28

OpenACC, 24, 27–29, 31, 32, 49, 50, 53–55, 63
OpenCAL, 29
OpenCL, 27, 28, 32
OpenMP, 24, 27–29, 31, 49, 50, 53–55, 63

PETSc, 29