UNIVERSIDADE FEDERAL DO PAMPA

Natiele Lucca

Avaliação de Estratégias de Paralelismo em Simulação de Meios Porosos

Alegrete
2022

Natiele Lucca

Avaliação de Estratégias de Paralelismo em Simulação de Meios Porosos

Dissertação apresentado ao Programa de Pós-graduação Stricto Sensu em Engenharia de Software da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Mestre em Engenharia de Software.

Orientador: Prof. Dr. Claudio Schepke

Alegrete
2022

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) .

L934a Lucca, Natiele
Avaliação de Estratégias de Paralelismo em Simulação de Meios Porosos / Natiele Lucca.
81 p.

Dissertação(Mestrado)-- Universidade Federal do Pampa, MESTRADO EM ENGENHARIA DE SOFTWARE, 2022.
"Orientação: Claudio Schepke".

1. Interfaces de Programação Paralela. 2. OpenMP. 3. OpenACC. 4. Aplicações de Alto Desempenho. 5. Paralelismo. I. Título.

NATIELE LUCCA

AVALIAÇÃO DE ESTRATÉGIAS DE PARALELISMO EM SIMULAÇÃO DE MEIOS POROSOS

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia de Software da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Mestre em Engenharia de Software.

Dissertação defendida e aprovada em: 19 de dezembro de 2022.

Banca examinadora:

_____

Prof. Dr. Claudio Schepke

Orientador

UNIPAMPA

_____

Prof. Dr. Diego Luis Kreutz

UNIPAMPA

_____

Prof. Dr. João Vicente Ferreira Lima

UFSM

Prof. Dr. Vinícius Garcia Pinto

FURG

Assinado eletronicamente por **DIEGO LUIS KREUTZ**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 19/12/2022, às 14:45, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **CLAUDIO SCHEPKE**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 19/12/2022, às 14:48, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **Vinicius Garcia Pinto**, **Usuário Externo**, em 19/12/2022, às 14:51, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **João Vicente Ferreira Lima**, **Usuário Externo**, em 19/12/2022, às 14:52, 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 **1010434** e o código CRC **19F932C9**.

AGRADECIMENTOS

Agradeço ao apoio e paciência da minha família e do meu orientador durante a realização deste trabalho.

Agradeço a bolsa de Mestrado Profissional CAPES PDPG-FAP (PDPG Parcerias Estratégicas nos Estados - N° 42046017016P1) 2021 e 2022. O trabalho também foi realizado dentro do âmbito do projeto "Simulação Eficiente de Secagem de Grãos" com apoio da FAPERGS Edital PqG 07/2021 (Nº 21/2551-0002055-5).

RESUMO

A computação de alto desempenho provê a aceleração de aplicações científicas através do uso de paralelismo. Aplicações deste tipo geralmente demandam de um grande tempo de computação para uma versão com um único fluxo de execução de código. Diferentes modelos de programação paralela podem ser utilizados para a execução concorrente de código. Em geral, opta-se, em aplicações científicas, pela divisão do processamento dos dados. O objetivo desta dissertação é avaliar diferentes abordagens de paralelismo. Para tanto, como estudo de caso, as abordagens foram avaliadas em uma aplicação de meios porosos que provê a simulação de secagem de grãos. Foram implementadas 4 versões: OpenMP *parallel do*, *target* e *teams* e OpenACC *parallel do*. As versões OpenMP Target e OpenACC exigiram alterações na estrutura no código-fonte. Foram avaliados três tamanhos de malha para cada versão implementada, variando o número de threads em 2, 4, 8, 16 e 32. Os resultados obtidos mostram que todas as versões apresentaram ganho de desempenho. O melhor resultado para a malha I foi o caso de teste Target 32 threads com *speedup* de 3,11. Para a malha II o melhor resultado foi o caso de teste Teams 16 threads com *speedup* de 6,27. Para a malha III o resultado com maior *speedup* foi 7,54 para o caso de teste OpenACC. Foi realizada a análise estatística dos resultados obtidos com a distribuição *t de student*. A análise validou o ganho de desempenho. Também foram gerados traços das execuções paralelas que permitiram, através da ferramenta Vampir, visualizar as sincronizações entre as operações concorrentes.

Palavras-chave: Interfaces de Programação Paralela. OpenMP. OpenACC. Aplicações de Alto Desempenho. Paralelismo.

ABSTRACT

High-performance computing provides for the acceleration of scientific applications through the use of parallelism. Applications of this type usually demand a large amount of computing time for a version with a single code execution stream. There are different parallel programming models to indicate the concurrent execution of a code. The division in data decomposition processing is the best choice for scientific applications in general. This dissertation aims to evaluate different parallelism approaches for a porous media application. The application case study provides the simulation of grain drying. We implement four versions: OpenMP *parallel do*, *target* and *teams*, and OpenACC *parallel do*. The OpenMP Target and OpenACC versions required structural changes in the source code. Three mesh sizes were evaluated for each implemented version, varying the number of threads in 2, 4, 8, 16, and 32. The results obtained show that all versions showed performance gains. The best result for loop I was the Target 32 threads test case with *speedup* of 3.11. For mesh II the best result was the Teams 16 threads test case with *speedup* of 6.27. For mesh III, the highest *speedup* result was 7.54 for the OpenACC test case. We perform a statistical analysis of the results obtained with the T-Student distribution. The analysis validated the performance gain. Traces of parallel executions were also generated, which allowed, through the Vampir tool, to visualize the synchronizations among concurrent operations.

Key-words: Parallel Programing Interfaces. OpenMP. OpenACC. High Performance Computing. Parallelism.

LISTA DE FIGURAS

Figura 1 – Representação das arquiteturas *multi-core* e *many-core* 21
Figura 2 – Discretização: Método lagrangeano com malha 26
Figura 3 – Silo de secagem 38
Figura 4 – Grade numérica esquemática 39
Figura 5 – Volumes de controle discreto dentro do domínio 41
Figura 6 – Figura esquemática de volume de controle único e pontos vizinhos. . . 41
Figura 7 – Interpolação de esquema Quick para propriedades avaliadas na face da célula 42
Figura 8 – 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. 43
Figura 9 – Fluxo de execução do algoritmo 43
Figura 10 – Detalhamento das chamadas executadas pelo algoritmo 50
Figura 11 – Fluxo de manipulação das variáveis entre *host* e *device* 52
Figura 12 – Fluxo de manipulação das variáveis entre *host* e *device* 54
Figura 13 – Tempo de Execução - Resultado da malha 51x63 61
Figura 14 – Tempo de Execução - Resultado da malha 100x124 62
Figura 15 – Tempo de Execução - Resultado da malha 200x249 62
Figura 16 – Execução sequencial malha 100x124 65
Figura 17 – Exemplo - Casos de Teste OpenMP 66
Figura 18 – Casos de Teste OpenMP - Malha 51x63 66
Figura 19 – Casos de Teste OpenMP - Malha 100x124 67
Figura 20 – Casos de Teste OpenMP - Malha 200x249 68
Figura 21 – Comparativo das malhas I, II e III com 16 Threads 68

LISTA DE TABELAS

Tabela 1 – Avaliação de Desempenho da Aplicação 32
Tabela 2 – Trabalhos Relacionados 34
Tabela 3 – Avaliação de Desempenho da Aplicação 48
Tabela 4 – Ambiente de execução: CPU 58
Tabela 5 – Ambiente de execução: GPU 58
Tabela 6 – Resultados Obtidos nos Testes 59
Tabela 7 – Resultados da Distribuição T de Student 63

LISTA DE ALGORITMOS

Algoritmo 1 – Exemplo do uso da diretiva `!$omp parallel for` 29
Algoritmo 2 – Exemplo do uso da diretiva `!$omp target` 30
Algoritmo 3 – Exemplo do uso da diretiva `!$omp target` 30
Algoritmo 4 – Exemplo do uso da diretiva `!$omp target` 30
Algoritmo 5 – Exemplo do uso da diretiva `!$acc parallel loop` 32

Algoritmo 6 – Estrutura do Código original 44

Algoritmo 7 – Exemplo do uso da diretiva `!$omp parallel do` 49
Algoritmo 8 – Exemplo do uso da diretiva `!$omp parallel sections` 49
Algoritmo 9 – Exemplo do código original 51
Algoritmo 10 – Exemplo do código modificado 51
Algoritmo 11 – Estrutura original 53
Algoritmo 12 – Exemplo do uso da diretiva `!$acc parallel loop` 53
Algoritmo 13 – Exemplo do uso da diretiva `!$omp target map` 54
Algoritmo 14 – Exemplo do uso da diretiva `!$omp teams` 55
Algoritmo 15 – Exemplo do uso da diretiva `!$omp teams distribute` 55
Algoritmo 16 – Exemplo do paralelismo aplicado em cada propriedade física 56
Algoritmo 17 – Comparação de números ponto flutuante 57

LISTA DE SIGLAS

DEM *Discrete Element Method*

FEM *Finite Element Method*

FVM *Finite Volume Method*

HPC *High Performance Computing*

MIMD *Multiple Instruction Multiple Data*

OpenMP *Open Multi-Processing*

SIMD *Single Instruction Multiple Data*

SUMÁRIO

1 INTRODUÇÃO 21
1.1 Objetivo 22
1.2 Justificativa 22
1.3 Organização do Trabalho 23

2 ASPECTOS CONCEITUAIS 25
2.1 Características de Aplicações Científicas 25
2.2 Discretização 25
2.2.1 Método dos Elementos Finitos (FEM) 26
2.2.2 Método dos Volumes Finitos (FVM) 27
2.2.3 Método dos Elementos Discretos (DEM) 27
2.3 Modelos de Programação HPC 28
2.3.1 OpenMP 28
2.3.2 OpenMP Target 29
2.3.3 OpenACC 31
2.3.4 Equivalência entre as Interfaces de Programação Paralela 31
2.3.5 Considerações Finais sobre o Capítulo 32

3 TRABALHOS RELACIONADOS 33
3.1 Considerações do Capítulo 35

4 ESTUDO DE CASO 37
4.1 Método para Secagem de Grãos 37
4.2 Modelagem Matemática 38
4.3 Decomposição por Volumes Finitos 41
4.4 *Quadratic Upstream Interpolation* 42
4.5 Algoritmo 42
4.6 Considerações Finais do Capítulo 45

5 ESTRATÉGIAS METODOLÓGICAS 47
5.1 Estratégia de Paralelismo 47
5.1.1 Paralelismo com OpenMP 48
5.1.2 Tratamento no código para executar na GPU 49
5.1.3 Paralelismo com OpenACC 51
5.1.4 Paralelismo com OpenMP Target 53
5.2 Parâmetros de Entrada 56
5.3 Compatibilidade Numérica 56
5.4 Métricas de Avaliação 57
5.5 Ambiente de Execução 58

6 ANÁLISE EXPERIMENTAL 59
6.1 Resultados das Execuções **59**
6.2 Discussão dos Resultados **60**
6.3 Validação Numérica das Soluções **63**
6.4 Distribuição *t de Student* **63**
6.5 Análise com Score-P e Vampir **65**

7 CONSIDERAÇÕES FINAIS 71
7.1 Conclusão **71**
7.2 Trabalhos Futuros **72**
7.3 Publicações **72**

REFERÊNCIAS 75

Índice 81

1 INTRODUÇÃO

A Computação de Alto Desempenho (*High Performance Computing* (HPC)) é uma abordagem que fornece poder computacional através da paralelização e é aplicada na resolução de problemas complexos (VETTER, 2013). A complexidade de um programa é mensurada pelo número de variáveis, tempo de execução, dependências de dados e instruções. Esses problemas complexos incluem a modelagem e a simulação de fenômenos físicos, produtos de engenharia ou ciência. São exemplos de aplicação a simulação do clima, o controle da distribuição de energia/eletricidade e o design de peças e veículos.

Cada arquitetura dispõe de métodos paralelos distintos e é eficiente sobre características distintas (COURTÈS, 2013). Na Figura 1 pode-se perceber os elementos que representam as diferenças entre a arquitetura de uma CPU e a arquitetura de uma GPU. À esquerda tem-se a representação de uma CPU multi-core e à direita de uma GPU many-core. As GPUs dispõem de recursos paralelos diferentes das CPUs, ou seja, são programadas em uma linguagem específica, como por exemplo, para GPU existem as interfaces CUDA e OpenACC; e para CPU pode-se utilizar OpenMP e PThreads. Consequentemente, o desafio é desenvolver aplicações que explorem os recursos computacionais disponíveis.

De acordo com a lei de Moore, a capacidade de processamento aumenta proporcionalmente a cada período de tempo. Esse aumento na capacidade de desempenho dos processadores ocorre devido ao pipeline, estruturas superescalares, paralelismo a nível de instrução, níveis e tamanho das caches e altas frequências de clock (PRATX GUILLEM E XING, 2011). No entanto, essa progressão foi atenuada devido a alta dissipação de energia quando o processamento atinge altas taxas de *clock*, o que motivou o surgimento das arquiteturas multi-core e a exploração de GPUs. Cada vez mais os algoritmos complexos, como por exemplo algoritmos de simulações físicas, exigem técnicas e métodos de programação paralela para reduzir o tempo de processamento (PRATX GUILLEM E XING, 2011).

Esses algoritmos de simulações em suma possuem grande volume de dados, leitura/gravação de parâmetros via arquivo, estruturas de dados complexas e custo de cálculos matemáticos (formulações). São exemplos, algoritmos que utilizam simulações do método

Figura 1 – Representação das arquiteturas *multi-core* e *many-core*

(NVIDIA, 2020)

de elementos discretos (DEM), métodos dos elementos finitos (FEM) ou método dos volumes finitos (FVM) (XU et al., 2011). Os algoritmos possuem alto custo de execução, devido a isso muitos testes são realizados com cenários/malhas reduzidos e com partículas e elementos que representam grandes escalas. Essa redução da malha e dos parâmetros se faz necessária para que os modelos de DEM, FEM e FVM possam ser validados pelas simulações. À medida que o poder da computação paralela vem se desenvolvendo, é possível aumentar o número de partículas e elementos avaliados nos modelos de simulação, e consequentemente prover melhores resultados numéricos (XU et al., 2011).

A interface que melhor explora o algoritmo é a que reduz o tempo de execução e mantém o resultado factível, ou seja, explora os recursos da API para distribuir a carga de trabalho de forma a utilizar os recursos disponíveis da arquitetura. Dentre as interfaces de programação tradicionais consolidadas na literatura destacam-se OpenMP, OpenACC, MPI e CUDA (ANSARI et al., 2015). OpenMP, por exemplo, distribui a carga de trabalho entre *threads* e utiliza diretivas para especificar os blocos de execução paralela. OpenACC também utiliza diretivas, mas é utilizado na arquitetura de GPU para grandes volumes de dados. MPI não utiliza diretivas e sim funções que distribuem os métodos entre processos ou até mesmo entre dispositivos distintos. CUDA também não utiliza diretivas, sendo necessário alterar o código para que o paralelismo possa ser executado.

1.1 Objetivo

O objetivo geral desta dissertação é avaliar o desempenho da programação de aplicações científicas em estratégias paralelas distintas. Para tanto, foram consideradas interfaces de programação que explorem arquiteturas distintas.

Como delimitação do tema escolheu-se uma aplicação de simulação de meios porosos como estudo de caso. Para avaliar o impacto do uso de uma interface de programação paralela baseada em paralelismo de tarefas, optou-se por avaliar as interfaces OpenMP e OpenACC.

As contribuições do trabalhos são:

- Avaliar o desempenho do estudo de caso submetido às versões da aplicação com diferentes abordagens paralelas oferecidas por OpenMP e OpenACC.
- Demonstrar o *profiling* dos versões através da visualização de traços das execuções.

1.2 Justificativa

A grande maioria das aplicações científicas limita-se à exploração do paralelismo de maneira tradicional, decompondo dados. Essas aplicações quando usam mais de um tipo de arquitetura são programadas com duas ou mais interfaces de programação. A pro-

posta deste trabalho é avaliar o desempenho de diferentes APIs de programação paralela submetidos ao estudo de caso Simulação de Meios Porosos.

1.3 Organização do Trabalho

O restante do documento está estruturado da seguinte forma. No Capítulo 2 são discutidos conceitos relacionados à formulação numérica/computacional dos problemas, o que inclui abordagens de discretização e modelos de programação paralela. O Capítulo 3 apresenta a revisão da literatura. No Capítulo 4 é apresentada a aplicação sequencial que será utilizada como estudo de caso. A aplicação consiste de uma simulação de secagem de grãos, como meios porosos. A metodologia de desenvolvimento e avaliação dos experimentos é definida no Capítulo 5. No Capítulo 6 são mostrados os resultados obtidos nos experimentos. Por fim, a conclusão e os trabalhos futuros estão apresentadas no Capítulo 7.

2 ASPECTOS CONCEITUAIS

Este capítulo aborda conceitos utilizados no desenvolvimento do trabalho. Primeiramente são apresentadas algumas características das aplicações científicas. Na sequência são descritos os métodos de discretização *Finite Element Method* (FEM), *Finite Volume Method* (FVM) e *Discrete Element Method* (DEM) para a simulação computacional dos algoritmos. Também são apresentadas as APIs utilizadas para desenvolver as versões do algoritmo. Por fim, são avaliadas as interfaces de programação tradicionais existentes e as interfaces baseadas em tarefas.

2.1 Características de Aplicações Científicas

Aplicações científicas representam computacionalmente um problema. Os modelos descrevem a relação causa e efeito entre as variáveis de entrada e saída de uma aplicação (AGUIRRE, 2004). Ou seja, o objetivo é representar as características de uma aplicação quando submetida a ações externas que podem alterar seu estado atual. São exemplos de ações externas a simulação do comportamento de um material como um parafuso submetido a temperaturas elevadas ou observar a deformação de uma barra de ferro sob compressão.

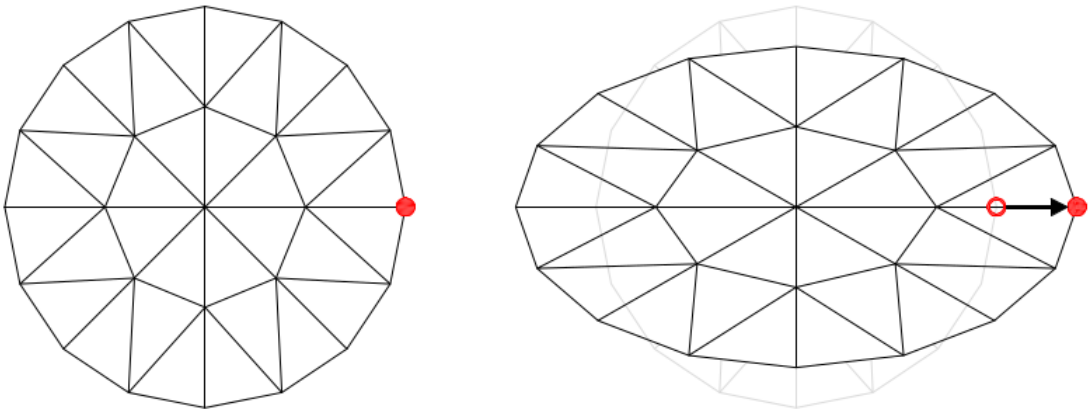
A análise experimental de problemas científicos como sistemas físicos, estruturais, aeroespaciais, meteorológicos e de energia são inviáveis devido ao elevado custo para produção dos testes e risco de falha (GOULD; TOBOCHNIK; CHRISTIAN, 2016). Uma alternativa para reduzir esses custos é simular o domínio e as suas características em softwares especializados. As simulações exigem alto poder de processamento e técnicas adequadas para explorar os recursos disponíveis e obter resultados precisos.

Os modelos ou domínios podem representar problemas físicos, matemáticos, industriais, como por exemplo: transferência de calor, elastodinâmica, eletroestática, eletromagnetismo, acústica, mecânica da fratura, hidráulica, hidrodinâmica, aerodinâmica, lubrificação, problemas de interação fluido-estrutura, e problemas de propagação de ondas. Dentre esses domínios podem ser analisadas as seguintes características: condução de calor, condução elétrica, campos gravitacionais, campos eletroestáticos, campos magnetoestáticos, fluxo irrotacional de fluidos ideais, percolação através de um meio poroso, torção de barras e pressão (SOUZA, 2003).

2.2 Discretização

A discretização transforma atributos contínuos em atributos discretos (GARCIA et al., 2012). Dessa forma, os dados são discretizados, ou seja, são reduzidos e simplificados. O processo converte equações de domínio contínuo para equações algébricas de domínio discreto (ALVES, 2007). Esse processo de conversão é estabelecido de acordo com o método aplicado e tem por objetivo permitir que um modelo seja representado matema-

Figura 2 – Discretização: Método lagrangeano com malha



(FIGUEIREDO, 2016)

ticamente (ANTON; BIVENS; DAVIS, 2014). Essa conversão mantém as características do domínio para que o resultado seja factível ao do modelo contínuo original.

A discretização é tradicionalmente classificada de acordo com o sistema de referência euleriano ou lagrangeano. Os métodos eulerianos utilizam uma malha para representar o volume da peça ou a seção no espaço. Através da discretização é possível observar o comportamento do fluido que passa pela peça ou seção. Já os métodos lagrangeanos se subdividem em métodos que utilizam uma malha que se movimenta de acordo com a trajetória do fluido e outros que não utilizam malhas. Também existem métodos híbridos que utilizam ambas abordagens (FIGUEIREDO, 2016).

Os métodos lagrangeanos podem ser representados com ou sem malha. Os métodos com malha possuem uma quantidade fixa de pontos e a malha se deforma ao passar do tempo de acordo com o escoamento. O método dos elementos finitos é um exemplo da discretização lagrangeanos com malha, ilustrado na Figura 2. Já os métodos lagrangeanos sem malha são utilizados para representar grandes deformações e geometrias complexas. O método considera um conjunto de pontos não estruturados distribuídos no domínio, além dos pontos são consideradas as distâncias entre os pontos, essas distâncias alteram no decorrer do tempo. Entretanto, para obter precisão no resultado é necessário utilizar muitos pontos (FIGUEIREDO, 2016). O método de Lattice-Boltzmann é um exemplo de método de discretização sem malha.

Neste trabalho são destacados os métodos de discretização FEM, FVM e DEM.

2.2.1 Método dos Elementos Finitos (FEM)

O método dos elementos finitos é um modelo de discretização aproximada que subdivide o domínio em áreas menores (MITCHELL, 2014). Essas áreas menores possuem dimensões finitas e são chamadas de elementos finitos. Cada elemento contém as mesmas propriedades do domínio de origem e são conectados através de pontos, também

denominados como nós ou pontos nodais. O conjunto de elementos finitos e pontos nodais é conhecido como malha de elementos finitos.

A precisão do FEM depende da quantidade de nós e elementos. Embora seja um método aproximado, à medida que o tamanho dos elementos finitos tende a zero a quantidade de nós tende a infinito e a solução converge para a solução exata do problema (SOUZA, 2003). Ou seja, quanto menor for o tamanho e maior for o número de elementos em uma determinada malha, mais precisos serão os resultados da análise.

2.2.2 Método dos Volumes Finitos (FVM)

Moukalled et al. (2016) define o Método de Volume Finito (do inglês Finite Volume Method - FVM) como uma técnica que transforma equações diferenciais parciais que representam leis de conservação sobre volumes diferenciais em equações algébricas discretas sobre volumes finitos. Assim como no método dos elementos finitos (FEM), a primeira etapa do processo de solução é a discretização do domínio geométrico. No FVM o domínio é discretizado em volumes finitos.

Dentre as propriedades do FVM destacam-se a possibilidade de modelar malhas poligonais não estruturadas e a forma com que o método calcula as condições de contorno. As condições de contorno são calculados pelo centroide e não pelas bordas do elemento, assim preservando as características locais do domínio (MOUKALLED et al., 2016). Dessa forma, o método possibilita a resolução de problemas complexos como por exemplo mecânica dos fluidos e dissipação de calor e de massa (EYMARD; GALLOUëT; HERBIN, 2000). Esses problemas não são representados de forma regular e possuem uma dependência causada pelo fluxo do elemento.

2.2.3 Método dos Elementos Discretos (DEM)

O método dos Elementos Discretos (DEM) é aplicado para simular o comportamento de corpos discretos em interação. Ou seja, o método simula o contato entre as partículas do domínio (MISHRA; RAJAMANI, 1992). Entretanto, o DEM só pode ser modelado se o problema permitir deslocamentos finitos e rotações nos elementos discretos. O DEM é utilizado para analisar e otimizar sistemas de manuseio de materiais a granel (COETZEE, 2017).

De acordo com Mustoe (1992) o DEM simula os fluxos de partículas de grânulos, o que possibilita a análise dos efeitos de cisalhamento. Esse método pode modelar a mecânica de rochas, fratura, britagem, corte e estabilidade ou falha do declive. Nessas aplicações, o DEM permite a análise contínua do material até a fratura ou a falha ocorrer. O DEM modela as descontinuidades devido a fratura, falha ou a remoção de pequenas partículas (COETZEE, 2017).

2.3 Modelos de Programação HPC

No paralelismo de Tarefas as aplicações são divididas em tarefas exclusivas que são independentes umas das outras e podem ser executadas em processadores diferentes (GALARZA, 2020). De acordo com a taxonomia de Flynn a arquitetura *Multiple Instruction Multiple Data* (MIMD) é um exemplo, pois as unidades de processamento operam instruções distintas.

O paralelismo de dados é uma técnica de programação que divide uma grande quantidade de dados em partes menores que podem ser operadas em paralelo. A arquitetura *Single Instruction Multiple Data* (SIMD) utiliza paralelismo de dados, ou seja, as unidades de processamento compartilham a mesma instrução, mas realizam a operação sobre dados distintos.

A interface de programação empregada no paralelismo de uma aplicação é de suma importância para a obtenção de ganho de desempenho. A interface utilizada deve atender às características da aplicação paralelizada, de forma a explorar o paralelismo em GPU, CPU ou híbrido.

Dentre as APIs que existem para programação, cada uma explora uma característica. Por exemplo, uma aplicação com grande volume de dados e com poucas instruções tende a usar o paralelismo de dados. Para aplicações com laços de repetição, é vantagem fazer a transferência dos dados para GPU e efetuar o processamento, como é o caso de uma multiplicação de matrizes. Já aplicações robustas com diversos fragmentos de código podem ser explorados por diferentes estratégias de paralelismo de forma a obter maior desempenho.

Cada estratégia de paralelismo é eficiente sobre certas características. A proposta deste trabalho é avaliar o desempenho de uma aplicação discretizada por FVM com paralelismo OpenMP, OpenACC e OpenMP Target.

Na sequência são descritas algumas das principais interfaces de programação paralela disponíveis para paralelismo de aplicações.

2.3.1 OpenMP

OpenMP é um modelo de programação amplamente usado para arquiteturas de memória compartilhada (DURAN et al., 2008). A interface suporta programação multiprocessador e multi-plataforma em linguagem C/C++ e Fortran. OpenMP possui um modelo de programação baseado em tarefas e em *loop* (FRIGO; LEISERSON; RANDALL, 1998).

OpenMP surgiu da necessidade de padronizar as linguagens de diretiva de vários fornecedores na década de 1990. Foi estruturado em torno de *loops* paralelos e foi criado para lidar com aplicações numéricas densas. A simplicidade de sua interface original, o uso de um modelo de memória compartilhada e o fato de que o paralelismo de um programa

é expresso em diretivas que são fracamente acopladas ao código, tornaram OpenMP bem aceito no contexto do desenvolvimento de aplicações paralelas (DURAN et al., 2008).

A partir da versão 4.0, o OpenMP possui diretivas específicas para tratar o paralelismo em GPU e paralelismo SIMD. É possível definir recursos de tratamento de erros e definir explicitamente reduções. A versão 5 do OpenMP possui suporte para as versões mais recentes de C, C++ e Fortran, como por exemplo C++11 (OPENMP, 2018).

As diretivas OpenMP são padronizadas por `#pragma omp` para C/C++ e `$omp` para Fortran seguida por `[atributos]`, sendo que os atributos são opcionais. Seguem algumas diretivas que compõem a API OpenMP (OPENMP, 2018). São descritas algumas diretiva compatível com C/C++ e Fortran.

- `parallel`: Essa diretiva descreve que a uma área do código será executada por n *threads*, sendo n o número de *threads* especificados por um atributo e/ou variável de ambiente.
- `do`: Essa diretiva especifica que as iterações do laço de repetição serão executadas em paralelo por n *threads*.
- `parallel do`: Especifica a construção de um laço paralelo, sendo que o laço será executado por n *threads*.
- `simd`: Essa diretiva descreve que algumas iterações de um laço de repetição podem ser executadas simultaneamente por unidades vetoriais.
- `do simd`: Essa diretiva especifica que um laço pode ser dividido em n *threads* que executam algumas iterações simultaneamente por unidades vetoriais.

Algoritmo 1: Exemplo do uso da diretiva `!$omp parallel for`

```
1          !$omp parallel do private(i,j)
2              DO i=0,imax
3                  DO j=0,jmax
4                      c(i,j) = a(i,j) * b(i,j)
5                  ENDDO
6              ENDDO
7          !$omp end parallel do
```

2.3.2 OpenMP Target

Diferente do OpenMP clássico a diretiva *target* especifica que o determinado bloco seja executado em um *device*. Essa transferência de dados ocorre entre um *host* (CPU) e um *device* (GPU). Assim como no OpenACC podem ser especificadas cláusulas na diretiva (BERTOLLI et al., 2014).

As diretivas OpenMP Target são padronizadas por `#pragma omp target` para C/C++ e para Fortran a seguinte estrutura `$omp target` ... `$omp end target`. A sintaxe no OpenMP Target é apresentada no Algoritmo 2.

Algoritmo 2: Exemplo do uso da diretiva `!$omp target`

```
1          !$omp target map(to: a, b) map(from:c)
2          !$omp teams distribute parallel do
3              DO i=0,imax
4                  DO j=0,jmax
5                      c(i,j) = a(i,j) * b(i,j)
6                  ENDDO
7              ENDDO
8          !$omp end teams
9          !$omp end target
```

A clausula *map(...)* especifica as transferências das variáveis manipuladas seja de *read*(clausula to) ou *write*(clausula from). No exemplo algoritmo 3 o uso do *to* e *from* discrimina as direções de cópia entre os ambientes de dados do *host* e do *device*.

Algoritmo 3: Exemplo do uso da diretiva `!$omp target`

```
1          !$omp target map(to: a, b) map(from: c)
2              DO i=0,imax
3                  DO j=0,jmax
4                      c(i,j) = a(i,j) * b(i,j)
5                  ENDDO
6              ENDDO
7          !$omp end target
```

Algoritmo 4: Exemplo do uso da diretiva `!$omp target`

```
1          !$omp target map(to: a, b) map(from: c)
2              !$omp teams
3                  b += 1
4                  !$omp parallel do
5                  DO i=0,imax
6                      DO j=0,jmax
7                          c(i,j) = a(i,j) * b(i,j)
8                      ENDDO
9                  ENDDO
10                 c *= 2
11             !$omp end teams
12         !$omp end target
```

A diretiva de *teams* agrupa as threads em equipes. Dessa forma, é criada uma abstração de paralelismo onde as threads da equipe colaboram. Podem ser especificadas

as cláusulas *num_team* e *thread_limit* que definem o número de equipes que devem ser criadas e o número limite de threads que podem ser alocadas. Cada equipe possui uma thread mestre. Quando a diretiva *teams* é aplicada as threads mestres executam sequencialmente as próximas instruções até identificar uma área *parallel*. O trabalho somente é distribuído entre as threads do time quando é especificado uma diretiva *parallel*.

No exemplo apresentado no algoritmo 4, a linha 3 é executada somente pela thread mestre de cada equipe. Enquanto, as linhas 5-9 são distribuídas entre o time de threads. O uso da diretiva *teams* não é obrigatório.

2.3.3 OpenACC

OpenACC é uma API desenvolvida em 2011 para programação em arquiteturas heterogêneas com aceleradores. O modelo de programação usa diretivas de alto nível, assim como o OpenMP, para expressar as áreas ou blocos paralelos (CHANDRASEKARAN; JUCKELAND, 2017).

A API está disponível para as linguagens C/C++ e FORTRAN (CHANDRASEKARAN; JUCKELAND, 2017). Assim como OpenMP, OpenACC possui diretivas padronizadas por `!$acc` para Fortran e `#pragma acc` para C/C++, seguida de `[atributos]` opcionais. Essas diretivas especificam *loops* e blocos de código que podem ser enviados da CPU para um acelerador. A seguir, são descritas algumas diretivas compatíveis com C/C++ e Fortran.

- `data copy`: Essa diretiva copia dados entre CPU e GPU.
- `kernels`: A diretiva permite que o compilador tome as decisões de paralelismo.
- `parallel`: A diretiva impõem ao compilador que o trecho de código a seguir deve ser paralelizado. Dependendo do uso a diretiva pode produzir resultados equivocados.
- `present`: A diretiva especifica para o compilador que as variáveis estão presentes na memória da GPU e não é necessário fazer cópia.
- `loop independent`: Essa diretiva especifica que o *loop* pode ser executado independentemente.

2.3.4 Equivalência entre as Interfaces de Programação Paralela

Apesar das interfaces OpenMP, OpenACC e OpenMP Target utilizarem diretivas para especificar o paralelismo, cada interface tem suas características e diretivas conforme apresentado anteriormente.

A Tabela 1 apresenta de forma resumida uma comparação da equivalência entre a sintaxe do OpenMP clássico com diretivas que são executadas em CPU, do OpenACC e do

Algoritmo 5: Exemplo do uso da diretiva `!$acc parallel loop`

```
1 !$acc parallel loop collapse(2)
2     DO i=2,imax
3         DO j=2,jmax
4             a(i,j) = b(i,j) * c(i,j)
5         ENDDO
6     ENDDO
7 !$acc end parallel
```

Tabela 1 – Avaliação de Desempenho da Aplicação

OpenMP CPU	**OpenACC**	**OpenMP Target (GPU)**
omp parallel	acc parallel	omp target
omp parallel for	acc loop gang	omp teams distribute
-	acc loop vector	omp parallel for
-	acc data copyin(<var>)	omp data map(to:<var>)
-	acc data copyout(<var>)	omp data map(to:<var>)
-	acc data create(<var>)	omp data map(alloc:<var>)
omp parallel simd	-	omp parallel simd
omp task	-	omp task
-	routine	-
-	acc update	target update

OpenMP Target para execução de tarefas em GPU. As APIs diferem em sua abordagem para especificar a paralelização. Como todas as versões utilizam diretivas, a proposta é especificar as equivalências entre as APIs. Embora as diretivas sejam semelhantes em cada API, o paralelismo é especificado de uma forma diferente.

2.3.5 Considerações Finais sobre o Capítulo

Neste capítulo apresentou-se os métodos de discretização FEM, FVM e DEM e os modelos de programação OpenMP, OpenACC e OpenMP Target com exemplos das diretivas utilizadas. Também foram apresentadas algumas tabelas com as diretivas que são equivalentes em cada API.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta alguns trabalhos relacionados encontrados na literatura. Para a pesquisa foram utilizadas as palavras-chaves de busca *OpenMP*, *OpenACC* e *Target*. Notou-se uma dificuldade em identificar trabalhos que abordassem as três APIs, os artigos encontrados utilizavam apenas duas abordagens.

A Tabela 2 apresenta o estudo selecionado, destacando se o estudo avalia a API OpenMP clássica (somente executada na CPU), OpenACC ou OpenMP Target. A coluna "API" é utilizada para especificar qual outra API o estudo avaliou. Enquanto a última coluna destaca o *speedup* apresentado no estudo.

O trabalho (GUO et al., 2016) apresenta um estudo sobre a reflexão aérea do alvo da radiação do Sol na superfície da Terra e da atmosfera. Esse estudo compara as implementações paralelas em CPU multi-core usando OpenMP e em GPU usando OpenACC e CUDA. O estudo investiga o cálculo paralelo da reflexão de alvos aéreos da radiação do Sol, da superfície da Terra e da atmosfera. Considerando a versão sequencial como *baseline* a versão OpenMP obteve uma aceleração de aproximadamente 15x. A implementação OpenACC teve uma aceleração de 140x Já a implementação CUDA obteve uma aceleração de 426x.

O trabalho (DALEY et al., 2020) compara o desempenho de paralelismo entre CUDA e OpenMP Target. Esse trabalho destaca que a avaliação de desempenho entre OpenMP Target e CUDA é feita com a aplicação não trivial HPGMG. HPGMG é um benchmark de volume finito multigrid. O estudo não apresenta resultados referentes a OpenMP Target, pois a versão apresenta bugs. A versão CUDA apresentou aceleração de 0,70x no caso de teste I e 2,04x no caso de teste II e 0,73x no caso de teste III.

O trabalho (LARREA et al., 2020) utiliza a linguagem de programação Fortran. A aplicação estudada representa os somatórios de energia própria do Pólo de Plasmon Geral e Frequência total. O trabalho descreve o número de equipes, número de threads, informações omitidas em outros trabalhos. O estudo apresenta as estratégias utilizadas para transcrever as versões OpenMP Target e OpenACC. São informados os tempos de execução, mas as versões não são comparadas. Foco na portabilidade.

O trabalho (KHALILOV; TIMOVEEV, 2021) compara o desempenho das APIs OpenACC, CUDA e OpenACC Target. O trabalho avaliar o desempenho das bibliotecas de álgebra linear BLAS e cuBLAS da Nvidia em GPUs e Intel (Math Kernel Library) para CPUs. O estudo analisou soma, multiplicação de matrizes e identificou que para casos simples o desempenho entre as três APIs é muito semelhante. Entretanto, a medida que a complexidade aumenta (tamanho da matriz), a diferença média no tempo de execução é expressiva. Segundo o autor o OpenMP é 80% mais lento que a versão do código CUDA. A versão CUDA apresentou em média os melhores resultados.

Em (Lobato Gimenes; Pisani; Borin, 2018) é apresentada uma comparação entre as interfaces CUDA, OpenCL, OpenACC e OpenMP. A aplicação estudada é eficiência

Tabela 2 – Trabalhos Relacionados

Estudo	**OpenMP**	**OpenACC**	**Target**	**API**	***Speedup***
(GUO et al., 2016)	Sim	Sim	Não	CUDA	15, 140 e 426
(DALEY et al., 2020)	Não	Não	Sim	CUDA	2,04
(LARREA et al., 2020)	Não	Sim	Sim	–	–
(KHALILOV; TIMOVEEV, 2021)	Não	Sim	Sim	CUDA	–
(Lobato Gimenes; Pisani; Borin, 2018)	Sim	Sim	Não	CUDA e OpenCL	–
(GOYAL; LI; KIMM, 2017)	Sim	Sim	Não	MPI	6,50
(SIMANJUNTAK; GUNAWAN, 2017)	Sim	Não	Não	–	2,86
(SIMANJUNTAK; GUNAWAN, 2017), (JULIATI; GUNAWAN, 2017)	Sim	Não	Não	–	6,05
(GUNAWAN, 2016)	Sim	Não	Não	–	2,00
(RAJ et al., 2018)	Não	Sim	Não	–	21,00

Fonte: Autora

energética e desempenho computacional, para os métodos numéricos Ponto Médio Comum e Reflexão Comum. Obteve-se um desempenho inferior em GPU usando OpenACC. 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.

(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 e MPI. OpenACC mostrou-se mais eficiente, obtendo uma aceleração de aproximadamente 6,5 vezes para os algoritmos.

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.

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 apro-

ximar 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).

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. *Open Multi-Processing* (OpenMP) é usado para a abordagem paralela, obtendo um ganho de desempenho de 2 vezes e uma eficiência de 50%.

(RAJ et al., 2018) trabalha com o método de métodos de fronteira imersa (IBM), usado para tratar fenômenos multi-físicos. 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 a CPU Intel Xeon Gold 6148 de 20 núcleos.

3.1 Considerações do Capítulo

Os trabalhos relacionados evidenciam lacunas. Os trabalhos sobre OpenMP Target descrevem a diretiva utilizada, mas não especificam, por exemplo, o número de *teams* utilizado. Alguns trabalhos não citam a compatibilidade numérica, novas versões principalmente com hardwares diferentes podem ter erros numéricos. Nenhum dos trabalhos apresenta *link* para o código original, nem para a versão desenvolvida no trabalho. A falta dessas informações tem impacto direto na reprodutibilidade dos experimentos.

4 ESTUDO DE CASO

Este capítulo descreve o estudo de caso aplicado para validar o objetivo do trabalho. O código-fonte utilizado para avaliar diferentes interfaces de programação paralelas foi desenvolvido a partir de um algoritmo de simulação de secagem de grãos, método modelado por (OLIVEIRA, 2020). A seguir é apresentado todo o desenvolvimento do problema, incluindo a formulação matemática e o fluxograma resultante para o método de resolução.

4.1 Método para Secagem de Grãos

Para os agricultores e empresas agrícolas envolvidas no manuseio e armazenamento de grãos, a secagem é uma etapa fundamental para a conservação dos grãos (BALA, 2016). Dessa forma os grãos mantêm a qualidade e podem ser armazenados por mais tempo. A secagem é um processo termodinâmico, ou seja, a umidade do corpo diminui à medida que ele é submetido a uma fonte de calor. A secagem consiste em retirar a umidade dos grãos, de acordo com (BALA, 2016) a umidade deve respeitar o intervalo de 12% à 14% de acordo com o tipo de grão. A retirada da umidade ocorre devido a diferença de pressão entre o vapor presente na superfície do grão e o ar.

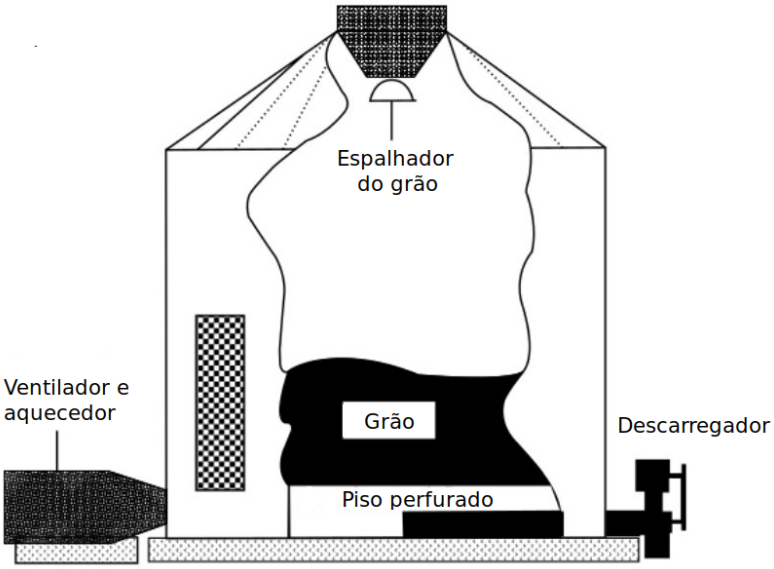
O processo de secagem artificial possui duas categorias: Silos e Secadores Portáteis. Os silos de secagem de fluxo contínuo são classificados de acordo com a direção do fluxo do ar: cruzado, concorrente e contrafluxo. Os grãos fluem pelos secadores na direção vertical, enquanto o ar pode seguir diferentes fluxos. Em secadores que utilizam fluxo cruzado, o ar é perpendicular aos grãos. Em secadores que utilizam fluxo concorrente, o ar é lançado na direção dos grãos. Já em secadores de contrafluxo ar é lançado na direção opostas aos grãos (PARRY, 1985).

Os grãos podem ficar armazenados em silos. Esses silos possuem um tubo central perfurado. Através desse tubo perfurado o ar é liberado pelos ventiladores ou aquecedores e pode circular entre os grãos, aquecendo-os ou resfriando-os. A velocidade e o tempo de secagem dependem de alguns fatores como: tamanho dos grãos, umidade relativa, sistema de secagem e teor de umidade (PARRY, 1985). A Figura 3 descreve os componentes genéricos de um silo de secagem de grãos.

A modelagem do método de secagem de grãos é representada por um fluido que escoa através de um recipiente com partículas, ou seja, o ar flui pelo silo dentre os grãos. Cada partícula tem certas propriedades físicas definidas. Se as partículas sólidas estão em movimento o problema pode ser definido com meio poroso não estacionário. Caso contrário o problema é definido como meio poroso (SOBIESKI, 2010).

O método de discretização por volumes finitos é tradicionalmente aplicado para problemas de dinâmica de fluidos por melhor se adaptar às características do problema. Dessa forma, para o problema de secagem de grãos a aplicação foi modelada computacionalmente pelo método dos volumes finitos.

Figura 3 – Silo de secagem



Fonte: Oliveira (2020)

Considerando que a massa de grãos é uma quantidade de espaços sólidos e vazios (buracos) pelo qual um fluido pode passar, pode-se assumir a secagem de grãos como um problema de meio poroso. O processo de secagem de grãos envolve transferência contínua de calor e massa entre sólidos (grãos) e fluido (ar quente e vapor). Considerando que o fluxo de ar quente preenche os espaços vazios entre os grãos, é possível identificar as características dos métodos de um meio poroso. Um meio poroso é definido como um espaço ocupado por um matéria em fase sólida e gasosa ou fase sólida e líquida.

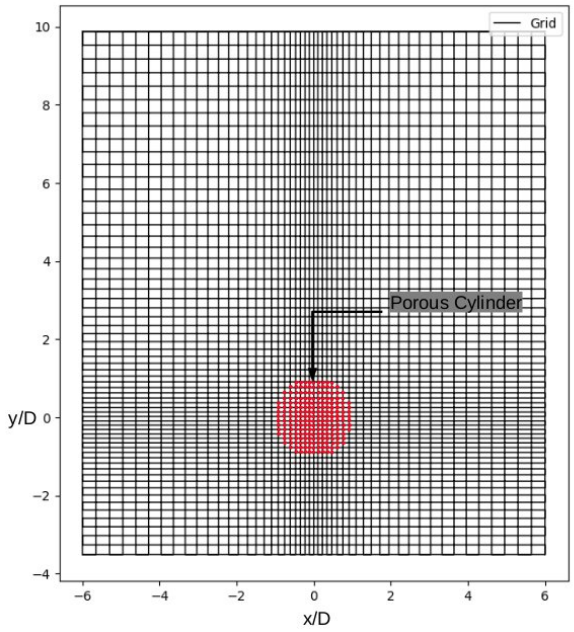
4.2 Modelagem Matemática

A modelagem matemática é utilizada para descrever a transferências de calor que ocorrem durante o processo de secagem dos grãos e permitir que o problema possa ser simulado computacionalmente. As propriedades térmicas são essenciais para simular a transferência de massa e calor.

Conforme Oliveira (2020) em estudos sobre a secagem de grãos é imprescindível a utilização das seguintes propriedades físicas: do calor específico a pressão constante (Cp), calor latente de vaporização (Qfg) e condutividade térmica (k). Para representar o meio poroso as seguintes propriedades são utilizadas: área de superfície, porosidade, tamanho dos poros, distribuição, tortuosidade e interconectividade.

A formulação do problema de meio poroso utiliza variáveis normalizadas. 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 4 mostra a grade numérica

Figura 4 – Grade numérica esquemática



Fonte: (OLIVEIRA, 2020)

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.

As equações que regem para a conservação de massa, momento e energia são dadas por equações:

$$\frac{1}{\beta}\frac{\partial p}{\partial \tau} + \frac{1}{\varepsilon}\frac{\partial u}{\partial x} + \frac{1}{\varepsilon}\frac{\partial v}{\partial y} = 0 \tag{1}$$

$$\frac{\partial u}{\partial \tau} + \frac{1}{\varepsilon}\frac{\partial (uu)}{\partial x} + \frac{1}{\varepsilon}\frac{\partial (vu)}{\partial y} = -\varepsilon\frac{\partial p}{\partial x} + \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + \varepsilon B\Phi_x \tag{2}$$

$$\frac{\partial v}{\partial \tau} + \frac{1}{\varepsilon}\frac{\partial (uv)}{\partial x} + \frac{1}{\varepsilon}\frac{\partial (vv)}{\partial y} = -\varepsilon\frac{\partial p}{\partial y} + \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) + \varepsilon B\Phi_y + \varepsilon F_g \tag{3}$$

$$\frac{\partial (T)}{\partial \tau} + \frac{1}{\varepsilon}\frac{\partial (uT)}{\partial x} + \frac{1}{\varepsilon}\frac{\partial (vT)}{\partial y} = \frac{1}{Pe}\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) \tag{4}$$

O meio poroso é modelado pela equação de Darcy-forchheimer, onde Φ é o arrasto de fricção interno dado por

$$\Phi_x = -\frac{u}{Re.Da} + \frac{c_f}{\sqrt{Da}} u\sqrt{u^2+v^2} \tag{5}$$

$$\Phi_y = -\frac{v}{Re \cdot Da} + \frac{c_f}{\sqrt{Da}} v\sqrt{u^2+v^2} \tag{6}$$

em que Re é o número de Reynolds de fluxo livre, Da é o número de Darcy definido como $Da = Kp/d^2$. O parâmetro c_f é uma constante de arrasto de forma adimensional. c_f é o coeficiente de Forchheimer e para um cilindro preenchido com esferas homogêneas a expressão adotada é:

$$C_f = \frac{1.75}{\sqrt{150\varepsilon^3}} \tag{7}$$

As condições iniciais e de contorno para as equações são especificadas como condições iniciais ($\tau = 0$):

$$u(x,y) = 0, \quad v(x,y) = v_i \quad \text{todo lugar} \tag{8}$$

$$T(x,y) = \begin{cases} T_\infty, & \text{fora do meio poroso} \\ T_p, & \text{dentro do meio poroso} \end{cases} \tag{9}$$

onde T_p é a temperatura do meio poroso.

As condições de contorno ($\tau > 0$):

$$u\left(x,-L_y\right) = 0, \quad v\left(x,-L_y\right) = v_i, \quad T\left(x,-L_y\right) = T_\infty \quad \text{no limite de entrada} \tag{10}$$

$$u\left(x,L_y\right) = 0, \quad v\left(x,L_y\right) = v_i, \quad T\left(x,L_y\right) = T_\infty \quad \text{no limite de saída} \tag{11}$$

$$u\left(-L_x,y\right) = 0, \quad \left.\frac{\partial v}{\partial x}\right|_{(-Lx,y)} = 0, \quad T\left(-L_x,y\right) = T_\infty \quad \text{no lado esquerdo} \tag{12}$$

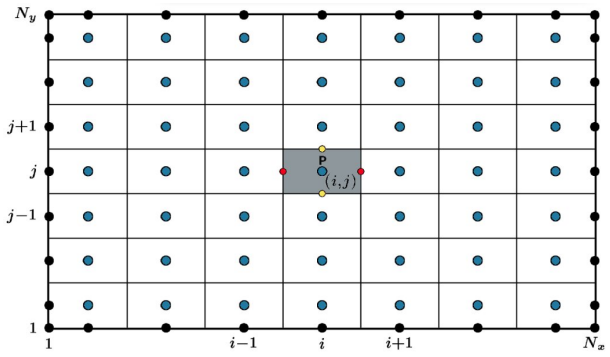
$$u\left(L_x,y\right) = 0, \quad \left.\frac{\partial v}{\partial x}\right|_{(Lx,0)} = 0, \quad T\left(L_x,y\right) = T_\infty \quad \text{no lado direito} \tag{13}$$

O centro do cilindro está localizado na origem do sistema cartesiano como mostrado em Figura 4.

4.3 Decomposição por Volumes Finitos

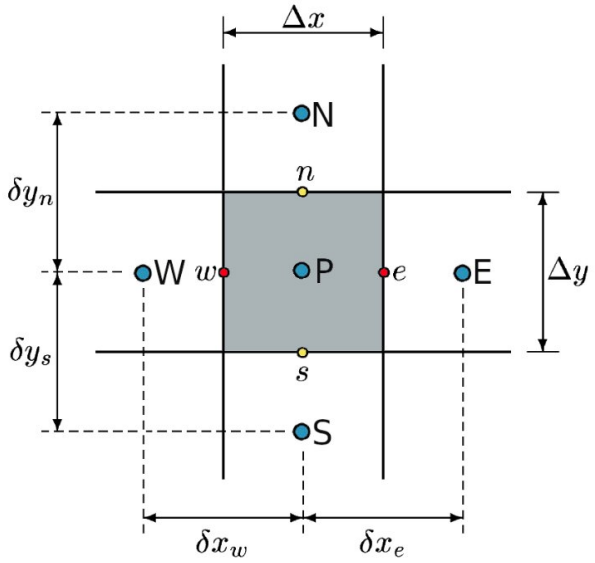
O Método de Volumes Finitos (FVM) é apresentado na seção 2.2.2. Neste método, o domínio é dividido em volumes de controle discretos, onde cada nó $P(i, j)$ é circundado por um volume de controle. Os limites dos volumes de controle são posicionados no meio do caminho entre os nós adjacentes. Dessa forma, os limites físicos coincidem com os limites do volume de controle, como mostrado no Figura 5. Cada ponto $P(x, y)$ é delimitado por pontos vizinhos colocados na interface entre os dois volumes adjacentes denominados $n(x, y)$, $s(x, y)$ para norte e sul, $w(x, y)$, $e(x, i)$ para leste e oeste. Da mesma forma, os pontos colocados no centro dos volumes de controle vizinhos são definidos como $N(x, y)$, $S(x, y)$ para norte e sul, $W(x, y)$, $E(x, y)$ para leste e oeste, veja Figura 6.

Figura 5 – Volumes de controle discreto dentro do domínio



Fonte: Oliveira (2020)

Figura 6 – Figura esquemática de volume de controle único e pontos vizinhos.



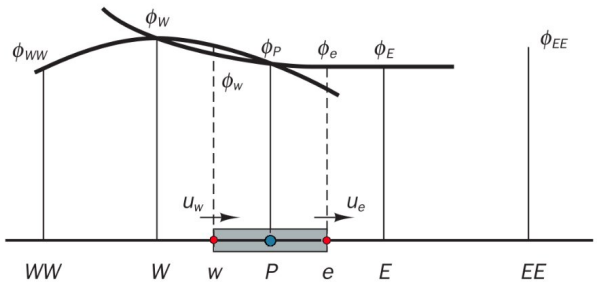
Fonte: Oliveira (2020)

4.4 *Quadratic Upstream Interpolation*

A *Quadratic Upstream Interpolation* (Quick) para Cinemática Convectiva, é um método utilizado para resolver problemas de convecção, baseado em uma formulação conservadora de controle de volume integral. Este método considera a 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 direção 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 7. 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 8. Este procedimento é conhecido como grade escalonada. Assim, o método Quick é usado e as velocidades nas interfaces de volume são calculadas por uma interpolação quadrática em esquema *upwind*.

Figura 7 – Interpolação de esquema Quick para propriedades avaliadas na face da célula

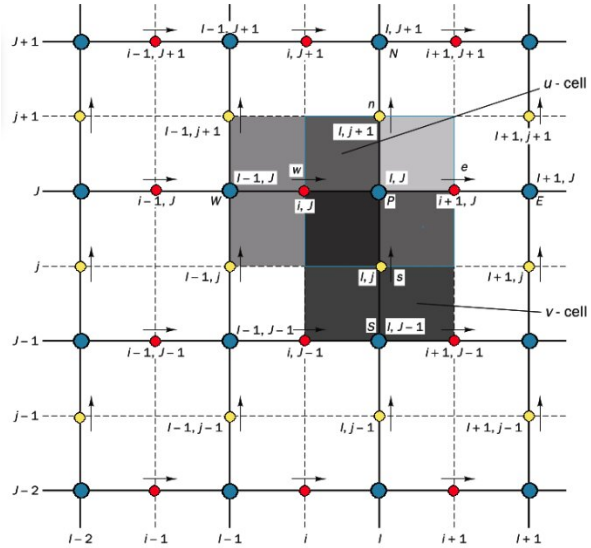


Fonte: Oliveira (2020)

4.5 Algoritmo

O algoritmo foi desenvolvido por (OLIVEIRA, 2020). Na primeira etapa os parâmetros de entrada são inicializados, são eles: número de iterações, tamanho da malha. Na sequência é calculado o método que calcula o movimento composto pelas rotinas *solve_u* e *solve_v*. Após o método que calcula a continuidade composto pela rotina *solve_p*. Por último, o cálculo da energia composto pela rotina *solve_z*. Se a compressibilidade artificial não tender a 0 é realizado cálculo para atualizar a pressão, caso contrário os resultados

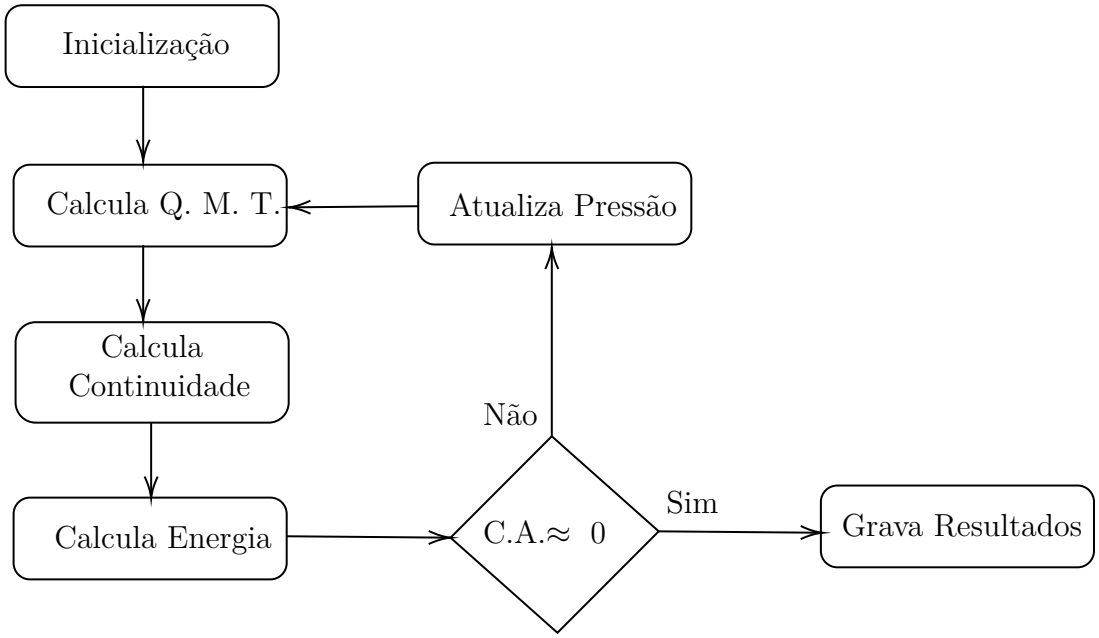
Figura 8 – Grade escalonada: os pontos azuis são a grade original, os pontos vermelhos e amarelos representam o local de avaliação dos componentes das velocidades u e v, respectivamente.



Fonte: Oliveira (2020)

são armazenados em arquivos para posterior processamento. A Figura 9 apresenta o fluxo de execução do algoritmo. O algoritmo 6 apresenta a estrutura original do algoritmo.

Figura 9 – Fluxo de execução do algoritmo



Fonte: Adaptado de (OLIVEIRA, 2022)

Algoritmo 6: Estrutura do Código original

```
...
!- -------- Leitura de dados de arquivos
CALL init

!- -------- Atribui valores as variaveis
CALL properties

DO WHILE (criterio parada)
    DO WHILE(criterio parada)
    !!-------- Calcula QUICK
        CALL solve_U(um,vm,um_n,um_tau,vm_tau,um_n_tau,pn,
           residual_u)
        CALL solve_V(um,vm,vm_n,um_tau,vm_tau,vm_n_tau,pn,T,
           InvFr2,residual_v)

    !!-------- Calcula Continuidade
        CALL solve_P(c2,p,um_n_tau,vm_n_tau,pn,residual_p)

    !-------- Calcula Energia
        CALL solve_Z(um_n_tau,vm_n_tau,T,T_n_tau,T_tau)

    !!-------- Verifica a convergencia
        CALL convergence(itc,c2,error,residual_p,residual_u,
           residual_v)

    !!--------- Avalia criterio de convergencia
        IF (itc .NE. 1 .AND. error.LT.eps) then
              EXIT
          ENDIF
    ENDDO

    !------ Grava dados em arquivos
         CALL output(um,vm,u,v,p,Z,T,H,itc)

ENDDO
```

4.6 Considerações Finais do Capítulo

O capítulo apresentou a formulação matemática da aplicação de secagem de grãos. Nele, destacam-se as fórmulas utilizadas para a modelagem do problema. No capítulo também é apresentado o fluxo de execução da aplicação que define o algoritmo.

5 ESTRATÉGIAS METODOLÓGICAS

Este capítulo descreve as estratégias definidas para alcançar os objetivos do trabalho. Nele é detalhado o paralelismo e os parâmetros adotados. Por fim, é descrito quais são os testes realizados e a forma com que serão medidos e avaliados os resultados.

5.1 Estratégia de Paralelismo

Este trabalho avalia o estudo de caso submetido a diferentes APIs de programação paralela. Foram escolhidas as APIs clássicas OpenMP para programação em CPU, OpenACC e OpenMP Target para programação com GPU. Para avaliar o desempenho foram estabelecidos os seguintes casos de teste:

- Caso de Teste 1.0 - Versão Sequencial
- Caso de Teste 2.0 - Versão com OpenMP
  - Caso de Teste 2.1 - Versão com OpenMP e 2 *threads*
  - Caso de Teste 2.2 - Versão com OpenMP e 4 *threads*
  - Caso de Teste 2.3 - Versão com OpenMP e 8 *threads*
  - Caso de Teste 2.4 - Versão com OpenMP e 16 *threads*
  - Caso de Teste 2.5 - Versão com OpenMP e 32 *threads*
- Caso de Teste 3.0 - Versão com OpenACC
- Caso de Teste 4.0 - Versão com OpenMP Target
  - Caso de Teste 4.1 - Versão com Target e 2 *teams*
  - Caso de Teste 4.2 - Versão com Target e 4 *teams*
  - Caso de Teste 4.3 - Versão com Target e 8 *teams*
  - Caso de Teste 4.4 - Versão com Target e 16 *teams*
  - Caso de Teste 4.5 - Versão com Target e 32 *teams*
- Caso de Teste 5.0 - Versão com OpenMP Teams
  - Caso de Teste 5.1 - Versão com Teams e 2 *teams*
  - Caso de Teste 5.2 - Versão com Teams e 4 *teams*
  - Caso de Teste 5.3 - Versão com Teams e 8 *teams*
  - Caso de Teste 5.4 - Versão com Teams e 16 *teams*
  - Caso de Teste 5.5 - Versão com Teams e 32 *teams*

Tabela 3 – Avaliação de Desempenho da Aplicação

Método (call)	**% de Tempo**	**Rotina**
funcoes_v	43,26	QMT
funcoes_u	40,69	QMT
funcoes_z	7,07	Energia
upwind_u	2,9	QMT
upwind_v	2,58	QMT
propriedade_v	0,91	QMT
propriedade_u	0,87	QMT
propriedade_p	0,65	Continuidade
funcoes_p	0,47	Continuidade
propriedade_z	00,40	Energia
atualiza_pressao	00,02	Pressão

Cada caso de teste é executado para cada tamanho de malha, conforme especificado:

- Malha I - 51x63
- Malha II - 100x124
- Malha III - 200x249

Para avaliar a aplicação foi utilizada a ferramenta *gprof* que permite mensurar as funções que mais demandam tempo na execução da aplicação. A Tabela 3 apresenta as funções, o percentual de tempo consumido por cada função na versão sequencial e a rotina do código em que a função pertence (rotina descritas conforme Figura 9). Dessa forma, é possível analisar os segmentos de código mais custosos.

Conforme apresentado na Tabela 3 a rotina mais custosa do código é a que calcula o método Quick (QMT) com aproximadamente 91% do processamento. A rotina que calcula a energia consome aproximadamente 7% do processamento. Enquanto que os métodos que calculam a continuidade e a pressão consomem pouco processamento, respectivamente 1,2% e 0,02%.

5.1.1 Paralelismo com OpenMP

Para paralelizar a aplicação com OpenMP foi utilizado o código original, ou seja, não foram feitas alterações. A diretiva aplicada foi `!$omp parallel do` sobre os laços de repetição das funções que não possuem dependência, começando com as que mais consomem tempo de processamento, e discriminando as variáveis privadas para cada laço. No algoritmo 7 é apresentado um exemplo de utilização da diretiva no código.

Também foi utilizada a diretiva `!$omp parallel sections` sobre o segmento de código que atualiza as condições de contorno das matrizes. Nesse segmento é paralelizado a rotina *atualiza_borda* apresentada na Figura 10. A rotina possui laços que iteram

Algoritmo 7: Exemplo do uso da diretiva `!$omp parallel do`

```
1 !$omp parallel do
2     DO i=2,imax-1
3         DO j=2,jmax-1
4             res_p(i,j) = dtau * RP(i,j) * c2
5             pn(i,j) = 1.0d0 / 3.0d0 * p(i,j) + 2.0d0 / 3.0d0 * (
                  pi(i,j) + res_p(i,j))
6         ENDDO
7     ENDDO
8 !$omp end parallel do
```

sobre matrizes distintas, sendo possível executá-los independentemente. No algoritmo 8 é apresentado um exemplo de utilização da diretiva no código.

Algoritmo 8: Exemplo do uso da diretiva `!$omp parallel sections`

```
1  !$omp parallel sections private (i, k)
2  !$omp section
3  DO i=1,imax
4          vm(i,2) = v_i
5          vm(i,1) = v_i
6  ENDDO
7  !$omp section
8  DO k=2,imax
9          um(k,1) = 0.d0
10 ENDDO
11 !$omp end parallel sections
```

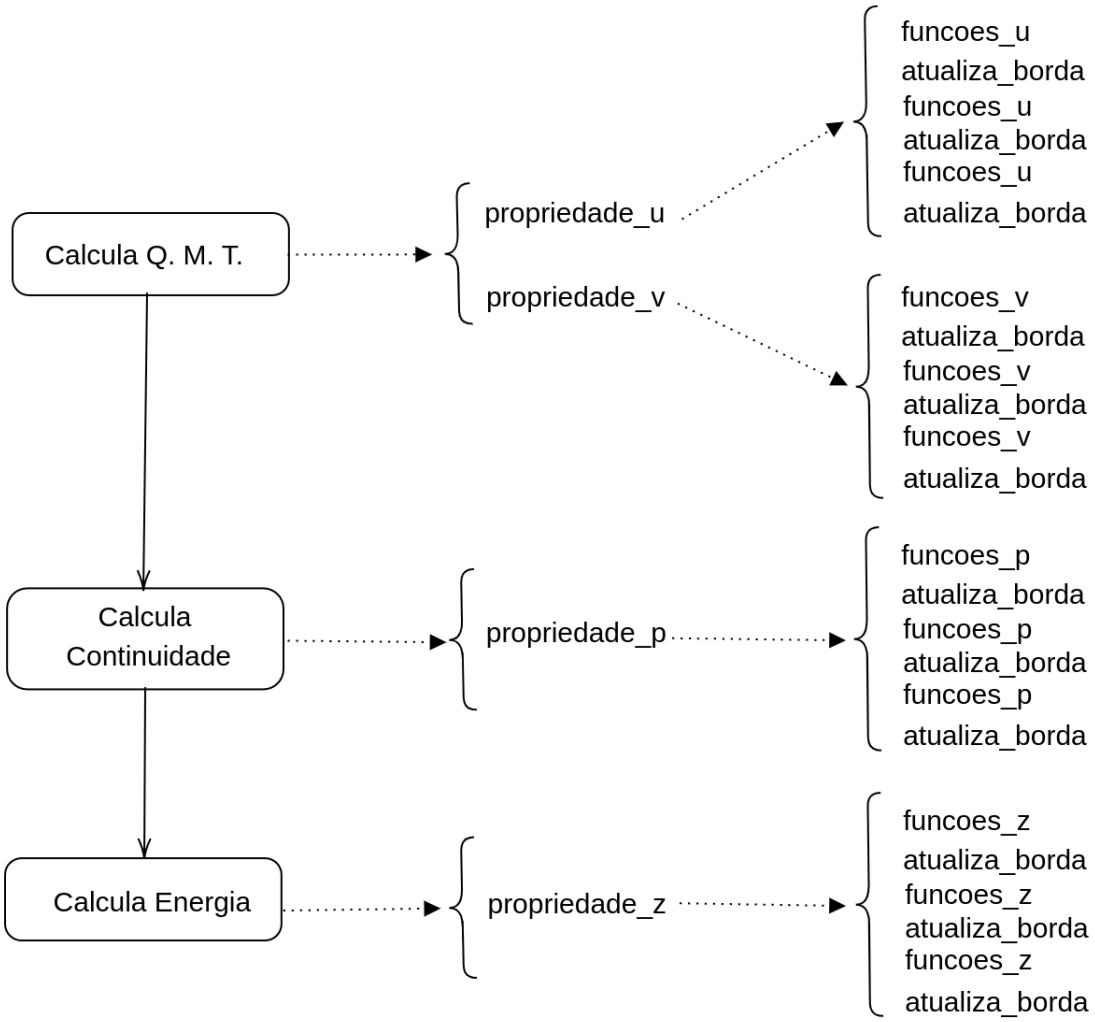
O número de *threads* é definido pela variável de ambiente `OMP_NUM_THREADS` de acordo com o caso de teste em execução.

5.1.2 Tratamento no código para executar na GPU

A estrutura original do algoritmo é apresentada na algoritmo 6. A proposta da versão com OpenACC é explorar os recursos disponíveis no *hardware*. Para explorar os recursos do dispositivo foi necessário alterar o código original. O código foi reestruturado de forma a permitir a comparação entre as versões e manter a compatibilidade numérica. Essas modificações são descritas de acordo com boas práticas de programação visando explorar os recursos disponíveis.

A estrutura original do código contém três rotinas principais. Estas rotinas são apresentadas na Tabela 3. Cada rotina possui uma chamada de função que repete 3 execuções dos métodos que calculam as funções ($funcao\_X$) e na sequência um método que atualiza o valor da borda da matriz principal ($atualiza\_borda$ - *boundary contorn*).

Figura 10 – Detalhamento das chamadas executadas pelo algoritmo



O algoritmo original utiliza variáveis globais para representar os parâmetros das propriedades físicas e das malhas. Algumas variáveis eram criadas no decorrer da computação. Essas declarações foram removidas e as variáveis foram declaradas todas na inicialização. O objetivo de remover essas variáveis é reduzir as sucessivas cópias de memória no decorrer da execução e realizar apenas uma cópia no início da execução.

Para facilitar o gerenciamento de variáveis simples, as variáveis que representavam inteiros ou decimais foram concatenadas em vetores. Cada vetor concatenou as variáveis respectivas a uma propriedade física. Após essas modificações as chamadas de função foram ajustadas. Foi acrescentado o vetor das respectivas variáveis simples e as matrizes que passaram a ser criadas na inicialização.

No algoritmo original cada propriedade física possui duas subestruturas, uma responsável pela computação (cálculos em geral) e outra responsável pela atualização das bordas. Na algoritmo 9 a rotina *funcao_u* representa uma propriedade física e *calc_u* representa a computação. A rotina *calc_u* possui um ou mais laços e uma expressão matemática.

Algoritmo 9: Exemplo do código original

```
1  subroutine calc_u(...)
2
3      DO i=1,10
4          DO j=1,10
5             a = (b * c) + a + 1.d+4
6          ENDDO
7      ENDDO
8
9  RETURN
10 END SUBROUTINE calc_u
11
12 subroutine funcao_u(...)
13
14    CALL calc_u(a, b, c)
15
16 RETURN
17 END SUBROUTINE funcao_u
```

A alteração feita nesse segmento do código é remover os laços internos das chamadas de função para o método principal, no caso o método *funcao_u*. Essa alteração é expressa no algoritmo 10.

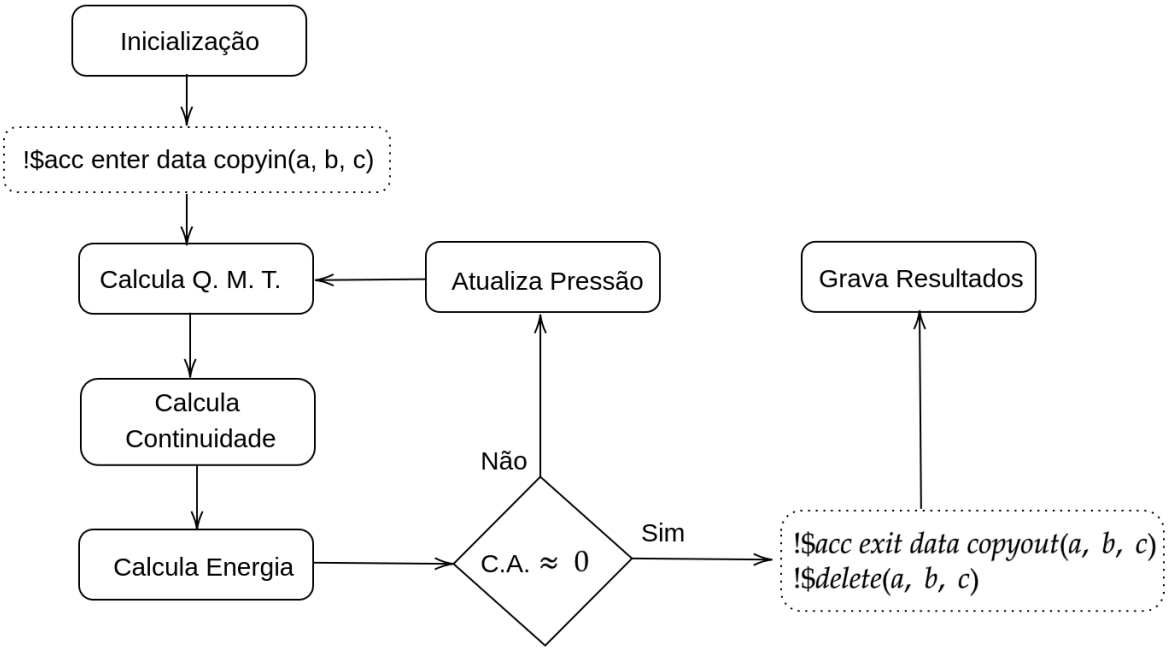
Algoritmo 10: Exemplo do código modificado

```
1  subroutine calc_u(...)
2     ...
3      a = (b * c) + a + 1.d+4
4     ...
5  RETURN
6  END SUBROUTINE calc_u
7
8  subroutine funcao_u(...)
9      DO i=1,10
10         DO j=1,10
11            CALL calc_u(a, b, c)
12         ENDDO
13     ENDDO
14 RETURN
15 END SUBROUTINE funcao_u
```

5.1.3 Paralelismo com OpenACC

A arquitetura SIMD (do inglês *Single Instruction Multiple Data*) presente nos *Streaming Multiprocessor* (SM) dos processadores da GPU possuem alta capacidade de

Figura 11 – Fluxo de manipulação das variáveis entre *host* e *device*



processamento. Essa capacidade ocorre devido ao número de unidades aritméticas disponíveis, possibilitando computar um grande volume de dados.

Um ponto muito importante na versão implementada em OpenACC é a transferência de memória entre os dispositivos. A rotina QMT consome a maior porcentagem de processamento, paralelizar apenas essa rotina implica em sucessivas transferências de informações ou uso da rotina *update* que atualiza as informações no respectivo dispositivo. Dado o volume de dados que são manipulados é inviável realizar sucessivas transferências de dados entre *host* e *device*. A fim de reduzir as transferências de memória, as rotinas que calculam o QMT, continuidade, energia e pressão foram executadas somente na GPU. A cópia é realizada no início da execução pela diretiva *!$acc enter data copyin*. Os dados são mantidos na GPU durante a execução e são descarregados para a CPU para a finalização, quando os dados são armazenados em arquivos. O fluxo de transferências entre os dispositivos é descrito na Figura 11. O código fonte original foi modificado como descrito na subseção 5.1.2 a estrutura original do código é apresentada no algoritmo 6. Essas alterações foram necessárias para manter a compatibilidade numérica entre as versões.

Em cada função paralela é definida a diretiva *!$acc data present* com as variáveis utilizadas. O objetivo desta diretiva é explicitar ao compilador que as variáveis estão presentes na memória da GPU. Os laços de repetição são paralelizados através da diretiva *!$acc parallel loop collapse (2)*, pois os laços são aninhados. O algoritmo 12 apresenta o mesmo segmento de código do algoritmo 7 entretanto paralelizado com diretivas OpenACC.

Algoritmo 11: Estrutura original

```
1       call inicializacao()
2       DO WHILE (criterioParada)
3          CALL calcula_QMT(a, b, c)
4          CALL calcula_continuidade(a, b, c)
5          CALL calcula_energia(a, b, c)
6          CALL calcula_pressao(a, b, c)
7       ENDDO
8       call resultados()
```

Algoritmo 12: Exemplo do uso da diretiva `!$acc parallel loop`

```
1  !$acc parallel loop collapse(2)
2      DO i=2,imax-1
3          DO j=2,jmax-1
4              res_p(i,j) = dtau * RP(i,j) * c2
5              pn(i,j) = 1.0d0 / 3.0d0 * p(i,j) + 2.0d0 / 3.0d0 * (
                   pi(i,j) + res_p(i,j))
6          ENDDO
7      ENDDO
8  !$acc end parallel
```

5.1.4 Paralelismo com OpenMP Target

A estrutura de paralelismo descrita pelo OpenMP Target é semelhante ao OpenACC, como apresentado na Tabela 1. A estratégia de paralelismo adotada é copiar os dados para GPU, manipular os dados, ou seja, realizar as computações e somente ao final da execução retornar os dados para a CPU. Assim, como foi feito na versão com OpenACC. Essa estratégia é utilizada para minimizar as cópias de memória entre *host* e *device*.

O algoritmo 13 apresenta a diretiva *target map*. O objetivo desta diretiva é reduzir as cópias de memória quando as variáveis são utilizadas por vários segmentos de código seja em laços de repetição ou métodos(*call* em Fortan). O *map(to: variaveis)* especifica as variáveis a serem mapeadas explicitamente das variáveis originais no ambiente de dados do *host* para as variáveis correspondentes no ambiente de dados do *device*.

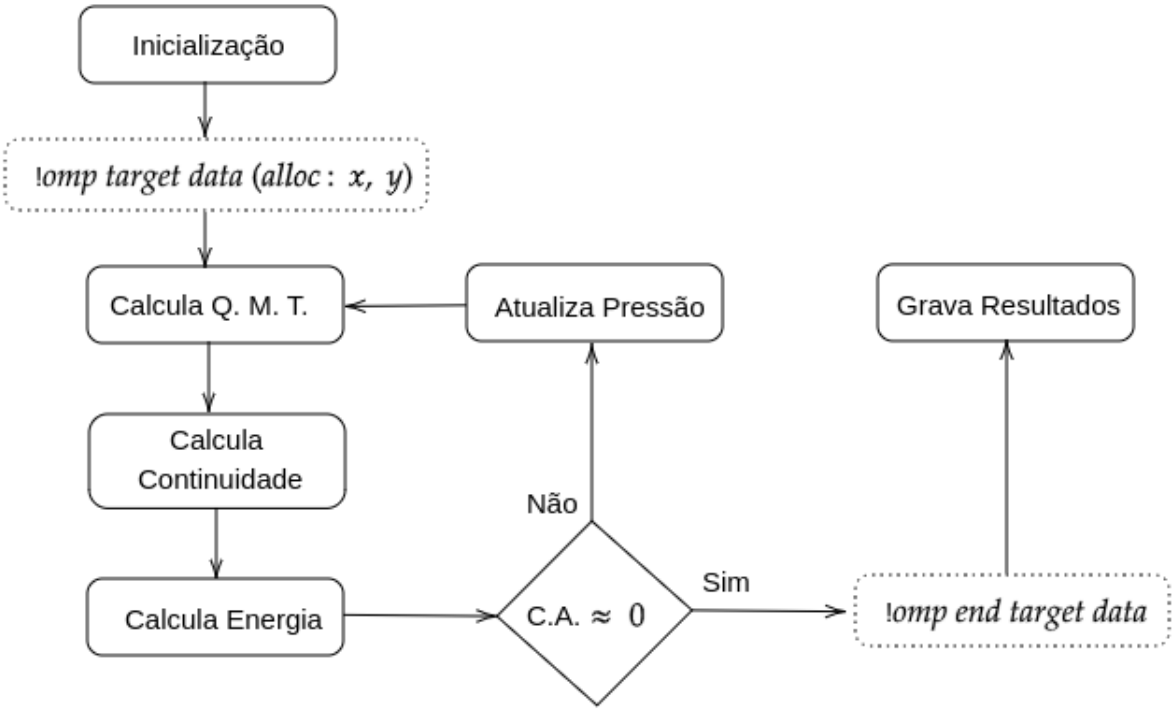
Como pode ser observado na Figura 12, a fase da inicialização é executada na CPU e após essa etapa as variáveis são copiadas para a GPU. A computação presente nos métodos de cálculo do Q.M.T., continuidade, energia e eventualmente pressão são todos feitos na GPU. Ao finalizar toda a computação as variáveis são copiadas para a CPU. Na etapa final os valores já atualizados são salvos em arquivos para pós-processamentos - geração de gráficos.

A diretiva *target* especifica que o segmento de código será executado em um *device*. É possível combinar outras diretivas para gerar paralelismo. A diretiva *teams* cria times ou

Algoritmo 13: Exemplo do uso da diretiva `!$omp target map`

```
1  CALL inicializacao()
2  DO WHILE (criterioParada)
3          CALL calcula_QMT(a, b, c)
4          CALL calcula_continuidade(a, b, c)
5          CALL calcula_energia(a, b, c)
6          CALL calcula_pressao(a, b, c)
7  ENDDO
8  CALL gravaResultados()
```

Figura 12 – Fluxo de manipulação das variáveis entre *host* e *device*



equipes de threads. É possível especificar o número de times que podem ser instanciados através da cláusula *num_teams* e também o número máximo de threads *thread_limit*. Nos testes realizados o número de times(num_teams) é fixado de acordo com o número de *threads*, ou seja, se o caso de teste possui 2 threads o número de times é igual a 2. A algoritmo 14 descreve o uso da diretiva *teams* definindo as cláusulas de número de equipes e número máximo de threads.

As equipes de threads exploram o recurso *Streaming Multiprocessor* disponível em GPUs. Para explorar ainda mais o paralelismo é utilizada a diretiva *distribute*. Essa diretiva distribui as iterações de um laço entre as threads mestres. A diretiva impõem uma restrição, sendo exigido uma construção *do-loop* na sequência da diretiva. Também é possível combinar com a diretiva *parallel for*. Assim como no OpenACC a cláusula *collapse(n)* pode ser especificada para tratar *loops* aninhados. A algoritmo 15 apresenta o uso da diretiva *teams distribute parallel for*.

Algoritmo 14: Exemplo do uso da diretiva `!$omp teams`

```
1 !$omp teams num_teams(numThreads) thread_limit(32)
2     DO i=1,10
3             CALL funcoes_u(a, b, c)
4     ENDDO
5 !$omp end teams
```

Algoritmo 15: Exemplo do uso da diretiva `!$omp teams distribute`

```
1  !$omp teams num_teams(numThreads) thread_limit(32)
2      !$omp distribute parallel do collapse(2) private(i, j, var)
3      DO i=1,10
4          DO j=1,10
5             CALL funcoes_u(a, b, c)
6          ENDDO
7      ENDDO
8
9      !$omp distribute parallel do private(i, j, var)
10     DO i=1,10
11         CALL atualiza_borda(a, b, c)
12     ENDDO
13 !$omp end teams
```

Todas as diretivas apresentadas são da documentação/especificação OpenMP 5.0 (OPENMP, 2018). Nessa versão é dispensado declarar a diretiva *!omp declare target* para que fosse possível paralelizar uma chamada de função (*call* em FORTRAN).

Como apresentado na Figura 10 cada propriedade física executada calcula funções que são responsáveis pela computação do método correspondente (propriedade física). De modo geral, cada propriedade física instancia uma equipe de n thread - a quantidade é especificada de acordo com o teste sendo executado. A etapa *funcoes_u* corresponde a laços de repetição com apenas chamadas de função. Esses laços de repetição são distribuídos entre as n threads como apresentado na algoritmo 16.

Na sequência é executado o método *atualiza_borda*, esse método possui laços internos que atualizam os valores da borda da matriz informada. Esse método possui laços que não foram paralelizados devido a baixa granularidade. De acordo com os testes realizados, o tempo de execução com diretivas *$omp distribute parallel do* foi 35% maior que o tempo da execução que não utiliza paralelismo no método. Como a diretiva *distribute* agrega o tempo para distribuir as iterações entre as threads, testamos o tempo de execução com a diretiva *$omp parallel do simd*, mas o tempo de execução foi 33% maior.

Algoritmo 16: Exemplo do paralelismo aplicado em cada propriedade física

```
1
2  subroutine propriedade_u(...)
3  !$omp teams num_teams(numThreads) thread_limit(32)
4      !$omp distribute parallel do collapse(2) private(a, b, c)
5      DO i=1,10
6          DO j=1,10
7             CALL funcoes_u(a, b, c)
8          ENDDO
9      ENDDO
10
11     !$omp parallel do simd private(a, b, c)
12     DO i=3,imax-1
13             res_u(i,j) =( (a - b)) +  b * dt) * 8.d-2
14     ENDDO
15
16     CALL atualiza_borda(a, b, c)
17
18 !$omp end teams
19
20 RETURN
21 END SUBROUTINE propriedade_u
```

5.2 Parâmetros de Entrada

Os parâmetros de entrada são definidos nos arquivos de entrada. Para exemplificar, alguns parâmetros são: número de iterações, taxa de erro, critérios de convergência, coeficiente de dissipação e critério de convergência de massa. Os parâmetros numéricos e físicos foram mantidos com exceção do número de iterações e tamanho da malha. O número de iterações foi fixado em 20.000 (vinte mil). Os tamanhos das malhas foram escolhidos para representar adequadamente o problema e o número de iterações definido para que o processo de secagem do grão fique completo.

5.3 Compatibilidade Numérica

A compatibilidade numérica é um parâmetro fundamental. O objetivo do trabalho é comparar os resultados obtidos nas versões desenvolvidas com o resultado obtido pela versão original. Quando utilizamos outros dispositivos como uma ou mais GPUs é comum que os resultados sejam aproximados.

A avaliação da compatibilidade numérica foi feita através de um script. O script foi desenvolvido em shell script e na linguagem de programação C++. São comparados os valores das matrizes que são salvas ao final da execução. Como os valores da saída são números de ponto flutuante que podem conter erros de precisão devido a arredondamentos internos, a comparação é realizada pelo método descrito no algoritmo 17. O valor 1e-5

corresponde ao ϵ.

Algoritmo 17: Comparação de números ponto flutuante

```
1 bool isEqual(double x, double y) {
2     return abs(x - y) <= 1e-5 * abs(x);
3 }
```

5.4 Métricas de Avaliação

Para uma melhor amostragem do tempo de execução da aplicação, seja sequencial ou paralela, foram realizadas 12 execuções para cada uma das malhas. O número de execuções foi estabelecido Desses resultados, o menor e o maior tempo de execução são descartados. A partir das 10 execuções restantes é mensurado o tempo médio de execução e o desvio padrão da amostra. O comando *time*, nativo do *command line interface*, foi utilizado para obter o tempo de execução total da aplicação.

Para medir o desempenho das versões dos algoritmos paralelos foi utilizado o conceito de *speedup*(S). O *speedup* é definido como a razão entre o tempo de computação do algoritmo serial (T_{serial}) e o tempo de computação do algoritmo paralelo ($T_{paralelo}$), dado pela Equação 1. O *speedup* mostra o ganho efetivo do tempo de processamento do algoritmo paralelo sobre o algoritmo serial (DONGARRA et al., 2003).

$$S = \frac{T_{serial}}{T_{paralelo}} \tag{1}$$

Uma outra forma de mensurar o quanto uma versão paralela é melhor que a versão sequencial é considerar o percentual de ganho de desempenho (aceleração) apresentado na Equação 2.

$$A = \frac{T_{serial} - T_{paralelo}}{T_{paralelo}} * 100 = (\frac{T_{serial}}{T_{paralelo}} - 1) * 100 \tag{2}$$

Para determinar se há uma diferença significativa entre as médias de cada um dos testes foi aplicada a distribuição *t de Student*. A distribuição *t student* é uma distribuição de probabilidade estatística para avaliar o nível de significância das amostras. A fim de constatar se os resultados obtidos são significativamente diferentes, ou seja, houve ganho ou perda no tempo de desempenho. A hipótese avaliada é: o tempo médio de execução entre a versão sequencial e a versão de teste é igual. Consideramos um nível de significância de 0,01, ou seja 99%. Para calcular o P-valor foi utilizado um software de planilha eletrônica[1].

[1] docs.google.com/spreadsheets

Score-p[2] é uma ferramenta de medição de desempenho para códigos paralelos (KNÜPFER et al., 2012). Foi desenvolvido pelo *Scalable Infrastructure for the Automated Performance Analysis of Parallel Codes* (SILC). O objetivo da ferramenta é simplificar a análise do código, permitindo que gargalos de desempenho possam ser identificados a fim de melhorar o desempenho da aplicação. Essa ferramenta salva em arquivos os parâmetros selecionados. Para interpretar esses dados foi utilizada a ferramenta Vampir[3]. Essa ferramenta permite a visualização de desempenho, ou rastreamento da execução. Foi utilizado o formato *otf2*. A versão Demo do Vampir é limitada a 16 threads.

5.5 Ambiente de Execução

Os testes desta dissertação foram executados nos ambientes descritos nas Tabelas 4 e 5. O compilador *pgf90* foi utilizado com a *flag* -O3 para a execução sequencial. Para as execuções do código com OpenMP além da *flag* -O3 foi utilizada a *tag* -fopenmp. Já para as execuções do código com OpenACC foram utilizadas as *tags* -fast, -acc, e Minfo=all.

Tabela 4 – Ambiente de execução: CPU

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

Tabela 5 – Ambiente de execução: GPU

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

[2] https://www.vi-hps.org/projects/score-p/
[3] https://vampir.eu/

6 ANÁLISE EXPERIMENTAL

Esse capítulo tem por objetivo expor e analisar os resultados obtidos pelos testes realizados. Os resultados são apresentados em forma tabular e gráfica, sendo respectivamente comentados.

6.1 Resultados das Execuções

A Tabela 6 apresenta os resultados obtidos sobre as malhas I, II e III, respectivamente 51x63, 100x124 e 200x249. A tabela descreve a malha, o caso de teste, o tempo médio das execuções, o *speedup*, a taxa de aceleração e o desvio padrão. Em destaque consta o teste com maior *speedup*. As versões estão disponíveis [1].

Tabela 6 – Resultados Obtidos nos Testes

Malha	Caso de Teste	Tempo(s)	S	A(%)	Desv. Pad.
51 x 63	1.0 Sequencial	302,155	-	-	0,267
	2.1 OpenMP 2 Threads	175,000	1,73	72,66	6,290
	2.2 OpenMP 4 Threads	168,500	1,79	79,32	6,443
	2.3 OpenMP 8 Threads	150,500	2,01	100,77	6,111
	2.4 OpenMP 16 Threads	117,000	2,58	158,25	1,829
	2.5 OpenMP 32 Threads	109,978	2,75	174,74	1,149
	3.0 OpenACC	259,974	1,16	16,23	0,349
	4.1 Target 2 Threads	170,942	1,77	76,76	0,163
	4.2 Target 4 Threads	168,959	1,79	78,83	1,892
	4.3 Target 8 Threads	149,116	2,03	102,63	0,551
	4.4 Target 16 Threads	112,153	2,69	169,41	1,118
	4.5 Target 32 Threads	**97,095**	**3,11**	**211,20**	**1,033**
	5.1 Teams 2 Threads	175,330	1,72	72,34	1,938
	5.2 Teams 4 Threads	165,100	1,83	83,01	0,555
	5.3 Teams 8 Threads	155,215	1,95	94,67	1,061
	5.4 Teams 16 Threads	112,760	2,68	55,20	3,752
	5.5 Teams 32 Threads	110,755	2,73	52,14	13,094
100 x 124	1.0 Sequencial	1223,657	-	-	0,952
	2.1 OpenMP 2 Threads	656,465	1,86	86,40	2,816
	2.2 OpenMP 4 Threads	420,165	2,91	191,23	2,086
	2.3 OpenMP 8 Threads	273,565	4,47	347,30	4,907
	2.4 OpenMP 16 Threads	221,190	5,53	453,22	2,526
	2.5 OpenMP 32 Threads	206,670	5,92	492,08	1,660
	3.0 OpenACC	348,535	3,51	251,09	0,488

[1] https://github.com/NatiLucca/POROUS-MEDIA-APPLICATION

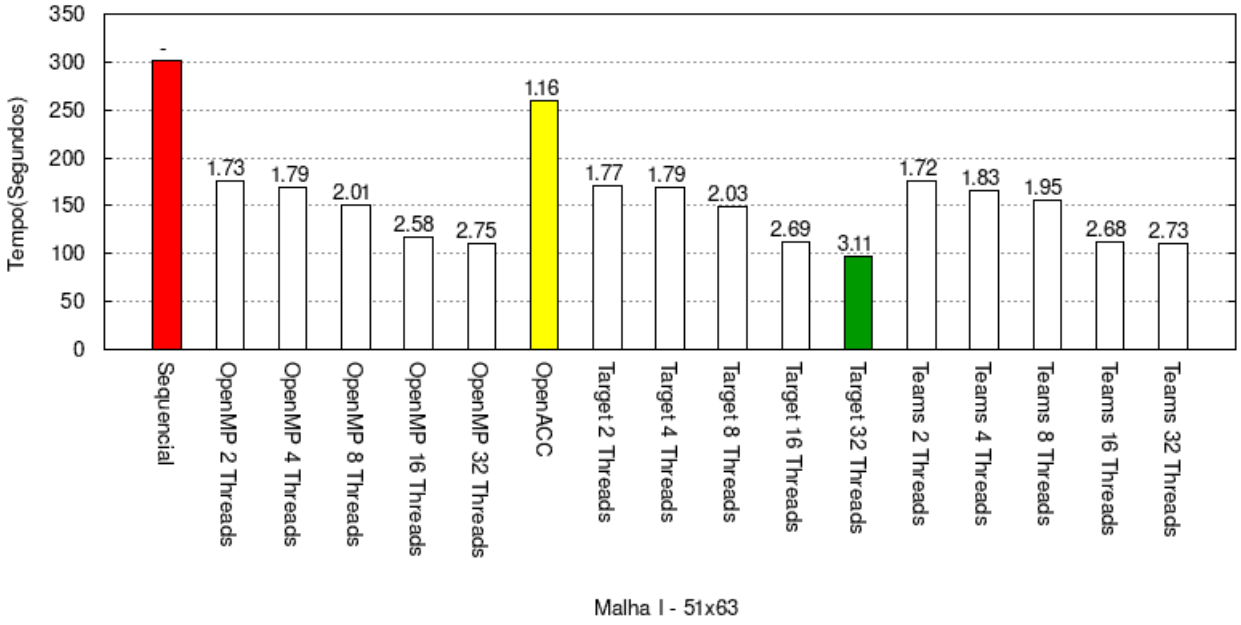
	4.1 Target 2 Threads	661,837	1,85	84,89	0,364
	4.2 Target 4 Threads	435,818	2,81	180,77	1,534
	4.3 Target 8 Threads	300,615	4,07	307,05	5,919
	4.4 Target 16 Threads	271,756	4,50	350,28	3,055
	4.5 Target 32 Threads	252,185	4,85	385,22	1,193
	5.1 Teams 2 Threads	658,710	1,86	85,77	2,948
	5.2 Teams 4 Threads	428,195	2,86	185,77	1,132
	5.3 Teams 8 Threads	277,785	4,41	340,50	3,049
	5.4 Teams 16 Threads	**195,260**	**6,27**	**526,68**	**9,868**
	5.5 Teams 32 Threads	209,590	5,84	483,83	2,906
200 x 249	1.0 Sequencial	4827,651	-	-	7,485
	2.1 OpenMP 2 Threads	2612,000	1,85	84,83	7,885
	2.1 OpenMP 4 Threads	1620,500	2,98	197,91	24,212
	2.1 OpenMP 8 Threads	986,000	4,90	389,62	14,583
	2.1 OpenMP 16 Threads	860,000	5,61	461,35	27,524
	2.1 OpenMP 32 Threads	1151,745	4,19	319,16	20,106
	3.0 OpenACC	**640,198**	**7,54**	**654,09**	**1,014**
	4.1 Target 2 Threads	2757,185	1,75	75,09	11,644
	4.2 Target 4 Threads	1708,541	2,83	182,56	14,947
	4.3 Target 8 Threads	1101,158	4,38	338,42	2,811
	4.4 Target 16 Threads	809,640	5,96	496,27	3,991
	4.5 Target 32 Threads	846,265	5,70	470,47	4,879
	5.1 Teams 2 Threads	2545,995	1,90	89,62	12,919
	5.2 Teams 4 Threads	1621,320	2,98	197,76	9,755
	5.3 Teams 8 Threads	985,045	4,90	390,09	6,751
	5.4 Teams 16 Threads	643,345	7,50	650,40	13,861
	5.5 Teams 32 Threads	889,170	5,43	442,94	26,112

Fonte: Autora

6.2 Discussão dos Resultados

Todos os casos de teste submetidos a Malha I - 51x63 obtiveram ganho de desempenho. O tempo médio de execução da versão sequencial foi 302,15 segundos. O *speedup* variou de 1,16 há 3,11, no pior e melhor resultado respectivamente. O melhor resultado foi o caso de teste 4.5 versão com uso do OpenMP Target e 32 Threads. Esse resultado obteve uma taxa de aceleração de 211,20%, com um desvio padrão de 1,03. O caso de teste 3.0, versão com uso de OpenACC obteve o pior resultado dos testes com uma ace-

Figura 13 – Tempo de Execução - Resultado da malha 51x63



leração de apenas 16.23%. A Figura 13 apresenta os tempos de execução de cada caso de teste da Malha I - 51x63. Sobre cada coluna é descrito o *speedup*. Em destaque, a coluna verde representa o caso de teste com o menor tempo de execução, a coluna vermelha o caso de teste com o maior tempo de execução e em amarelo o caso de teste com o pior *speedup*(desconsiderando o caso de teste sequencial).

Assim como a Malha I, a Malha II - 100x124 também obteve ganho de desempenho em todos os casos de teste realizados. O tempo médio de execução da versão sequencial foi 1223,66 segundos. O *speedup* variou de 1,85 há 6,27, no pior e melhor resultado, respectivamente. O melhor resultado foi o caso de teste 5.4, versão OpenMP Teams com 16 threads. Esse resultado obteve uma taxa de aceleração de 526,68% e desvio padrão de 9,87. O caso de teste 4.1 OpenMP Target com 2 threads obteve o pior resultado. É notável que os testes com uso de 2 threads obtiveram os piores resultados da malha. Os resultados da malha II são apresentados na Figura 14.

Todos os testes realizados na Malha III - 200x249 obtiveram ganho de desempenho. O tempo médio de execução da versão sequencial foi 4827,65 segundos. O *speedup* variou de 1,75 há 7,54, no pior e melhor resultado respectivamente. O melhor resultado foi o caso de teste 3.0, versão OpenACC. Esse resultado obteve uma taxa de aceleração de 654,09% e desvio padrão de 1,02. O caso de teste 4.1 OpenMP Target com 2 threads obteve o pior resultados com uma taxa de aceleração de 75,09%. Os resultados da malha III são apresentados na Figura 15.

Figura 14 – Tempo de Execução - Resultado da malha 100x124

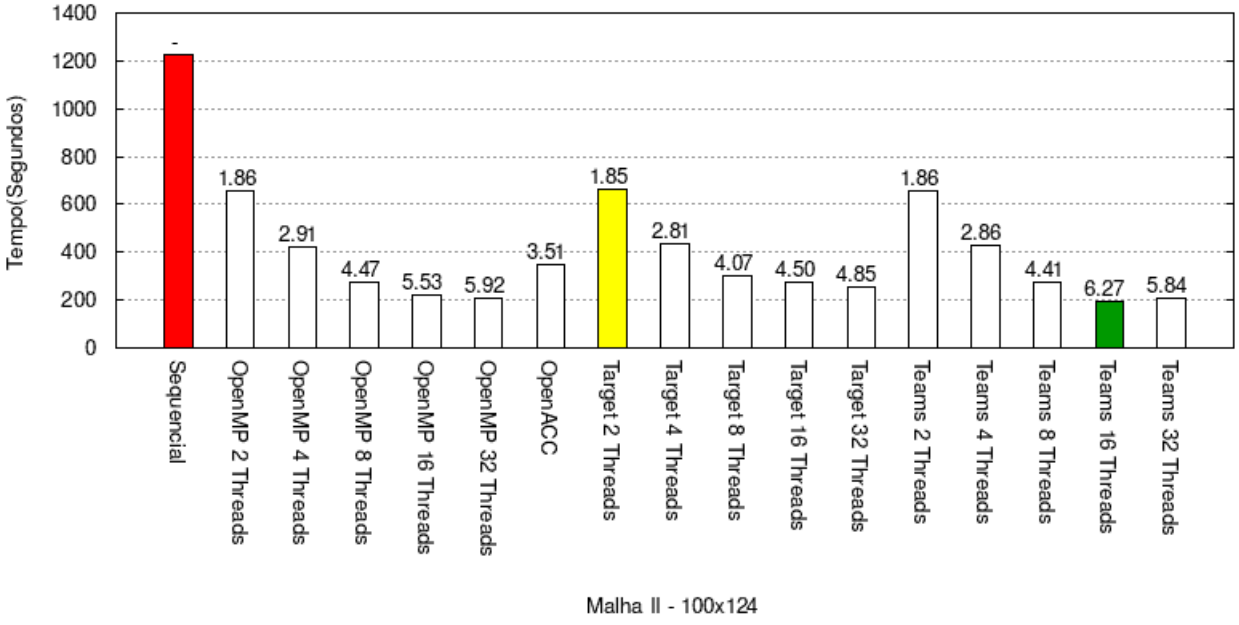
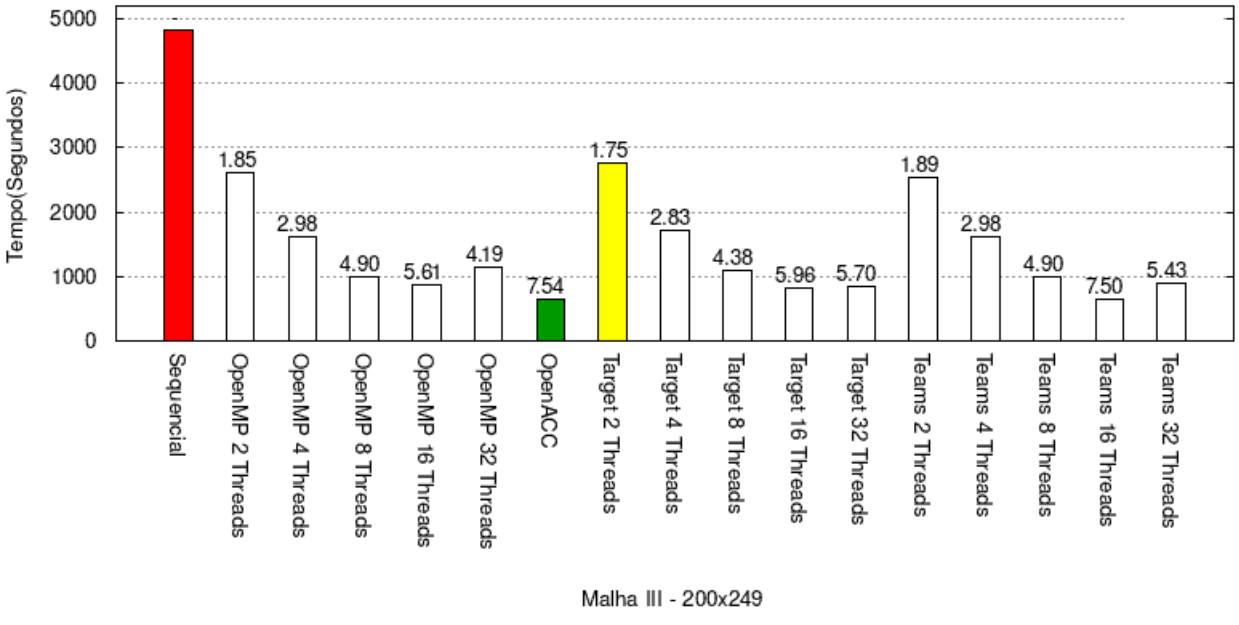


Figura 15 – Tempo de Execução - Resultado da malha 200x249



6.3 Validação Numérica das Soluções

Como apresentado na Figura 9 a última etapa do código consiste em gravar os resultados obtidos. A aplicação armazena diversos arquivos com as informações das matrizes. Esses arquivos podem ser utilizados para gerar gráficos e/ou imagens que ilustram o comportamento do fluido durante o processo de secagem. A validação numérica é importante para garantir que os resultados se mantenham consistentes após as alterações de código.

Como comparamos números decimais oriundos de hardwares distintos, podem haver diferenças geradas por arredondamentos nos cálculos. Para avaliar os arquivos de saída foi desenvolvido um script em *shell* descrito na seção 5.3. O script compara o arquivo de saída da versão sequencial com o do caso de teste em questão a fim de identificar erros numéricos. Para todos os resultados apresentados houve compatibilidade numérica dentro da tolerância estabelecida.

6.4 Distribuição *t de Student*

Para avaliar se os resultados paralelos obtidos são significativamente diferentes dos resultados da versão original sequencial foi aplicada a análise estatística *t de student*. Os resultados obtidos são apresentados na Tabela 7. A tabela descreve a caso de teste, o valor do *teste F* e o *P-valor*.

Para calcular o p-valor foram especificadas as amostras, o tipo de distribuição e o tipo de teste. A distribuição utilizada foi a bicaudal. Para o tipo de teste foi calculado o *teste F*. Esse teste identifica se a variação de duas amostras é igual (homoscedástica) ou desigual (heteroscedástica). Os resultados estão disponíveis[2].

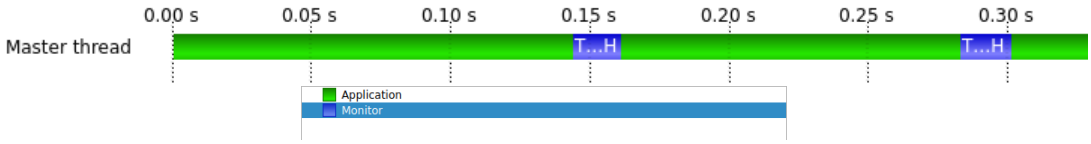
Tabela 7 – Resultados da Distribuição T de Student

Malha	Caso de Teste	Teste F	P-valor
51 x 63	OpenMP 2 Threads	5,78E-11	2,09E-13
	OpenMP 4 Threads	9,00E-16	2,11E-08
	OpenMP 8 Threads	7,49E-11	4,08E-14
	OpenMP 16 Threads	3,38E-06	3,23E-20
	OpenMP 32 Threads	1,76E-04	2,20E-23
	OpenACC	4,39E-01	8,10E-35
	Target 2 Threads	1,47E-01	2,09E-46
	Target 4 Threads	2,51E-06	1,09E-18
	Target 8 Threads	7,65E-02	1,02E-40

[2] https://docs.google.com/spreadsheets/d/155W2o2MdzIkca8zunMIKI5hQ6bi_KPJPkfxRyjnd7IQ/edit?usp=sharing

	Target 16 Threads	2,20E-04	1,47E-23
	Target 32 Threads	4,18E-04	1,42E-24
	Teams 2 Threads	3,98E-02	5,54E-41
	Teams 4 Threads	3,39E-04	1,26E-22
	Teams 8 Threads	2,72E-08	9,96E-17
	Teams 16 Threads	5,91E-09	5,83E-17
	Teams 32 Threads	7,97E-14	3,83E-12
100 x 124	OpenMP 2 Threads	3,46E-03	2,76E-26
	OpenMP 4 Threads	2,86E-02	5,96E-45
	OpenMP 8 Threads	3,93E-05	1,94E-23
	OpenMP 16 Threads	7,69E-03	1,41E-30
	OpenMP 32 Threads	1,13E-01	3,23E-48
	OpenACC	6,00E-02	1,38E-51
	Target 2 Threads	7,91E-06	3,76E-22
	Target 4 Threads	1,86E-03	9,94E-28
	Target 8 Threads	5,12E-01	1,25E-49
	Target 16 Threads	1,86E-03	9,94E-28
	Target 32 Threads	5,12E-01	1,25E-49
	Teams 2 Threads	2,45E-03	1,05E-25
	Teams 4 Threads	6,14E-01	2,61E-48
	Teams 8 Threads	1,89E-03	1,02E-27
	Teams 16 Threads	8,96E-08	5,82E-20
	Teams 32 Threads	2,73E-03	1,21E-28
200 x 249	OpenMP 2 Threads	8,79E-01	1,25E-40
	OpenMP 4 Threads	1,08E-23	5,31E-37
	OpenMP 8 Threads	1,71E-02	1,08E-23
	OpenMP 16 Threads	6,38E-04	2,12E-23
	OpenMP 32 Threads	7,04E-03	1,31E-26
	OpenACC	1,78E-06	4,61E-27
	Target 2 Threads	2,04E-01	2,75E-38
	Target 4 Threads	5,15E-02	4,74E-40
	Target 8 Threads	7,48E-03	1,13E-31
	Target 16 Threads	7,49E-02	2,54E-47
	Target 32 Threads	2,18E-01	7,60E-47
	Teams 2 Threads	1,20E-01	1,82E-38
	Teams 4 Threads	2,72E-40	2,72E-40
	Teams 8 Threads	7,64E-01	1,18E-42
	Teams 16 Threads	8,06E-02	8,36E-43
	Teams 32 Threads	9,69E-04	7,19E-24

Figura 16 – Execução sequencial malha 100x124



Fonte: Autora

Como especificado na metodologia, o nível de significância utilizado é 99%, ou seja, para que as amostras sejam significativamente diferentes o *P-valor* deve ser menor que 0,01. Em todos os casos o resultado do P-valor foi inferior a 0,01.

6.5 Análise com Score-P e Vampir

Score-P é uma ferramenta robusta para análise de desempenho (SCORE-P, 2022). Essa ferramenta possui integração com compiladores, como por exemplo, gcc, intel e pgi. Também possui integração com paradigmas de programação como MPI, OpenMP, PThreads, CUDA, OpenCL e OpenACC. Entretanto, não foi possível instalar a ferramenta utilizando o compilador pgi e as APIS OpenMP e OpenACC. A instalação foi impedida por erros entre bibliotecas. Foi possível contornar e instalar apenas a versão com suporte a OpenMP. O Vampir é um visualizador de desempenho de software focado em aplicativos altamente paralelos. Ele apresenta uma visão unificada da execução.

A visualização da versão sequencial é uma execução constante descrita na Figura 16. Em verde é apresentado o processamento da aplicação e em azul a ferramenta de monitoramento Score-P.

A visualização das execuções com OpenMP foram geradas com sucesso. A Figura 18 apresenta todas as versões com OpenMP com 2, 4, 8 e 16 threads. Não foi possível visualizar os testes com 32 threads devido a limitação da ferramenta Vampir.

A Figura 17 apresenta 1 ciclo de processamento para uma execução com 16 threads. Esse ciclo consiste na execução das funções que calculam QMT, continuidade, energia e eventualmente pressão (Figura 10). A primeira parte da execução correspondente ao cálculo do QMT, ou seja, propriedade física u e propriedade física v. Essa parte do código compreende 12 *loops* para cada propriedade física, ao todo são 24 laços de repetição nessa rotina. Como destacado na Tabela 3 a rotina QMT compreende 91% do processamento. A segunda parte do ciclo corresponde ao cálculo da continuidade ou propriedade p. Essa propriedade física compreende 6 *loops* e corresponde a 1.2% do processamento. A última parte do ciclo corresponde ao cálculo da energia, ou seja, a propriedade física z que corresponde a 7% do processamento.

A Figura 18 apresenta a malha 51x63 para as execuções com 2, 4, 8 e 16 threads.

Figura 17 – Exemplo - Casos de Teste OpenMP

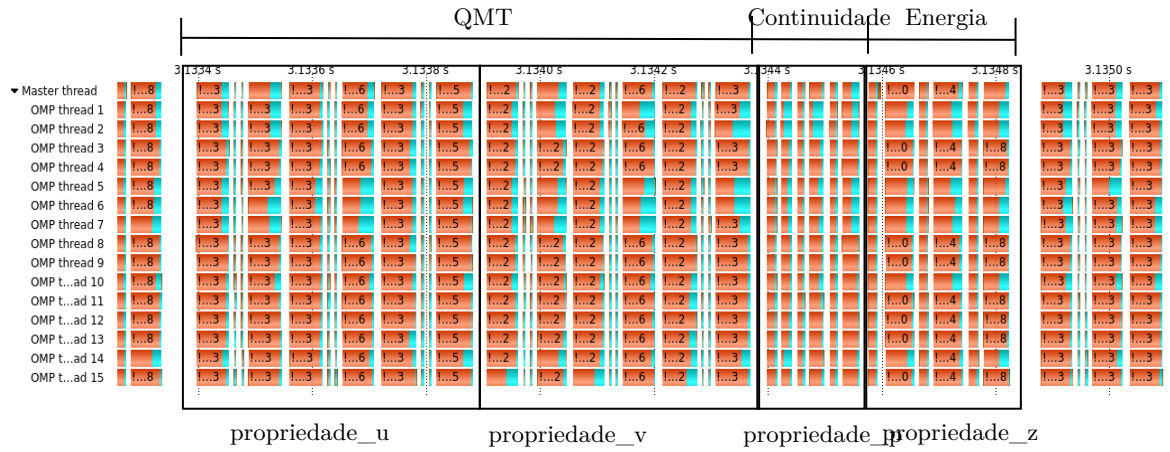


Figura 18 – Casos de Teste OpenMP - Malha 51x63

Em vermelho constam os blocos de código que compreendem instruções *omp*, em azul as etapas do processamento onde são feitas sincronizações de dados e os espaços vazios são os demais segmentos do código que não são monitorados pelo *profiling*. Os segmentos do código que não são mensurados pelo *profiling* são especificados no arquivo na compilação. Nas execuções para a malha 51x63 é observável que as sincronizações consomem parte do tempo de processamento e para alguns casos é quase equivalente ao tempo de processamento do laço de repetição. Os resultados numéricos apontam que a execução com 16 threads é aproximadamente 1,5 vezes mais eficiente do que a execução com 2 threads.

A Figura 19 apresenta os resultados da Malha II - 100x124 com 2, 4, 8 e 16 threads. Nessa figura é possível notar que a execução com 16 threads possui um custo de sincronização superior às demais execuções, mas os blocos vermelhos que correspondem

Figura 19 – Casos de Teste OpenMP - Malha 100x124

a computação são significativamente menores. Os resultados numéricos apontam que a execução com 16 threads é aproximadamente 3 vezes mais eficiente do que a execução com 2 threads.

A Figura 20 apresenta os resultados da Malha III - 200x249 com 2, 4, 8 e 16 threads. Nessa figura é observável que os segmentos que compreendem a computação são significativamente menores para a execução com 16 threads e os segmentos de sincronização são menos aparentes do que nas malhas anteriores. Os resultados numéricos apontam que a execução com 16 threads é aproximadamente 3,1 vezes mais eficiente do que a execução com 2 threads.

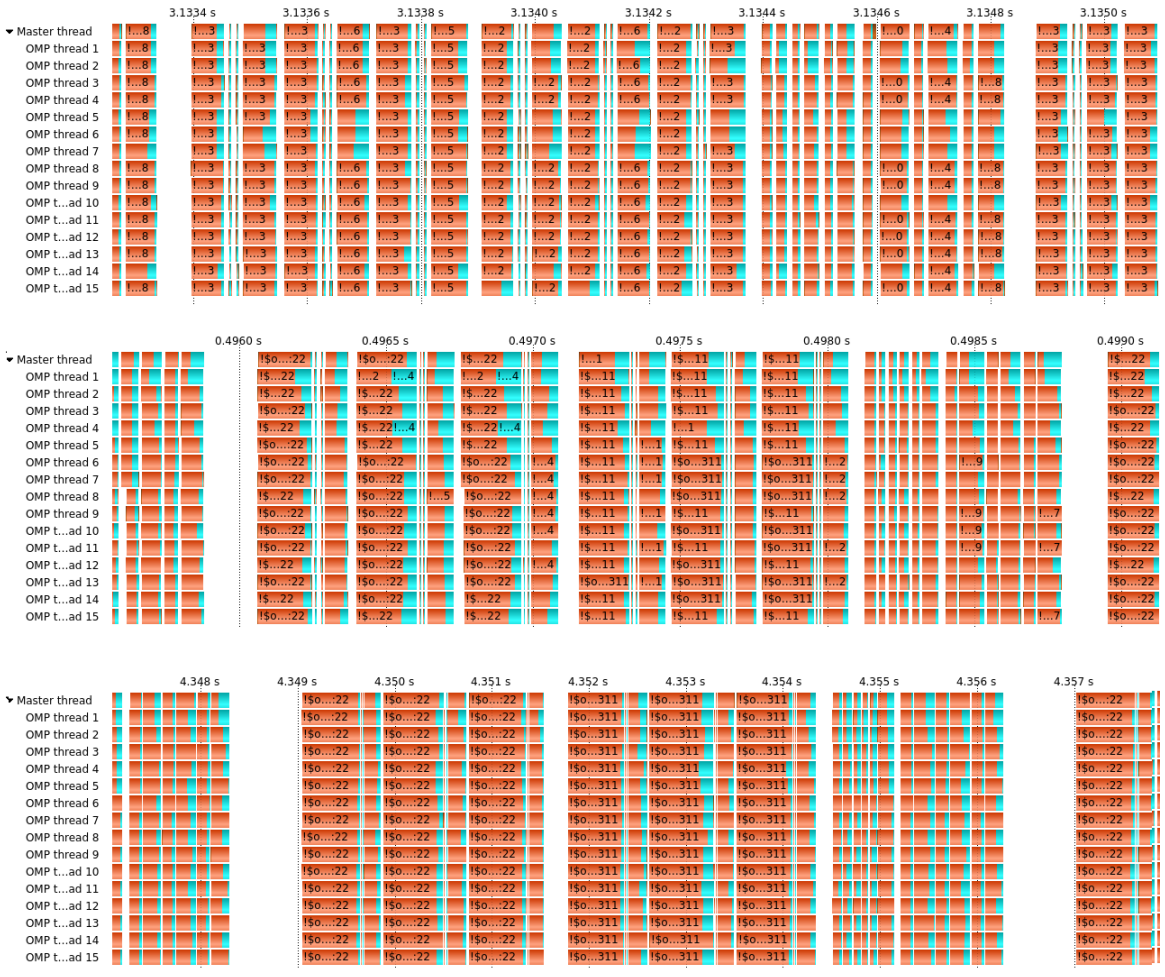
A Figura 21 apresenta as malhas I, II e III com 16 threads. Para a Malha I, Malha II e Malha III o *speedup* desse caso de teste é respectivamente 2,6, 5,5 e 5,6. Para a malha III é possível observar que os segmentos de sincronização não se sobressaem sobre os blocos de computação como ocorrem nas outras malhas, ou seja, a granularidade da computação para as malhas menores é baixa.

O comportamento das execuções com OpenMP *Teams* foram semelhantes às execuções com OpenMP *parallel do*. Para a malha I o tempo de execução do OpenMP *Teams* variou de 110 segundos a 175 segundos. Enquanto o OpenMP *parallel do* variou de 109 à 175 segundos. Para a malha II o tempo de execução do OpenMP *Teams* variou de 195 segundos a 658 segundos. Enquanto o OpenMP *parallel do* variou de 206 à 656 segundos. Para a malha III o tempo de execução do OpenMP *Teams* variou de 643 segundos a 2545 segundos. Enquanto o OpenMP *parallel do* variou de 860 à 2612 segundos.

O comportamento das execuções OpenMP Target e OpenACC obtiveram os resultados mais discrepantes entre os testes. Para a malha I, o OpenACC foi o pior resultado,

Figura 20 – Casos de Teste OpenMP - Malha 200x249

Figura 21 – Comparativo das malhas I, II e III com 16 Threads



com a maior média de execuções. Enquanto o OpenMP Target e 32 threads foi o melhor resultado. Comparando com o OpenMP Target e 32 threads o tempo de execução do OpenACC foi 2,7 vezes maior. Com relação ao caso de teste OpenMP Target e 2 threads o tempo de execução do OpenACC foi 1,5 vezes maior. Para a malha II, o OpenACC comparado com OpenMP Target e 32 threads foi 1,4 vezes pior. Já quando comparado com OpenMP Target e 2 threads o OpenACC foi 1,9 vezes melhor. Para a malha III, o caso de teste OpenACC obteve o melhor resultado. O OpenACC comparado com OpenMP Target e 32 threads foi 1,3 vezes melhor e quando comparado com OpenMP Target e 2 threads foi 4,3 vezes melhor. Apesar de ambas estrategias computarem os dados na GPU, os resultados não foram semelhantes. Enquanto o OpenMP Target obteve a melhor média para a malha I, o OpenACC obteve a pior média. Em contra ponto, para a malha III o OpenACC obteve a melhor média e o OpenMP Target (caso com 2 threads) obteve a pior média de execuções. Essa diferença pode ocorrer devido a maturidade do compilador com relação a interpretação das diretivas ou ainda pode ser necessário ajustar a diretiva para obter um melhor desempenho (LARREA et al., 2020).

7 CONSIDERAÇÕES FINAIS

Este capítulo tem por objetivo apresentar as considerações finais sobre a análise dos resultados. Também são descritos os possíveis trabalhos futuros e as publicações realizadas.

7.1 Conclusão

A dissertação estudou uma aplicação de meios porosos que simula o processo de secagem de grãos. A implementação original foi modificada para versões paralelas eficientes utilizando OpenMP para ambientes multi-core e OpenACC para GPU.

Nesta dissertação foram implementadas 4 versões de diretivas paralelas: OpenMP *parallel do*, *target* e *teams* e OpenACC *parallel do*, para a aplicação. Como destacado no decorrer do texto as versões OpenMP Target e OpenACC exigiram alterações na estrutura no código-fonte para que fosse possível paralelizar o código. O desempenho foi avaliado para três tamanhos de malha e para 2, 4, 8, 16 e 32 threads/times paralelos.

A avaliação dos experimentos foi apresentada no Capítulo 6. O melhor resultado foi obtido pelo caso de teste OpenACC *speedup* de 7,54. A avaliação dos casos de teste com um *profiling* não foi executada para todas as versões implementadas devido a limitações da ferramenta.

Os resultados mostraram a escalabilidade da implementação para diferentes números de thread e um aumento do *speedup* para malhas maiores. Em todas as malhas, a medida que o número de threads aumenta o *speedup* também aumenta.

As versões OpenMP *parallel do* e *Teams* apresentaram resultados semelhantes, como discutido no Capítulo 6. Os resultados mostram que os times e as threads são opções para a abordagem tradicional de paralelismo de *loop* paralelo usada em meios porosos e outras aplicações de demanda de alto desempenho semelhantes. Nos testes realizados ambas versões obtiveram resultados semelhantes.

As versões OpenACC e OpenMP Target obtiveram resultados diferentes. Enquanto o OpenMP Target foi eficiente para a malha menor, o OpenACC foi eficiente para os testes com a malha maior. Essa discrepância pode ocorrer devido às características do compilador ou a necessidade de aperfeiçoar as diretivas utilizadas para explorar o paralelismo nas versões. A aceleração utilizando uma única GPU foi de 7,54 para o tamanho de malha 200×249. Consideramos este valor aceitável devido ao tamanho desta malha não explorar todas as capacidades da GPU.

Apresentamos também alguns rastros de execução coletados variando o número de threads e o tamanho das malhas. Utilizamos a ferramenta Vampir para visualização de desempenho, ou rastreamento de execução, identificando as etapas paralelas e o tempo de sincronização. Os resultados mostraram os 36 *loops* paralelos chamados em cada etapa de convergência da implementação do OpenMP *parallel* do. A ferramenta permite identificar tempos de espera para sincronização e etapas sequenciais do código.

7.2 Trabalhos Futuros

Em trabalhos futuros pretendemos acrescentar a avaliação da interface de programação CUDA. Também pretendemos utilizar a ferramenta *nvprof* para avaliar a performance das APIs que utilizam GPU. Outra experiência inclui o uso de uma malha maior para verificar o desempenho dos casos de teste. Além disso, podemos considerar o compilador Clang, devido às citações na literatura e documentações de diversos trabalhos relacionados.

7.3 Publicações

Um artigo foi publicado em *journal* de Qualis A2:

- Hígor Uélinton da Silva, Claudio Schepke, Dalmo Paim de Oliveira, Natiele Lucca, César F. da C. Cristaldo. Parallel OpenMP and OpenACC Porous Media Simulation. The Journal of the Supercomputing. Springer Nature, 2022 (SILVA et al., 2022b).

Durante a realização do mestrado alguns artigos foram publicados em conferências:

- Natiele Lucca, Claudio Schepke, "Evaluation of SIMD Instructions on Bio-Inspired Algorithms,"2020 19th International Symposium on Parallel and Distributed Computing (ISPDC), 2020, pp. 52-59, doi: 10.1109/ISPDC51135.2020.00017 (LUCCA; SCHEPKE, 2020a).

- Natiele Lucca, Claudio Schepke. 2020. A New Library of Bio-Inspired Algorithms. In Computational Science and Its Applications – ICCSA 2020: 20th International Conference, Cagliari, Italy, July 1–4, 2020, Proceedings, Part I. Springer-Verlag, Berlin, Heidelberg, 474–484. DOI:https://doi.org/10.1007/978-3-030-58799-4_35 (LUCCA; SCHEPKE, 2020b).

- Natiele Lucca, Claudio Schepke. Operações vetoriais aplicadas em uma Biblioteca de Algoritmos Bio-inspirados. In: WORKSHOP EM DESEMPENHO DE SISTEMAS COMPUTACIONAIS E DE COMUNICAÇÃO (WPERFORMANCE), 19. , 2020, Cuiabá. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2020 . p. 169-174. ISSN 2595-6167. DOI: https://doi.org/10.5753/wperformance.2020.11116 (LUCCA; SCHEPKE, 2020c).

- Glener Lanes Pizzolato, Claudio Schepke, Natiele Lucca. Aceleração de uma aplicação de simulação de câmara de combustão em multi-core. In: Anais do XXII Simpósio em Sistemas Computacionais de Alto Desempenho. Porto Alegre, RS, Brasil: SBC, 2021. p. 36–47 (PIZZOLATO; SCHEPKE; LUCCA, 2021).

- Hígor Uélinton da Silva, Claudio Schepke, Natiele Lucca, Dalmo Paim de Oliveira, César F. da C. Cristaldo. An Efficient Parallel Model for Coupled Open-Porous Medium Problem Applied to Grain Drying Processing. In: Proceedings of Latin America High Performance Computing Conference. Guadalajara/Mexico, 2021 (SILVA et al., 2021).

- Hígor Uélinton da Silva, Claudio Schepke, Dalmo Paim de Oliveira, Natiele Lucca, César F. da C. Cristaldo. Parallel OpenMP and OpenACC Mixing Layer Simulation. In: Proceedings of Euromicro International Conference on Parallel, Distributed and Network-Based Processing. Valladolid/Spain: IEEE, 2022 (SILVA et al., 2022a).

- Glener Lanes Pizzolato, Natiele Lucca, Mariana Toledo Costa, Claudio Schepke. Parallel Mixing Chamber Application using OpenMP and OpenACC. In: Proceedings of Latin America High Performance Computing Conference. Porto Alegre/Brasil, 2022 (PIZZOLATO et al., 2022).

Os três primeiros artigos possuem relação com a continuidade das pesquisas do tema desenvolvido na graduação. O quarto artigo envolve avanços obtidos a partir de um trabalho desenvolvido na disciplina de Projeto e Desenvolvimento de Programas Paralelos (PPGES) e de um Trabalho de Conclusão de Curso. Os dois últimos artigos possuem relação direta com os resultados preliminares desta dissertação.

Também foram publicados dois capítulos de Livros que foram apresentados sob forma de minicurso na Escola Regional de Alto Desempenho:

- João Vicente Ferreira Lima; Claudio Schepke, Natiele Lucca. Além de Simplesmente: #pragma omp parallel for In: XXI Escola Regional de Alto Desempenho / Região Sul (ERAD /RS 2021).1, 2021, p. 1-20. ISBN: 9786587003009. (LIMA; SCHEPKE; LUCCA, 2021).

- Claudio Schepke, Natiele Lucca. Programando Aplicações com Diretivas Paralelas In: Minicursos da XX Escola Regional de Alto Desempenho da Região Sul.1 ed. Porto Alegre: SBC, 2020, p. 89-104. ISBN: 9786587003009 (LUCCA; SCHEPKE, 2020d).

REFERÊNCIAS

AGUIRRE, L. A. **Introdução à Identificação de Sistemas–Técnicas Lineares e Não-lineares Aplicadas a Sistemas Reais**. Minas Gerais: Editora UFMG, 2004. Citado na página 25.

ALVES, L. M. Métodos dos Elementos Finitos. **Departamento de Engenharia Civil. Universidade Federal do Paraná, Curitiba**, 2007. Citado na página 25.

ANSARI, Z. et al. Literature survey for the comparative study of various high performance computing techniques. **Int J Comput Trends Technol (IJCTT)**, v. 27, n. 2, p. 80–86, 2015. Citado na página 22.

ANTON, H.; BIVENS, I.; DAVIS, S. **Cálculo I**. 10. ed. Porto Alegre: Bookman Editora, 2014. v. 1. ISBN 9788582602454. Citado na página 26.

BALA, B. K. **Drying and Storage of Cereal Grains**. Jessore, Bangladesh: John Wiley & Sons, 2016. Citado na página 37.

BERTOLLI, C. et al. Coordinating gpu threads for openmp 4.0 in llvm. In: **LLVM Compiler Infrastructure in HPC**. New Orleans, LA, USA: IEEE, 2014. p. 12–21. Citado na página 29.

CHANDRASEKARAN, S.; JUCKELAND, G. **OpenACC for Programmers: Concepts and Strategies**. Albuquerque, EUA: Addison-Wesley Professional, 2017. Citado na página 31.

COETZEE, C. Calibration of the Discrete Element Method. **Powder Technology**, Elsevier, v. 310, p. 104–142, 2017. Citado na página 27.

COURTÈS, L. C Language Extensions for Hybrid CPU/GPU Programming with StarPU. **CoRR**, abs/1304.0878, 2013. Disponível em: <http://arxiv.org/abs/1304.0878>. Citado na página 21.

DALEY, C. et al. A case study of porting hpgmg from cuda to openmp target offload. In: MILFELD, K. et al. (Ed.). **OpenMP: Portable Multi-Level Parallelism on Modern Systems**. Cham: Springer International Publishing, 2020. p. 37–51. ISBN 978-3-030-58144-2. Citado 2 vezes nas páginas 33 e 34.

DONGARRA, J. et al. **Sourcebook of parallel computing**. San Francisco: Morgan Kaufmann, 2003. v. 3003. Citado na página 57.

DURAN, A. et al. Extending the OpenMP Tasking Model to Allow Dependent Tasks. In: EIGENMANN, R.; SUPINSKI, B. R. de (Ed.). **OpenMP in a New Era of Parallelism**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 111–122. ISBN 978-3-540-79561-2. Citado 2 vezes nas páginas 28 e 29.

EYMARD, R.; GALLOUëT, T.; HERBIN, R. Finite volume methods. In: **Solution of Equation in Rn (Part 3), Techniques of Scientific Computing (Part 3)**. Elsevier, 2000, (Handbook of Numerical Analysis, v. 7). p. 713–1018. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1570865900070058>. Citado na página 27.

FIGUEIREDO, R. A. **Simulação numérica de escoamentos viscoelásticos multifásicos complexos**. Tese (Doutorado) — Universidade de São Paulo, 2016. Citado na página 26.

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The implementation of the cilk-5 multithreaded language. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 33, n. 5, p. 212–223, maio 1998. ISSN 0362-1340. Disponível em: <https://doi.org/10.1145/277652.277725>. Citado na página 28.

GALARZA, E. L. V. Taxonomías de flynn. 2020. Citado na página 28.

GARCIA, S. et al. A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning. **IEEE Transactions on Knowledge and Data Engineering**, IEEE, v. 25, n. 4, p. 734–750, 2012. Citado na página 25.

GOULD, H.; TOBOCHNIK, J.; CHRISTIAN, W. An introduction to computer simulation methods. **Comput. Phys**, Open Source Physics, 2016. Citado na página 25.

GOYAL, A.; LI, Z.; KIMM, H. Comparative study on edge detection algorithms using openacc and openmpi on multicore systems. In: **11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)**. Seul, Coreia do Sul: IEEE, 2017. p. 67–74. Citado na página 34.

GUNAWAN, P. H. Scientific parallel computing for 1d heat diffusion problem based on openmp. In: **4th International Conference on Information and Communication Technology (ICoICT)**. Bandung, Indonésia: IEEE, 2016. p. 1–5. Citado 2 vezes nas páginas 34 e 35.

GUO, X. et al. Parallel computation of aerial target reflection of background infrared radiation: Performance comparison of openmp, openacc, and cuda implementations. **IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing**, v. 9, n. 4, p. 1653–1662, 2016. Citado 2 vezes nas páginas 33 e 34.

JULIATI, S.; GUNAWAN, P. H. Openmp architecture to simulate 2d water oscillation on paraboloid. In: **5th International Conference on Information and Communication Technology (ICoIC7)**. Melaka, Malásia: IEEE, 2017. p. 1–5. Citado na página 34.

KHALILOV, M.; TIMOVEEV, A. Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA v100 GPU. **Journal of Physics: Conference Series**, IOP Publishing, v. 1740, n. 1, p. 012056, jan 2021. Disponível em: <https://doi.org/10.1088/1742-6596/1740/1/012056>. Citado 2 vezes nas páginas 33 e 34.

KNÜPFER, A. et al. Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir. In: BRUNST, H. et al. (Ed.). **Tools for High Performance Computing 2011**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 79–91. ISBN 978-3-642-31476-6. Citado na página 58.

LARREA, V. G. V. et al. Experiences in porting mini-applications to openacc and openmp on heterogeneous systems. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 32, n. 20, p. e5780, 2020. Citado 3 vezes nas páginas 33, 34 e 69.

LIMA, J. V. F.; SCHEPKE, C.; LUCCA, N. Além de simplesmente: #pragma omp parallel for. In: **Minicursos da XXI Escola Regional de Alto Desempenho da Região Sul**. Joinville, Santa Catarina: Sociedade Brasileira de Computação, 2021. p. 1–18. ISBN 978-65-87003-00-9. Citado na página 73.

Lobato Gimenes, T.; Pisani, F.; Borin, E. Evaluating the performance and cost of accelerating seismic processing with cuda, opencl, openacc, and openmp. In: **IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. Vancouver, Canadá: IEEE, 2018. p. 399–408. ISSN 1530-2075. Citado 2 vezes nas páginas 33 e 34.

LUCCA, N.; SCHEPKE, C. Evaluation of simd instructions on bio-inspired algorithms. In: **19th International Symposium on Parallel and Distributed Computing (ISPDC)**. Varsóvia, Polônia: IEEE, 2020. v. 1, p. 52–59. Citado na página 72.

LUCCA, N.; SCHEPKE, C. A new library of bio-inspired algorithms. In: **Computational Science and Its Applications – ICCSA 2020: 20th International Conference, Cagliari, Italy, July 1–4, 2020, Proceedings, Part I**. Berlin, Heidelberg: Springer-Verlag, 2020. p. 474–484. ISBN 978-3-030-58798-7. Disponível em: <https://doi.org/10.1007/978-3-030-58799-4_35>. Citado na página 72.

LUCCA, N.; SCHEPKE, C. Operações vetoriais aplicadas em uma biblioteca de algoritmos bio-inspirados. In: **Anais do XIX Workshop em Desempenho de Sistemas Computacionais e de Comunicação**. Porto Alegre, RS, Brasil: SBC, 2020. p. 169–174. ISSN 2595-6167. Disponível em: <https://sol.sbc.org.br/index.php/wperformance/article/view/11116>. Citado na página 72.

LUCCA, N.; SCHEPKE, C. Programando aplicações com diretivas paralelas. In: **Minicursos da XX Escola Regional de Alto Desempenho da Região Sul**. Santa Maria, Rio Grande do Sul: Sociedade Brasileira de Computação, 2020. p. 89–104. ISBN 978-65-87003-50-4. Citado na página 73.

MISHRA, B.; RAJAMANI, R. K. The discrete element method for the simulation of ball mills. **Applied Mathematical Modelling**, Elsevier, v. 16, n. 11, p. 598–604, 1992. Citado na página 27.

MITCHELL, A. Of the finite element method. In: ACADEMIC PRESS. **The Mathematics of Finite Elements and Applications**. London, Reino Unido: Proceedings of the Brunel University Conference of the Institute of Mathematics and Its Applications, 2014. p. 37. Citado na página 26.

MOUKALLED, F. et al. **The finite volume method in computational fluid dynamics**. Suíça: Springer, 2016. v. 113. Citado na página 27.

MUSTOE, G. G. A generalized formulation of the discrete element method. **Engineering computations**, MCB UP Ltd, 1992. Citado na página 27.

NVIDIA. **CUDA C Programming Guide**. 2020. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Accessado em: 2020-02-28. Citado na página 21.

OLIVEIRA, D. P. **Fluid Flow Through Porous Media with the One Domain Approach: A Simple Model for Grains Drying.** Dissertação (Mestrado) — Dissertação (Dissertação de Mestrado) Universidade Federal do Pampa - UNIPAMPA, Alegrete, 2022. Citado na página 43.

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 6 vezes nas páginas 37, 38, 39, 41, 42 e 43.

OPENMP. **The OpenMP API specification for parallel programming**. 2018. [Online; acesso 01 Dezembro 2021]. Disponível em: <https://www.openmp.org/>. Citado na página 29.

OPENMP. **OPENMP API Specification: Version 5.0**. 2018. <https://www.openmp.org/spec-html/5.0/openmpsu43.html>. Accessado em: 2022-09-24. Citado na página 55.

PARRY, J. Mathematical modelling and computer simulation of heat and mass transfer in agricultural grain drying: A review. **Journal of Agricultural Engineering Research**, Elsevier, v. 32, n. 1, p. 1–29, 1985. Citado na página 37.

PIZZOLATO, G.; SCHEPKE, C.; LUCCA, N. Aceleração de uma aplicação de simulação de câmara de combustão em multi-core. In: **Anais do XXII Simpósio em Sistemas Computacionais de Alto Desempenho**. Porto Alegre, RS, Brasil: SBC, 2021. p. 36–47. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/wscad/article/view/18510>. Citado na página 72.

PIZZOLATO, G. L. et al. Parallel mixing chamber application using openmp and openacc. In: **Proceedings of Latin America High Performance Computing Conference**. Porto Alegre, Brasil: Springer, 2022. Citado na página 73.

PRATX GUILLEM E XING, L. computação gpu em física médica: uma revisão. **física médica**, Wiley Online Library, v. 38, n. 5, p. 2685–2697, 2011. Citado na página 21.

RAJ, A. et al. Acceleration of a 3d immersed boundary solver using openacc. In: **25th International Conference on High Performance Computing Workshops (HiPCW)**. Bengaluru, Índia: IEEE, 2018. p. 65–73. Citado 2 vezes nas páginas 34 e 35.

SCORE-P. **Manual Score-P**. 2022. [Online; acesso 01 Setembro 2022]. Disponível em: <https://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-6.0/html/installationfile.html>. Citado na página 65.

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**. Valladolid, Espanha: IEEE, 2022. Citado na página 73.

SILVA, H. U. et al. Parallel openmp and openacc porous media simulation. **Computing**, Springer, In press, n. In press, 2022. Citado na página 72.

SILVA, H. U. et al. An efficient parallel model for coupled open-porous medium problem applied to grain drying processing. In: **Proceedings of Latin America High Performance Computing Conference**. Cham: Springer, 2021. Citado na página 73.

SIMANJUNTAK, C. A.; GUNAWAN, P. H. Computing two-layer swe for simulating submarine avalanches on openmp. In: **International Conference on Control, Electronics, Renewable Energy and Communications (ICCREC)**. Yogyakarta, Indonesia: IEEE, 2017. p. 190–195. Citado na página 34.

SOBIESKI, W. Examples of using the finite volume method for modeling fluid-solid systems. **Technical Sciences**, Citeseer, v. 13, p. 256–265, 2010. Citado na página 37.

SOUZA, R. M. de. O método dos elementos finitos aplicado ao problema de condução de calor. **Apostila, Universidade Federal do Pará, Belém**, 2003. Citado 2 vezes nas páginas 25 e 27.

VERSTEEG, H. K.; MALALASEKERA, W. **An introduction to computational fluid dynamics: the finite volume method**. London: Pearson Education, 2007. Citado 2 vezes nas páginas 38 e 42.

VETTER, J. S. **Contemporary High Performance Computing: from Petascale Toward Exascale**. Chapman: CRC Press, 2013. Citado na página 21.

XU, J. et al. Quasi-real-time simulation of rotating drum using discrete element method with parallel gpu computing. **Particuology**, v. 9, n. 4, p. 446–450, 2011. ISSN 1674-2001. Multiscale Modeling and Simulation of Complex Particulate Systems. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1674200111000654>. Citado na página 22.

ÍNDICE

DEM, 25, 26

FEM, 25, 26
FVM, 25, 26

HPC, 21

MIMD, 28

OpenMP, 35

SIMD, 28