UNIVERSIDADE FEDERAL DO PAMPA

Glener Lanes Pizzolato

Desafios na Paralelização de uma Aplicação de Câmara de Mistura

Alegrete
2021

Glener Lanes Pizzolato

Desafios na Paralelização de uma Aplicação de Câmara de Mistura

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

Orientador: Prof. Dr. Claudio Schepke

Alegrete
2021

GLENER LANES PIZZOLATO

DESAFIOS NA PARALELIZAÇÃO DE UMA APLICAÇÃO DE CÂMARA DE MISTURA

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

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

Banca examinadora:

_____

Prof. Dr. Claudio Schepke

UNIPAMPA

_____

Prof. Dra. Aline Vieira de Mello

UNIPAMPA

_____

Bel. Matheus Castro Nicolau da Silva

UNIPAMPA

Assinado eletronicamente por **ALINE VIEIRA DE MELLO**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 10/03/2022, às 21:34, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **CLAUDIO SCHEPKE**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 10/03/2022, às 21:43, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **MATHEUS CASTRO NICOLAU DA SILVA**, **Usuário Externo**, em 10/03/2022, às 21:44, 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 **0750198** e o código CRC **7522CD61**.

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) .

P695d Pizzolato, Glener Lanes
Desafios na Paralelização de uma Aplicação de Câmara de Mistura / Glener Lanes Pizzolato.
59 p.

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

1. Otimização de Código. 2. Programação Paralela. 3. Câmara de Mistura. 4. OpenMP. 5. OpenACC. I. Título.

AGRADECIMENTOS

Gostaria de expressar meus sinceros agradecimentos aos meus pais, Glecio Brum Pizzolato e Eliane Lanes Pizzolato que me apoiaram incondicionalmente neste desafio de sair de casa para estudar longe, e que nada seria possível sem este apoio.

Ao Dr. Claudio Schepke por todo conhecimento compartilhado. Saibas que a tua atenção e compreensão comigo durante esta caminhada me manteve sempre motivado e dedicado com este trabalho, muito obrigado!

A todos meus amigos, que em todos momentos juntos me transmitiam energias positivas que me acalmavam e não me deixavam desistir, todos vocês fizeram parte desta etapa da minha vida e isto sempre será lembrado.

RESUMO

Um programa para realizar simulações de uma câmara de mistura foi modelado utilizando as equações de Navier-Stokes e discretizado em um modelo bidimensional. O programa foi implementado de forma sequencial em Fortran90. Consequentemente possui um tempo computacional expressivo, levando horas para computar todos os dados. Neste trabalho é proposto a otimização desse programa utilizando programação paralela. Para acelerar a execução do código foram utilizadas operações paralelas fornecidas pelas interfaces de programação paralela OpenMP e OpenACC, usando duas malhas de tamanhos distintos como estudo de caso. Dessa forma foi possível calcular o ganho de desempenho ao se utilizar às APIs com diferentes tamanhos de entradas para o problema, mensurando o quão eficaz se tornou a programação paralela para este problema, desde uma malha pequena até uma malha mais complexa, precisa e, consequentemente, mais custosa. Com os resultados dos testes, comprovou-se que a versão paralela do algoritmo desenvolvido mantém a qualidade da solução e reduz o tempo de execução para as duas malhas utilizadas nos experimentos dentro dos limites possíveis para a arquitetura e trechos paralelizáveis do código.

Palavras-chave: Otimização. Tempo. OpenMP. OpenACC. Câmara de mistura.

ABSTRACT

A program to perform simulations of a mixing chamber was modeled using the Navier-Stokes equations and discretized in a two-dimensional model. The program was implemented sequentially in Fortran90. Consequently it has an expressive computational time, taking hours to compute all the data. This work proposes the optimization of this program using parallel programming. To speed up code execution, parallel operations provided by the parallel programming interfaces OpenMP and OpenACC were used, using two meshes of different sizes as a case study. In this way, it was possible to calculate the performance gain when using APIs with different sizes of inputs for the problem, measuring how effective parallel programming has become for this problem, from a small mesh to a more complex, precise mesh and, consequently, more costly. With the test results, it was proved that the parallel version of the developed algorithm maintains the quality of the solution and reduces the execution time for the two meshes used in the experiments within the possible limits for the architecture and parallelizable parts of the code.
Key-words: Optimization. Time. OpenMP. OpenACC. Mixing chamber.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A simulação computacional possibilita representar discretamente ambientes, sistemas ou equipamentos, sem a necessidade da construção ou existência dos mesmos em um mundo real (GOULD; TOBOCHNIK; CHRISTIAN, 2016). Prever falhas, realizar diversos testes sem risco de segurança e com baixíssimo custo financeiro, além de conseguir representar algumas situações improváveis do ambiente real, são alguns exemplos do que pode ser alcançado através da simulação computacional. Diante disso, a simulação tornou-se indispensável para a maioria das áreas da ciência e experimentação, sendo esta uma das principais contribuições da computação.

Um exemplo de aplicação computacional que provê a capacidade de simulação de uma câmara de mistura foi desenvolvido por (MANCO, 2014). A aplicação tem como objetivo avaliar a mistura de substâncias, como é o caso de combustível e oxidante em uma combustão. Para tanto, o modelo físico é descrito através das equações de Euler em um espaço bidimensional.

(SILVA et al., 2017) adaptaram a aplicação para que o mesmo usasse as equações de Navier-Stokes como descrição do modelo físico. Isso amplia a possibilidade de representação de fluidos com diferentes características físicas. A câmera de mistura compressível serve como um modelo para a análise e a capacidade de simular problemas como, por exemplo: propulsão de ar de alta velocidade, mistura de reagentes, geração de ruído em bocais de exaustão, etc. Em todos os casos, duas correntes paralelas em velocidades diferentes podem ser compostas de diferentes espécies químicas ou com grandes diferenças de temperatura. Essas simulações numéricas diretas de alta ordem são frequentemente usadas para resolver as escalas espaço-temporais de tais fluxos.

É possível agilizar esse tipo de aplicação mas com alguns cuidados. A maioria das operações de código são sobre *loops* que percorrem as estruturas de dados das propriedades físicas. Mas questões como dependência de dados entre seções de código, necessidade de sincronizar variáveis ou usar variáveis privadas, resultados numéricos consistentes com a versão sequencial e *loops* de ordem pequena influenciam a forma de como a paralelização deve ser feita. Assim, **a contribuição do artigo** é conhecer e identificar esses contextos específicos para apresentá-los e explorá-los.

1.1 Objetivos Gerais

Simulações como o da câmara de mistura requerem alta precisão numérica o que geralmente resulta em um custo de processamento computacional significativo. Uma simulação típica demora em torno de 2 horas.

A implementação original da aplicação de simulação não explora a concorrência na execução do código em arquiteturas Multicore e GPU (*Graphics Processing Unit*). Diante disso, o objetivo deste trabalho é otimizar e paralelizar a computação dos valores numéricos das propriedades físicas do problema. Desta forma, a aplicação de simula-

ção da câmera de mistura é executada em múltiplos fluxos de execução concorrente em arquiteturas Multicore e GPU.

1.2 Objetivos Específicos

O objetivo específico deste trabalho é realizar a aceleração da aplicação através da utilização das operações paralelas oferecidas pelas interfaces de programação paralela OpenMP (OPENMP, 2022) e OpenACC (OPENACC, 2022). As interfaces proveem a criação implícita de threads tanto em laços, como em seções paralelas, através do uso de diretivas de compilação. As interfaces são simples e, em teoria, fáceis de se usar, sendo compatível com a linguagem de programação FORTRAN 90, utilizada na codificação da aplicação.

Além de acelerar a aplicação, deseja-se apresentar uma abordagem de como a paralelização pode ser feita para cada uma das interfaces de programação escolhidas. Embora a grande maioria das operações do código sejam do tipo laços que percorrem as estruturas de dados das propriedades físicas, questões como: dependência de dados entre seções de código, necessidade de sincronização de variáveis ou uso de variáveis privadas, resultados numéricos consistentes com a versão sequencial, laços de pequena ordem de grandeza e cópia de dados entre CPU a GPU influenciam na forma como a paralelização deve ser feita. Conhecer e identificar esses contextos específicos são explorados e os resultados são apresentados.

1.3 Organização do documento

O trabalho está organizado da seguinte forma. O Capítulo 2 detalha os trabalhos relacionados com o tema da pesquisa. O Capítulo 3 apresenta a fundamentação matemática da aplicação, a definição do algoritmo do problema e a relação esquemática entre as rotinas usadas na implementação. O Capítulo 4 traz uma introdução sobre os conceitos de programação paralela e sobre o funcionamento das interfaces OpenMP e OpenACC, incluindo suas diretivas e variáveis. O Capítulo 5 explica os métodos e passos aplicados no desenvolvimento das versões paralelas do código da aplicação. O Capítulo 6 e o Capítulo 7 apresentam os resultados de performance obtidos com as diferentes versões implementadas para as interfaces de programações OpenMP e OpenACC para as diferentes malhas utilizadas como estudo de caso tais como os desafios e obstáculos da implementação. Por fim, no Capítulo 8 são destacadas as considerações finais sobre os resultados obtidos e os trabalhos futuros.

2 TRABALHOS RELACIONADOS

A ideia de otimizar e deixar aplicações mais rápidas por meio de programação paralela é constantemente utilizada (FOSTER, 1995). Reduzir o tempo de processamento possibilita a obtenção de respostas em um período de tempo menor. Este capítulo aborda alguns trabalhos relacionados que utilizam APIs paralelas para a execução concorrente de aplicações. As informações apresentadas dos trabalhos relacionados analisam aplicações, metodologias, comparações, testes e resultados obtidos.

Os trabalhos relacionados foram selecionados em buscas feitas na plataforma da *IEEE* e *Google Acadêmico*, seguindo as chaves de buscas: *OpenMP Applications*; *OpenACC Applications*; *OpenMP*; *OpenACC*. A Tabela 1 mostra uma comparação das interfaces de programação paralela e objetivos de cada um dos 6 trabalhos selecionados. Os trabalhos selecionados foram:

1. (ALDINUCCI et al., 2021) propõe a paralelização de quatro aplicações científicas projetadas com uma programação puramente sequencial. É aplicado uma metodologia sistemática a fim de transformar uma base de código antiga em código moderno, ou seja, código paralelo e robusto.

2. (KHALILOV; TIMOVEEV, 2021) compara as APIs CUDA e OpenACC levando em consideração que o OpenACC é uma API mais alto nível onde é necessário apenas localizar os blocos paralelizáveis e utilizar os *pragmas* para realizar de fato a

Tabela 1 – Contexto dos Trabalhos Relacionados

Nome do Artigo	**Interfaces Utilizadas**	**Tipo de Trabalho**
[**1**]-Practical Parallelization of Scientific Applications with OpenMP, OpenACC and MPI	OpenMP, OpenACC e MPI	Realiza a paralelização de quatro aplicações sequenciais
[**2**]-Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU	CUDA, OpenMP e OpenACC	Compara as duas APIs
[**3**]-Experiences in porting mini-applications to OpenACC and OpenMP on heterogeneous systems	OpenACC e OpenMP	Metodologia para migrar pequenas aplicações para OpenMP e OpenACC
[**4**]-Parallel programming languages on heterogeneous architectures using OpenMPC, OmpSs, OpenACC and OpenMP	OpenMPC, OpmSs, OpenACC e OpenMP	Revisa três estruturas de programação que resolvem o método iterativo de Jacobi
[**5**]-Parallel Computation of a Dam-Break Flow Model Using OpenACC and OpenMP	OpenACC e OpenMP	Paraleliza um modelo de simulação de fluxo *dam-break*
[**6**]-Concurrent Parallel Processing on Graphics and Multicore Processors with OpenACC and OpenMP	OpenMP, OpenACC e MPI	Explora um sistema híbrido de memória compartilhada e distribuída

Fonte: O Autor.

paralelização. Para os testes foi utilizado a GPU *NVIDIA Tesla V100* em cenários típicos que surgem na programação, como a multiplicação de matrizes em alguns códigos de simulação física, que foram implementados.

3. O trabalho (LARREA et al., 2020) apresenta puramente a metodologia utilizada para migrar pequenas aplicações (*MiniSweep*, *GenASiS*, *GPP* e *FF*) para versões paralelas com OpenMP e OpenACC.

4. O Artigo (HERNANDEZ; PALACIOS; MARíN, 2015) revisa estruturas para a programação de dispositivos gráficos usando CUDA, comparando com as novas diretivas introduzidas no OpenMP 4. O método iterativo de Jacobi foi usado como estudo de caso. Os desafios dos limites de compatibilidade com linguagens de programação com as APIs OpenMP e OpenACC são claros, visto que suportam apenas as linguagens C, C++ e Fortran. Conforme (OPENMP, 2022), OpenMP 4 possui suporte para aceleradores, construções *SIMD* para vetorizar *loops* seriais e paralelizados, manipulação de erros, reduções definidas pelo usuário, etc.

5. Em (ZHANG et al., 2016), um modelo de simulação de fluxo *dam-break* de alto desempenho foi utilizado. Um esquema explícito para discretizar as equações de controle de águas rasas 2D foi definido e a solução aproximada de *Riemann de Roe* do método de volumes finitos foi adotada para o fluxo de interface dos pontos da malha. Neste caso não há correlação entre os cálculos da malha em uma etapa de tempo. Então o modelo pode ser facilmente paralelizado com as APIs OpenMP e OpenACC, onde se obteve $17,64$ e $8,6$ de speedup (S) para cada interface.

6. O artigo (STONE ROGER L. DAVIS, 2018) apresenta métodos para a programação paralela com MPI (Memória Distribuída), visando aproveitar ao máximo cada nó do sistema com computação heterogênea, utilizando balanceamento de carga automatizado para determinar quais partes são calculadas pelas CPUs e quais partes pela GPUs. Os resultados foram obtidos sobre modelos de programação usados comumente em turbomáquinas e códigos de simulação de engenharia. Os mesmos indicam que uma aceleração geral moderada é possível em um sistema híbrido de memória compartilhada e distribuída.

Todos os trabalhos relacionados avaliam as interfaces de programação paralela OpenMP e OpenACC. Alguns consideram também outras interfaces. O artigo (ALDINUCCI et al., 2021) avalia a paralelização de 4 códigos, mas para cada um é utilizado uma interface ou arquitetura específica. O artigo (KHALILOV; TIMOVEEV, 2021) traz uma avaliação de arquitetura específica de GPU através de dois *benchmarks*. O artigo (LARREA et al., 2020) avalia a performance de 4 rotinas/métodos em 5 ambientes computacionais, explorando paralelismo heterogêneo (CPU/GPU) das especificações mais recentes de OpenMP e OpenACC. O método de Jacobi foi utilizado para avaliar o paralelismo de

interfaces que proveem paralelismo de tarefas em (HERNANDEZ; PALACIOS; MARíN, 2015). No entanto, os códigos apresentados mostram paralelismo de laços nas versões OpenMP e CUDA. Já os resultados apresentado limitam-se a OpenMP 4, OpenACC, OpenMPC. O artigo (ZHANG et al., 2016) paraleliza com OpenMP e OpenACC uma aplicação de controle de águas rasas em 2D, utilizando discretização por volumes finitos. O artigo (STONE ROGER L. DAVIS, 2018) também utiliza as interfaces OpenMP e OpenACC para uma aplicação chamada de MBFLO3 que provê solução multidisciplinar de propósito geral 3D utilizando malhas estruturadas discretizadas por volumes finitos. Os artigos (ZHANG et al., 2016) e (STONE ROGER L. DAVIS, 2018) são similares a nossa proposta, mudando essencialmente a aplicação em si, enquanto o artigo (HERNANDEZ; PALACIOS; MARíN, 2015) é apenas a paralelização de um método numérico.

3 SIMULAÇÃO DE UMA CÂMARA DE MISTURA

O desenvolvimento de motores, combustíveis e a avaliação da capacidade de oxidação e catálise pode ser feita através da representação computacional discreta da estrutura e dos fenômenos físicos associados (LIPATNIKOV, 2020). Assim, o desempenho dos sistemas de mistura em turbinas de motores de foguete depende fortemente na injeção adequada e mistura entre combustível e oxidante (MANCO, 2020). Como um exemplo de mistura, a Figura 1 mostra os 122 injetores coaxiais de cisalhamento usada no motor principal do ARIANE 6 da Agência Espacial Europeia. Cada um tem o papel de misturar o oxigênio líquido (LOX) e o hidrogênio gasoso (GH2).

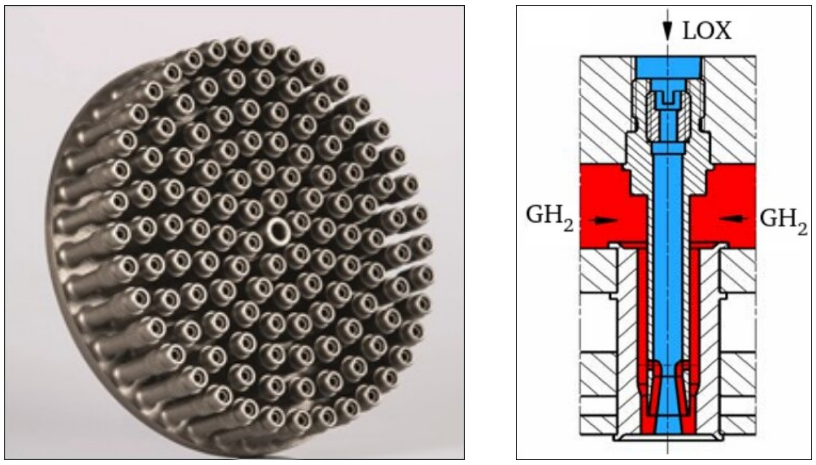
Para simular o processo de mistura em câmeras, (MANCO, 2014) desenvolveu uma aplicação. A representação da estrutura do modelo matemático utilizado é apresentada na Figura 2. É importante enfatizar a região de interesse da câmara, isto é, onde ocorrem as interações entre substâncias (como combustível e oxidante, em uma combustão), que são modeladas através das equações Navier-Stokes. No modelo foi utilizado uma condição de contorno (NRBC), indicado por D_x e D_y, no qual os vértices turbulentos não são reinjetados nas equações. Todo domínio x,y é considerado durante a execução da aplicação, mas o resultado final é definido pela computação discreta das equações de Navier-Stokes.

3.1 Formulação Numérica da Câmara de Mistura

A aplicação aborda a questão de como os gradientes de temperatura afetam o desenvolvimento da instabilidade de Kelvin-Helmholtz 3. Essa instabilidade é decorrente da camada cisalhante devido ä diferença de velocidade na interface de 2 fluídos. Neste caso, há dois fluidos com velocidades distintas misturando-se, gerando vórtices. Diante disso, é importante compreender este fenômeno.

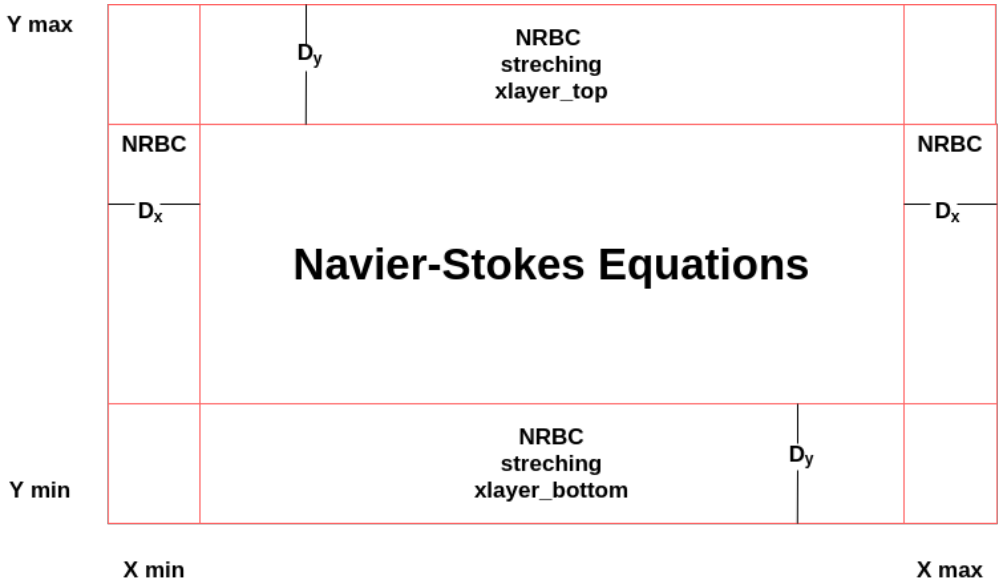
A aplicação contribui para o cálculo da estabilidade e acústica de escoamentos

Figura 1 – (a) Injetores coaxiais do motor principal (b) Visão esquemática de um injetor.



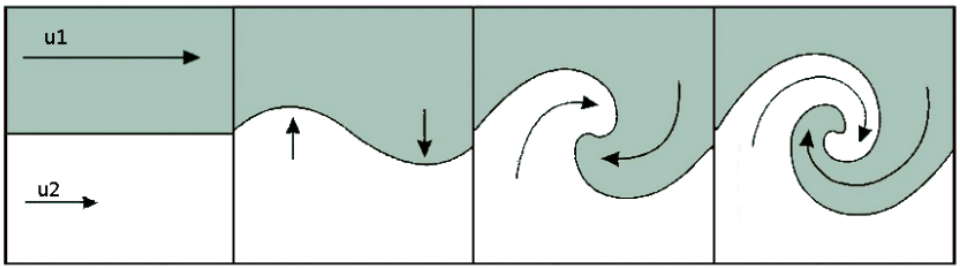
Fonte: (MANCO, 2020)

Figura 2 – Estrutura da Câmara de Mistura



Fonte: Autor, Baseado em (MANCO; MENDONCA, 2019)

Figura 3 – Instabilidade de Kelvin-Helmholtz

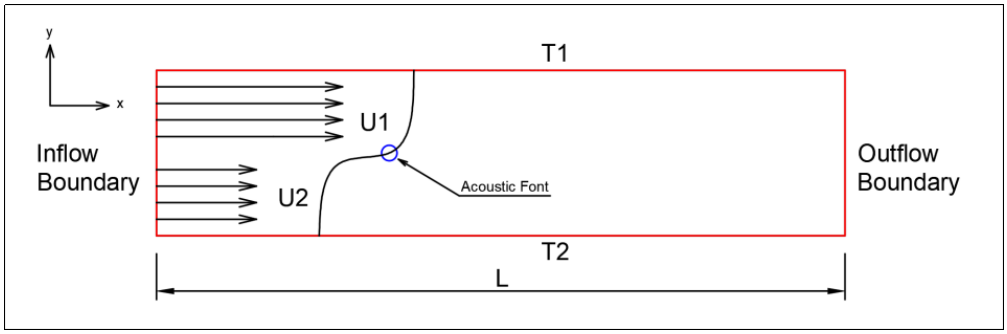


Fonte: (PAPERIN, 2007)

compressíveis relevantes para sistemas de propulsão. Para todos os casos dos experimentos realizados, duas correntes paralelas em velocidades diferentes podem ser compostas por diferentes espécies químicas ou com grandes diferenças de temperatura, conforme Figura 4.

O problema é solucionado através da simulação numérica de um fluxo de camada de mistura compressível. As equações governantes são as equações de Navier-Stokes para escoamento compressível e inerte em duas dimensões, sem forças corporais, fontes de calor

Figura 4 – Região da Camada de Mistura



Fonte: (SILVA, 2020)

e radiação, conforme descrito na Equação 3.1.

$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho e \end{bmatrix}, \; E = \begin{bmatrix} \rho u \\ \rho u^2 + p - \tau_{xx} \\ \rho uv - \tau_{xy} \\ \rho eu + q_x - u\tau_{xx} - v\tau_{xy} \end{bmatrix}, \tag{3.1}$$

$$F = \begin{bmatrix} \rho v \\ \rho uv - \tau_{xy} \\ \rho v^2 + p - \tau_{yy} \\ \rho ev + q_y - u\tau_{xy} - v\tau_{yy} \end{bmatrix}$$

Assim, o conjunto de equações é definido conforme apresentado na Equação 3.2.

$$\frac{\partial U}{\partial t} + \frac{\partial E}{\partial x} + \frac{\partial F}{\partial y} = 0. \tag{3.2}$$

Onde as tensões viscosas estão relacionadas linearmente ao tensor de deformação, conforme é mostrado na Equação 3.3.

$$\tau_{ij} = -\frac{2}{3}\mu \frac{\partial u_k}{\partial x_k} + \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right). \tag{3.3}$$

Essas equações são adimensionais, usando as variáveis de fluxo superior como condições de referência e a espessura de vorticidade da camada de mistura como o comprimento de referência, tal como descrito na Equação 3.4.

$$Re = \frac{\rho_1 U_1 \delta_w}{\mu_1}, Pr = \frac{\mu_1 cp_1}{k_1}, Ma = \frac{U_1}{a_1} \tag{3.4}$$

Onde, ρ é a densidade, U_1 é a velocidade do fluxo rápido, μ_1 é o coeficiente de viscosidade do fluxo rápido, c_p é o calor específico a pressão constante, k_1 é a condutividade e a_1 é a velocidade do som. O subscrito 1 refere-se ao fluxo rápido.

A metodologia numérica é baseada em esquemas de alta ordem, baixa dissipação e baixa dispersão. Para a simulação são usados métodos numéricos de alta ordem e

diferentes condições de contorno não reflexivas. Estas permitem uma simulação espacial da camada de mistura, de maneira condizente com o processo físico real.

O método de Runge-Kutta de sexta ordem foi escolhido para a solução numérica das derivadas pois garante que erros de precisão não impactem nos valores obtidos. No cálculo realizado para um determinado ponto discreto, o método leva em considerações os pontos vizinhos para a solução final.

3.2 O Algoritmo

O algoritmo é composto essencialmente por um número `N` de iterações pré-definidos pelo usuário, onde, em cada iteração, é executado o método Runge-Kutta de sexta ordem seguindo um intervalo de tempo `dt` também pré-definido. O fluxograma do algoritmo é apresentado na Figura 5. Na figura, pode-se observar que além da etapa iterativa, o algoritmo é composto por uma etapa de inicialização, composta de leitura de dados de entrada e alocação e inicialização de variáveis. Após a etapa iterativa são realizadas as desalocações de memória e finalização da aplicação.

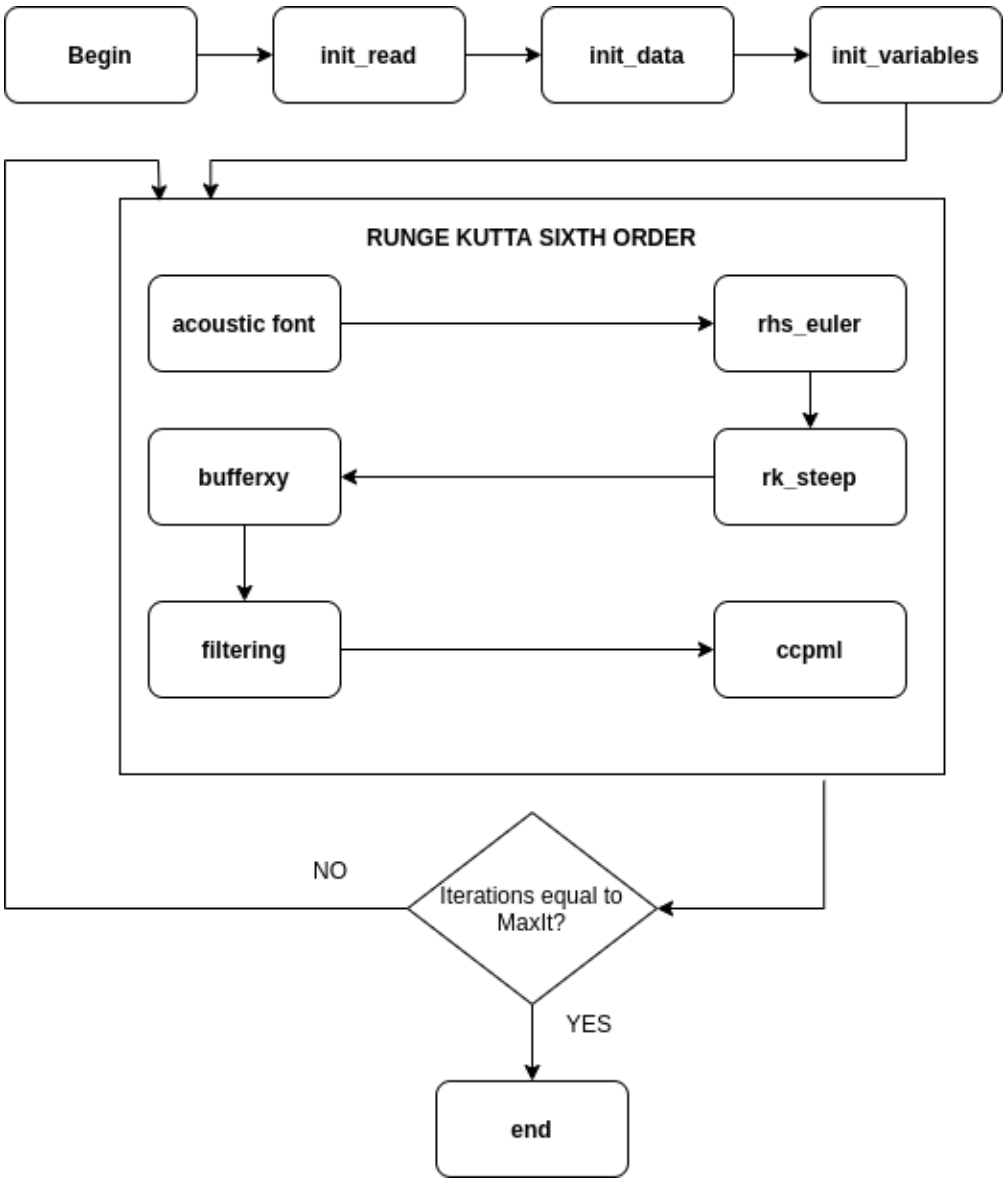
As operações realizadas dentro do laço principal que itera o passo de tempo `dt` são:

- `acousticfont`: Simula uma perturbação (turbulência), atuando como um pulso dentro da câmara.
- `rhs_euler`: Realiza todos os cálculos de derivada para todas as propriedades físicas de interesse.
- `bufferxy`: Faz o tratamento das condições de contorno não reflexivas.
- `rk_steep`: Concatena os passos do método de Runge-Kutta de sexta ordem.
- `filtering`: Trata o ruído numérico para cada ponto discreto, considerando 7 pontos vizinhos de cada lado. A Figura 6 apresenta os vórtices gerados para uma saída com e sem a aplicação do filtro (MANCO, 2014).
- `ccpml`: Faz o tratamento das condições de contorno da região de interesse.

3.3 Relação Esquemática entre as Rotinas do Código

A aplicação de simulação de câmera de mistura é formada por 13 arquivos Fortran 90, que totalizam quase 3.300 linhas de código. Boa parte destas linhas está concentrada em 53 sub-rotinas de código. A Figura 7 mostra a relação esquemática entre as chamadas das sub-rotinas. Nota-se que o código é extenso, o que exige um mapeamento preciso das funções e variáveis do código.

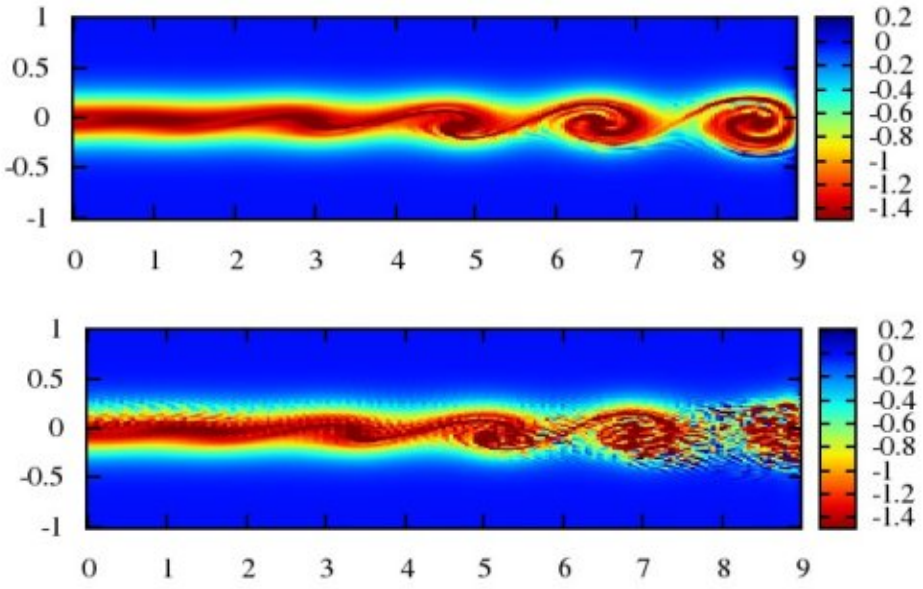
Figura 5 – Fluxograma



Fonte: O Autor.

Os arquivos que compõem o código são responsáveis pelo armazenamento e processamento das propriedades físicas simuladas. Por exemplo, no arquivo `global.f90`, estão contidas todas as estruturas de dados, incluindo as que armazenam os valores discretos de cada propriedade física, além de constantes, parâmetros e limites utilizados. Já o arquivo `init.f90` possui rotinas responsáveis pela leitura de um arquivo de configuração e pela alocação de memória das estruturas de dados, enquanto que o arquivo `outt.f90` possui rotinas que escrevem dados de saída, que posteriormente serão plotados usando gnuplot, através de scripts elaborados em Python. As demais rotinas são responsáveis por controlar o laço principal do código (`euler.f90`) e pelas sub-rotinas chamadas a partir deste laço.

Figura 6 – Vórtices gerados com e sem aplicação de filtro



Fonte: (MANCO, 2014)

3.4 Considerações Finais do Capítulo

Este capítulo apresentou a formulação matemática de uma aplicação de simulação de câmera de mistura, o algoritmo e a organização do código. Desta forma, são apresentados diferentes níveis de abstração, que envolvem a representação do problema discreto, até a sua representação computacional.

O código-fonte da aplicação é distribuído em diferentes arquivos e sub-rotinas. Embora a execução itere um grande laço iterativo, e cada iteração tenha uma sequência de rotinas a serem executadas, essas rotinas estão distribuídas em diferentes arquivos e níveis de chamada das rotinas. Desta forma, paralelizar a aplicação tende a ser mais difícil do que aplicações com menos linhas de código, menos arquivos, ou uma região clara e única que deve ser paralelizada.

Figura 7 – Relação Esquemática entre as funções

Fonte: O Autor.

4 PROGRAMAÇÃO PARALELA

A programação paralela consiste em explorar a concorrência da execução de código que as arquiteturas de computadores proveem. Isto é feito através do uso de interfaces de programação, que podem ser bibliotecas, *frameworks* ou APIs que possibilitam a criação de processos ou processos leves (threads) que podem ser executados em arquiteturas do tipo GPU, Multicore e/ou *Cluster* (LUCCA; SCHEPKE, 2020). Neste capítulo são apresentadas as arquiteturas e interfaces paralelas usadas no desenvolvimento do trabalho.

4.1 Arquiteturas MultiCore e GPU

O crescente desenvolvimento computacional previsto pela Lei de Moore (MOORE, 1965) possibilitou executar softwares cada vez mais rápidos, incluindo melhores funcionalidades e se tornando mais úteis para os usuários. Estes cada vez mais requeriam desta tecnologia, gerando uma fase positiva para a indústria de computadores, que se responsabilizava por manter o contínuo desenvolvimento, praticamente dobrando o número de transistores por processador a aproximadamente cada dois anos ao mesmo tempo em que mantinha os custos.

O fato é que essa tendência diminuiu a partir do ano de 2003 devido ao consumo elevado de energia e problemas com a dissipação de calor nos processadores (KIRK; HWU, 2010). A partir disso, surgiram as arquiteturas multicore e manycore. Na Figura 8 pode-se perceber os elementos que representam as diferenças entre as duas arquiteturas. À esquerda tem-se a representação de uma CPU multicore e à direita de uma GPU manycore.

Com isto pode-se perceber que cada arquitetura é especializada em um tipo de tarefa diferente, sendo os processadores multicore voltados tanto para a performance paralela quanto sequencial, dando ênfase para o desempenho individual das threads, porém limitado a um número menor de cores quando comparado com um processador manycore.

Figura 8 – Representação das arquiteturas multicore e manycore

Fonte: (CALISKAN, 2011)

Arquiteturas manycore são totalmente focadas e otimizadas para terem níveis altíssimos de paralelismo, tendo muitos *cores* mais simples e com menos performance individual. Uma arquitetura heterogênea é aquela que combina as duas arquiteturas, aproveitando o melhor das duas estratégias para alcançar máxima eficiência computacional.

Para alcançar este melhor desempenho oferecido pelas arquiteturas paralelas, é necessário expressar o paralelismo nos programas através da programação paralela. Este paradigma de programação permite ao programador especificar as áreas de código que devem ser executadas concorrentemente entre núcleos de um dispositivo paralelo, garantindo assim uma utilização mais eficiente do poder computacional proporcionado por estes dispositivos. Dada a importância do desenvolvimento de aplicações paralelas, atualmente existem diversas *Application Programming Interface* (APIs) voltadas para este fim. A maioria delas oferece suporte para a programação de GPUs ou foram projetadas para este fim, como é o caso das APIs CUDA, OpenACC e OpenCL. As próximas seções deste capítulo irão abordas algumas das diretivas e variáveis das APIs OpenACC e OpenMP (LUCCA; SCHEPKE, 2020).

4.2 OpenMP

OpenMP é uma API para programação paralela de memória compartilhada e multi plataforma disponível nas linguagens C/C++ e Fortran (OPENMP, 2022). A API é fundamentada no modelo de execução *fork-join*. Esse modelo possui uma thread mestre que inicia a execução e gera threads de trabalho para executar as tarefas em paralelo. OpenMP aplica o modelo de execução *fork-join* em segmentos do código que são informados pelo programador (ou usuário), exemplificado na Figura 9. Dessa forma um código sequencial é executado pela thread mestre até um bloco ou área de execução paralela.

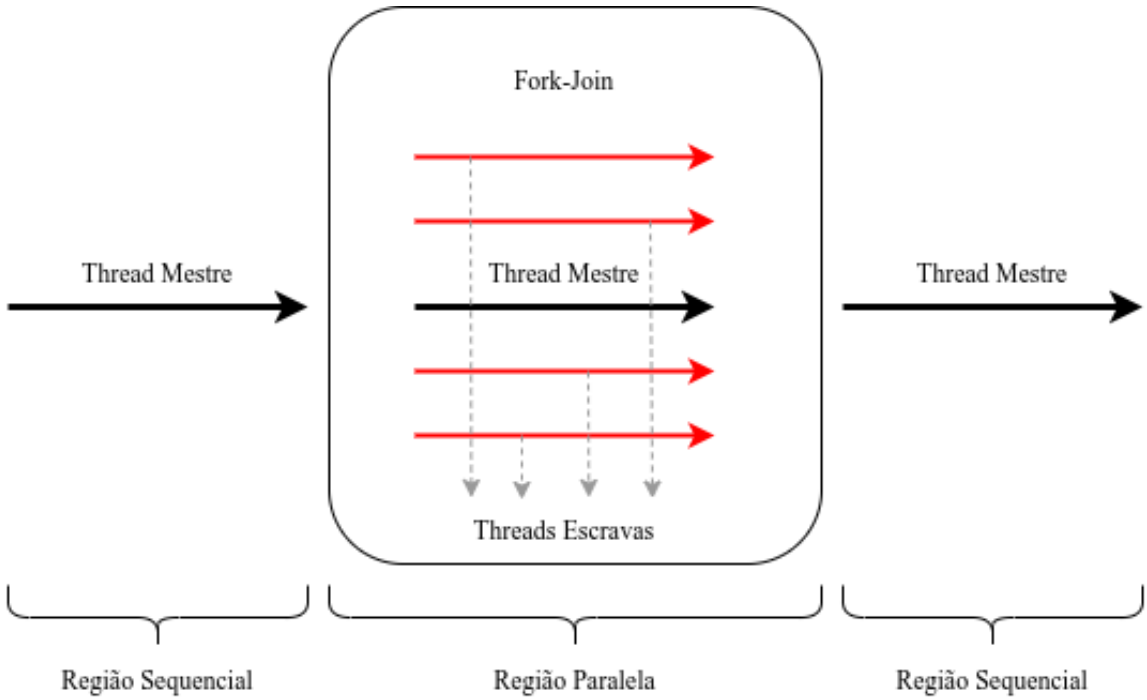
O inicio da área paralela é demarcado por uma diretiva OpenMP que é responsável por sinalizar que as threads de trabalho devem ser lançadas (*fork*). Todo o código seguinte é executado em paralelo pelas threads até o fim da área paralela, que pode ser demarcado explicitamente por um símbolo ou, de forma implícita, como por exemplo, em um laço de repetição *for*, onde o fim do laço de repetição também é o fim da área paralela. Este fim implica no encerramento das threads de trabalho, sincronização (*fork*) e retorno das threads a thread mestre para o prosseguimento da execução.

4.2.1 Diretivas e variáveis de ambientes da interface paralela OpenMP

A API OpenMP possui um conjunto de diretivas de compilação, funções e variáveis de ambiente para a programação paralela. Na linguagem Fortran utilizada, uma diretiva é precedida obrigatoriamente por `!$omp` e seguida por `[atributos]`, sendo que os atributos são opcionais. Seguem algumas diretivas que compõem a API OpenMP (OPENMP, 2022).

- `parallel:` Essa diretiva descreve que a uma área do código será executada por `N`

Figura 9 – Modelo *fork-join* em OpenMP



Fonte: O Autor.

threads, sendo `N` o número de threads especificados por um atributo e/ou variável de ambiente.

- `for`: Essa diretiva especifica que as iterações do laço de repetição serão executadas em paralelo por `N` threads.
- `parallel for`: 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.
- `for simd`: Essa diretiva especifica que um laço pode ser dividido em `N` threads que executam algumas iterações simultaneamente por unidades vetoriais.

Na sequência são apresentados alguns atributos da API OpenMP (OPENMP, 2022). Para todos os casos, `lista` representa uma ou mais variáveis.

- `private (lista)`: Esse atributo informa que o bloco paralelo possui variáveis privadas para cada uma das `N` threads. As variáveis do bloco que não são informadas na lista são consideradas públicas.
- `shared (lista)`: O atributo especifica que as variáveis são públicas e compartilhadas entre as `N` threads.

- `num_threads(int)`: Esse atributo determina o número `N` de threads utilizadas no bloco paralelo. O valor de `N` é válido apenas para o bloco em que foi definido.
- `reduction (operador: lista)`: A redução é utilizada para executar cálculos em paralelo. Cada thread tem seu valor parcial. Ao final da região paralela o valor final da variável é atualizado com os cálculos parciais. O operador pode ser, por exemplo, `+, -, *, max e min`.
- `nowait`: Uma diretiva OpenMP possui uma barreira implícita ao seu fim, com o objetivo de garantir a sincronização. Entretanto a diretiva `nowait` omite a existência dessa barreira. Dessa forma, as threads não ficam em espera até que as demais também terminem o trabalho.

As variáveis de ambiente do OpenMP especificam características que afetam as execução dos programas. Seguem algumas variáveis (OPENMP, 2022).

- `OMP_NUM_THREADS`: Especifica o número de `N` threads utilizados nos blocos paralelos do algoritmo.
- `OMP_THREAD_LIMIT`: Descreve o número máximo de threads.
- `OMP_NESTED`: Permite ativar ou desativas o paralelismo aninhado.
- `OMP_STACKSIZE`: Especifica o tamanho da pilha para as threads.

4.3 OpenACC

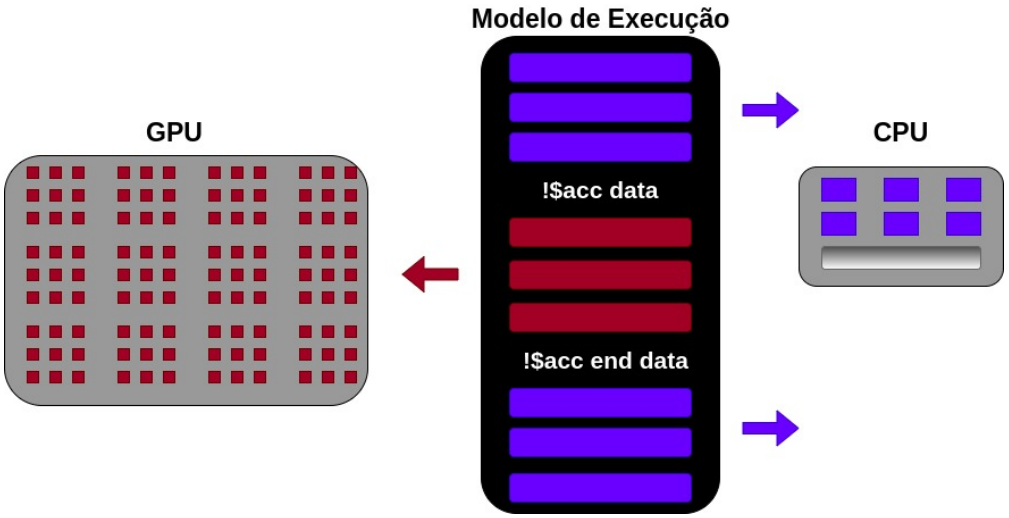
OpenACC é uma API que contém um conjunto de diretivas de compilação para GPUs, similar às fornecidas por OpenMP para CPU. Enquanto uma área paralela no OpenMP é executada por threads de trabalho localizadas na CPU, OpenACC especifica instruções que criam threads localizadas na GPU (OPENACC, 2022). Na Figura 10 observa-se o modelo de execução OpenACC (OPENACC, 2022).

4.3.1 Estrutura da GPU

OpenACC permite dividir as operações dentro do modelo *fork-join* entre as 3 estruturas bases da GPU: `Workers`, `Vector` e `Gang`. Dividir as tarefas entre essas estruturas permite realizar operações simultaneamente e consequentemente mais rápido. A fim de desmistificar a ideia das estruturas mencionadas, segue uma imagem ilustrativa na Figura 11.

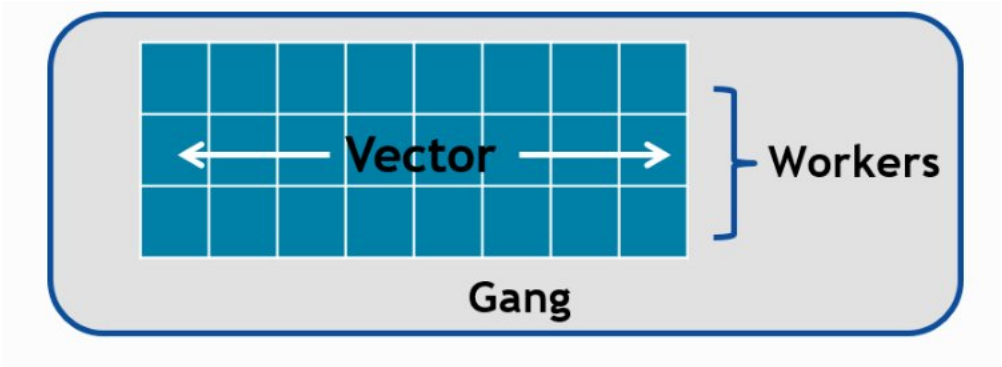
- `vector`: Realiza uma dada operação: as threads trabalham em sincronia (SIMD / SIMT paralelismo).

Figura 10 – OpenACC modelo de execução



Fonte: O Autor.

Figura 11 – Estrutura GPU



Fonte: (OPENACC, 2022)

- `worker`: Possui um ou mais `vectors`.
- `gang`: Contém um ou vários `workers` e divide seus recursos. Múltiplos `gangs` trabalham independentemente um do outro. Dessa forma, um `gang` não compartilha recurso com outros `gangs`.

4.3.2 Diretivas e variáveis de ambientes da interface paralela OpenACC

Uma diretiva OpenACC em Fortran possui o seguinte formato (OPENACC, 2022): `!$acc diretiva [clausulas [,] clausulas]`. Seguem algumas diretivas que compõem a API OpenACC (OPENACC, 2022).

- `parallel:` Essa diretiva descreve um bloco em paralelo executado por `N` threads disponíveis.
- `kernels:` Especifica que o compilador dividirá o bloco paralelo em *kernels*.

- `loop`: Paraleliza o(s) *loop(s)* aninhados imediatamente após a diretiva.
- `serial`: Essa diretiva especifica um bloco de código que será executado sequencialmente.
- `data`: Define um bloco no qual os dados são acessíveis pela GPU.

Seguem algumas cláusulas que compõem a API OpenACC em relação a movimentação dos dados (OPENACC, 2022). Para todos os casos, `lista` representa uma ou mais variáveis.

- `copy (lista)`: Essa cláusula copia os dados do *host* para a GPU e da GPU para o *host*.
- `copyin (lista)`: Essa cláusula copia os dados do *host* para a GPU.
- `copyout (lista)`: Essa cláusula copia os dados da GPU para o *host*.
- `present (lista)`: Essa cláusula informa que os dados utilizados no bloco paralelo já foram copiados para a memória da GPU.

A API OpenACC possui variáveis de ambiente que especificam alterações na execução dos programas. Para as ocorrências, `type` representa o tipo do dispositivo e `size` o tamanho de memória que será alocado. Seguem algumas variáveis (OPENACC, 2022).

- `acc_get_num_devices (type)`: Retorna o número de dispositivos do tipo especificado.
- `acc_set_device_type (type)`: Define o tipo de dispositivo utilizado.
- `acc_get_device_type ()`: Retorna o tipo de dispositivo que está sendo usado pela thread.
- `acc_wait_all ()`: Aguarda até que todas as atividades assíncronas sejam concluídas.
- `acc_malloc(size)`: Retorna o endereço da memória alocada no dispositivo.
- `acc_is_present`: Testa se os dados do *host* já foram copiados para a GPU.

4.4 Considerações do Capítulo

Neste capítulo foram descritas as interfaces de programação OpenMP e OpenACC. Tais interfaces permitem a programação em arquiteturas multicore e manycore e foram utilizadas para a exploração da concorrência da aplicação de simulação da câmera de

mistura. As notações aqui utilizadas seguem o padrão FORTRAN, embora a grande maioria dos recursos são exatamente os mesmas em C/C++.

OpenMP é o padrão de fato para a programação paralela em Multicore. Além disso, as especificações mais recentes da linguagem permitem gerar paralelismo para GPU também. Já OpenACC é uma interface bastante similar a OpenMP, o que é bastante positivo. OpenACC, comparado com a interface de programação CUDA, exige menos linhas e alterações no código, produzindo um código compilado bastante similar em termos de eficiência.

5 ASPECTOS METODOLÓGICOS

Nesse capítulo são definidos alguns aspectos metodológicos que conduziram o andamento do trabalho.

5.1 Ciclo de Otimização

Um ciclo de otimização foi definido para seguir em cada um dos testes realizados, baseado em três principais pilares: *Observar*, *Otimizar* e *Avaliar*.

- *Observar:* Tarefas como mapeamento de variáveis e funções, verificação de trechos paralelizáveis e planejamento para o próximo pilar eram definidos.
- *Otimizar:* Processo de otimização era realizado, tais como paralelizar trechos de código, exclusão de partes desnecessárias, modificação de funções e movimentação de blocos de código.
- *Avaliar:* O código é então compilado e executado. Caso nenhum erro é constatado, ainda sim, a saída da versão serial original do código é verificado e comparado com o atual, em processo de otimização utilizando o comando `diff` ou `vimdiff`. É obrigatório que as duas saídas não tenham nenhuma alteração numérica. O código deve também ser executado em menos tempo que a versão antecessora. Caso contrário, a otimização teve um impacto negativo no desempenho.

5.2 Parâmetros de Entrada

Todos os parâmetros de entrada do algoritmo são flexíveis. Porém, é necessário um cuidado extra ao realizar quaisquer modificações nos mesmos, visto que afetam a sensibilidade do código. Qualquer alteração equivocada nos parâmetros pode gerar erros de inconsistência entre as saídas dos experimentos em relação ao fenômeno físico real.

Além disso, erros de precisão numérica podem gerar distorções nos resultados. Isso se deve pois em cada passo de tempo (iteração), o cálculo da derivada da forma discretizada carrega um erro. Quanto mais pontos são considerados na malha, mais combinações de erro serão adicionados. Este erro pode se propagar e impedir que o programa chegue a uma solução final válida (geração de valores do tipo *Not a Number - NaN*).

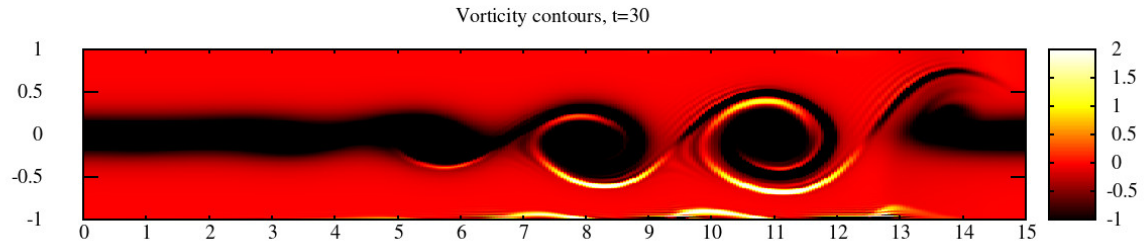
Os parâmetros de execução são definidos no arquivo `dados.in`. Para os testes realizados, utilizou-se duas entradas, basicamente alterando o tamanho da malha da câmara de mistura. A malha padrão é representada pela *Malha I* e uma segunda malha maior do que a citada anteriormente é representada pela *Malha II*, ambas apresentadas na Tabela 2. A Malha II tem aproximadamente o dobro de pontos, sendo mais precisa e consequentemente, mais custosa em termos de processamento.

Tabela 2 – *Malha I* e *Malha II*

Descrição	**Variável**	**Valor *Malha I***	**Valor *Malha II***
Dimensão x da Câmara	imax	461	921
Dimensão y da Câmara	jmax	381	761
Avanço temporal	dt	0.01	0.005
Número de Iterações	maxit	100	200

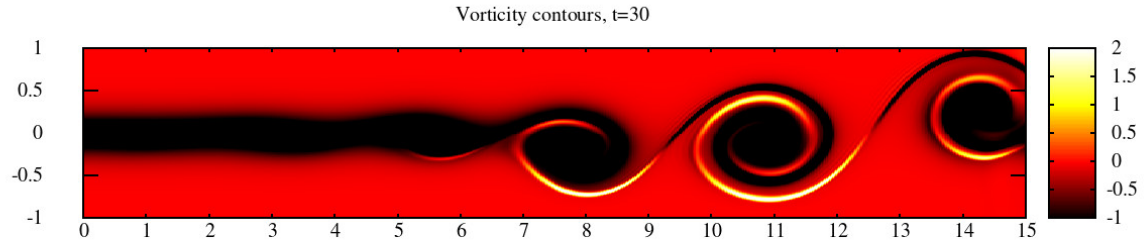
Fonte: O Autor.

Figura 12 – Vorticidade *Malha I*



Fonte: O Autor.

Figura 13 – Vorticidade *Malha II*



Fonte: O Autor.

A Figura 12 e Figura 13 apresentam a saída da execução e cálculo da vorticidade para a *Malha I* e *Malha II*, respectivamente. É nítido ver a melhoria ao utilizar a *Malha II*, que possui mais pontos.

5.2.1 Número de iterações

Em um experimento é necessário representar aproximadamente pelo menos 30 segundos de transcorrer de tempo da simulação para que se torne visível a geração dos vórtices dentro da câmara de mistura. Estes vórtices são causados pela perturbação que

Tabela 3 – Custo por iteração variando número total de Iterações para versão sequencial utilizando à *Malha I* e *Malha II*

Número de iterações	**Custo por Iteração *Malha I***	**Custo por Iteração *Malha II***
100	0,8862s	5,5914s
200	0,8912s	5,5861s
400	0,8891s	5,5599s
800	0,8897s	5,5378s

Fonte: O Autor.

a rotina `acoustic font`, mencionada anteriormente no Capítulo 3 realiza. A geração dos vórtices é utilizada para avaliar a convergência e corretude dos valores.

Para representar 30 segundos de execução são necessárias 3000 iterações para a *malha I*, visto que o passo de tempo (`dt`) definido para a mesma é igual a $0,01$. Ou seja, $3000 \times 0,01$ é equivalente a 30.

Já para a *malha II* são necessárias 6000 iterações, levando em consideração que a malha é maior e mais precisa, sendo necessário um passo de tempo (`dt`) menor, definido em 0,005. Com isso, ao realizar-se o cálculo $6000 \times 0,005$, obtém-se o valor 30.

Para os testes de desempenho constantes neste trabalho não se torna necessário manter o número de iterações tão alto, visto que para representar 30 segundos da câmara é necessário aproximadamente 2 horas de execução. Além disso, observou-se que o tempo médio de cada iteração é computacionalmente (número de instruções/operações) e na prática o mesmo. A Tabela 3 apresenta testes distintos retirando uma média de 30 experimentos e variando o número total de iterações em 100, 200, 400 e 800, visando medir o custo por iteração em cada teste, comprovando que independente do número de iterações, o custo por cada uma é (número de instruções/operações) aproximadamente o mesmo.

Como o custo médio de cada iteração é praticamente igual, foi definido o número de iterações de 100 para à *malha I* e de 200 para a *malha II*. Desta forma, ambas irão representar apenas 1 segundo de simulação da câmara de mistura.

Para fins de testes de desempenho e agilizar os experimentos não é necessário utilizar um número de iterações maior, visto que a carga de computação não muda. Tal simplificação possibilitou um grande número de testes, tornando possível avaliar experimentalmente o paralelismo da aplicação e o ganho/perda de desempenho em cada pequena alteração realizada no código.

5.3 Métricas Avaliadas

Para medir o desempenho das versões dos algoritmos paralelos foi utilizado o conceito de speedup (S). O speedup (S) é definido como a razão entre o tempo de computação do algoritmo serial (T_{serial}) e o tempo de computação do algoritmo paralelo

Figura 14 – Arquivo `euler.f90` dividido em blocos de acordo com suas funções

Fonte: O Autor.

($T_{paralelo}$) (FOSTER, 1995).

Para encontrar a eficácia de cada thread foi utilizado o cálculo da eficiência, que é definido como a razão entre o speedup (S) e o número de threads daquele dado experimento.

É importante considerar, para efeitos de resultado, a média e o desvio padrão. Assim, definiu-se um número de 30 experimentos para cada caso testado.

5.3.1 Pré-processamento, Pós-processamento e Etapa Iterativa

A aplicação principal do algoritmo se encontra no arquivo `euler.f90` que atua como a *main* do código. O mesmo está dividido em três blocos de acordo com suas funções conforme a Figura 14.

No pré-processamento, o tempo gasto do algoritmo é somente o de inicialização, que inclui a leitura de arquivos de configurações, inicialização de variáveis e alocação de memória. No bloco iterativo são realizadas as iterações sobre a câmara de mistura, e, de fato, o cálculo das equações de Navier-Stokes, além das escritas de valores temporários de monitoramento parcial nos arquivos de saída. Já no pós-processamento, é liberado a memória alocada para o experimento e o algoritmo é então finalizado.

5.3.2 Avaliando o Custo por Iteração

Medir o tempo de execução de cada um dos blocos provê a capacidade de desconsiderar os tempos de processamento que não são de interesse e, assim, avaliar especificamente a parte iterativa da execução. Para avaliar o custo por cada iteração, utilizou-se a fórmula da Equação 5.1, onde `M_Custo_Total` representa a média de 30 execuções do tempo total de execução, com o número de iterações definidas. Já `M_Custo_Processamento` representa a média de 30 execuções somente do tempo de processamento (*Pré-Processamento*+

Tabela 4 – Principais rotinas que demandam tempo de sequencial de processamento

Rotina	**Arquivo**	**Porcentagem de tempo da execução total**
dudy	diff.f90	24,37%
lddfiltery	filltering.f90	12,65%
dudx	diff.f90	12,61%
lddfilterx	filltering.f90	12,56%
rk_euler	filltering.f90	7,65%

Fonte: O Autor.

Pós-Processamento). Dessa forma, é desconsiderado o tempo de inicialização e término na fórmula.

Em seguida, na fórmula o valor é dividido por `N` que representa o número de iterações. No nosso caso, este valor é igual a 100 para a malha menor e 200 para a malha maior. Com esse método, é encontrado o custo exato de tempo que cada iteração possui.

$$Custo_{\text{Iteração}} = \frac{M_{Custo\_Total} - M_{Custo\_Processamento}}{N} \tag{5.1}$$

5.4 Perfilamento das Aplicações

A versão sequencial original do código implementado em FORTRAN foi inicialmente avaliada. Foram feitos testes para mensurar o custo de processamento (leitura de arquivo de entrada, alocação de memória e preenchimento de estruturas de dados) e o tempo da etapa iterativa da execução da aplicação. Desta forma, obteve-se o tempo total sequencial da execução da aplicação.

Na sequência, o código foi então avaliado com a ferramenta gprof (GRAHAM; KESSLER; MCKUSICK, 1982), que faz coletas estatísticas do tempo de execução demandado por cada rotina que compõe o código.

Na Tabela 4 observa-se as rotinas que mais demandam tempo de execução.

Com base nos resultados de *profile* obtidos foram feitas inspeções nas rotinas, identificando-se laços aninhados nas rotinas. A *flag* `-Minfo=all` foi utilizada em algumas execuções específicas para auxiliar na identificação desses blocos paralelizáveis, esta diretiva está contida nas duas APIs. Ao compilar com esta *flag*, o compilador provê um *log* ao programador com possíveis soluções de paralelizações.

Em um segundo momento foram feitas inspeções em outras rotinas com menor porcentagem do tempo total de execução (menor que 5%, de acordo com o GProf), e que previamente não foram investigadas. Essas rotinas também foram paralelizadas utilizando diretivas. Elas também contribuíram para a redução do tempo total de execução da aplicação. Isso foi possível observar através de avaliações de desempenho parciais, a medida que o código foi alterado ou paralelizado. Algumas dessas rotinas encontram-se na etapa de inicialização do código.

Tabela 5 – Ambiente CPU

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

Tabela 6 – Ambiente GPU

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

Fonte: O Autor.

5.5 Ambiente de Validação

Todos os testes foram realizados em uma *workstation* da Universidade Federal do Pampa (UNIPAMPA), campus Alegrete, que possui a configuração apresentada Na Tabela 5 e Tabela 6. Como sistema operacional foi utilizado Ubuntu na versão 20.04 de 64 bits e *kernel* 5.11.0-27-generic.

Os testes foram feitos considerando o compilador `pgf90` versão 20.9 do pacote HPC_SDK da NVIDIA. Foram utilizadas as diretivas `-O3 -mp` para os experimentos com a API OpenMP e as diretivas `-O3 -acc` para os experimentos com a API OpenACC. Outros testes também foram previamente feitos com o compilador `gfortran`, com diretivas similares. No entanto, o tempo de execução sequencial e paralelo foram maiores para o código gerado por este compilador.

Vale salientar que para os testes finais, que representam os resultados no Capítulo 6 e Capítulo 7, o ambiente foi reservado somente para os experimentos, a fim de evitar quaisquer interferências de outros processos/usuários, e consequentemente, imprecisões nos resultados.

6 RESULTADOS EXPERIMENTAIS COM A VERSÃO OPENMP

Este capítulo apresenta os resultados de performance obtidos para as implementações paralelas geradas com a interface de programação paralela OpenMP utilizando a *Malha I* e a *Malha II*.

6.1 Custo por Iteração

A Tabela 7 apresenta a média dos valores de cada uma das 30 execuções para o custo por iteração da *Malha I* e da *Malha II* em segundos. Para cada um dos experimentos, foram medidos os tempos usando 2, 4, 8, 16 e 32 threads.

Ao se utilizar duas threads, nota-se que o custo por iteração foi reduzido consideravelmente com as duas entradas. Utilizando-se mais threads o tempo do custo por iteração continuou sendo reduzido expressivamente com exceção do *hyper-threading* ao utilizar 32 threads. Como a aplicação é *CPU-bound*, utiliza-se praticamente 100% do processamento da CPU, na etapa iterativa, independente do número de threads adotadas, conforme monitoramento feito durante a execução do processamento da etapa iterativa, conforme a Tabela 8.

O *Hyper-Threading* (uso de 32 threads no nosso caso) não surtiu uma redução de tempo mais expressivo para nenhuma das entradas. Essa tecnologia permite que cada núcleo de um processador possa executar duas threads de uma única vez (ÉTIENNE, 2012). A resposta para isso é que o custo das operações (trabalho) que são realizadas dentro das threads é de fato pouco custoso, e o que gera o ganho de desempenho é executar essas pequenas tarefas em paralelo (simultaneamente). Com 16 threads a solução do problema na atual implementação já se aproxima de um ápice de paralelismo e ganho de desempenho, visto que existem trechos que não podem ser paralelizáveis. Esta afirmação é abordada de forma mais completa no Capítulo 8

6.2 SpeedUp (S)

O speedup (S) é apresentado na Figura 15. O melhor resultado para ambas as entradas foi obtido ao utilizar 16 threads. Assim, obteve-se o speedup (S) de aproxima-

Tabela 7 – Custo por Iteração *Malha I* e *Malha II*

Modo de Execução	**Custo por Iteração *Malha I***	**Custo por Iteração *Malha II***
Sequencial	0,898s	5,586s
2 Threads	0,540s	3,013s
4 Threads	0,437s	2,014s
8 Threads	0,337s	1,414s
16 Threads	0,225s	1,133s
32 Threads	0,244s	1,158s

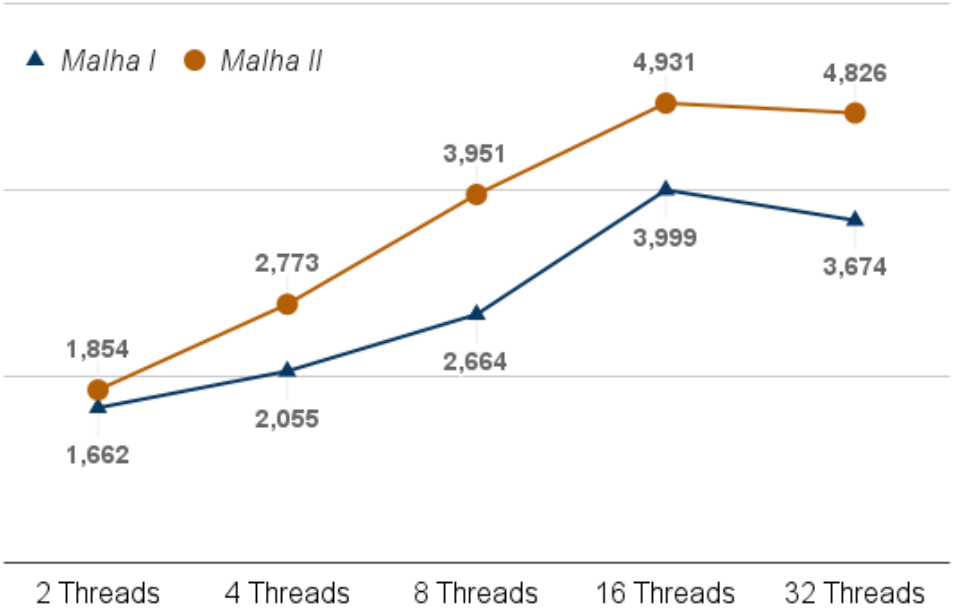
Fonte: O Autor.

Tabela 8 – Utilização da CPU com e sem Hyper-Threading

Número de Threads	**Comando *top***
16 Threads	1556 %
32 Threads	3075 %

Fonte: O Autor.

Figura 15 – SpeedUp (S) *Malha I* e *Malha II*



Fonte: O Autor.

damente 4 para à *Malha I* e de aproximadamente 5 para a *Malha II*, para o custo por cada iteração.

Ao comparar as duas entradas, a *Malha II* em relação a *Malha I* obteve melhores resultados para todos os casos avaliados. É viável afirmar que quanto mais precisa é a malha utilizada, maior é o nível de paralelismo e ganho de desempenho para a versão paralela implementada.

6.3 Tempo Total de Execução

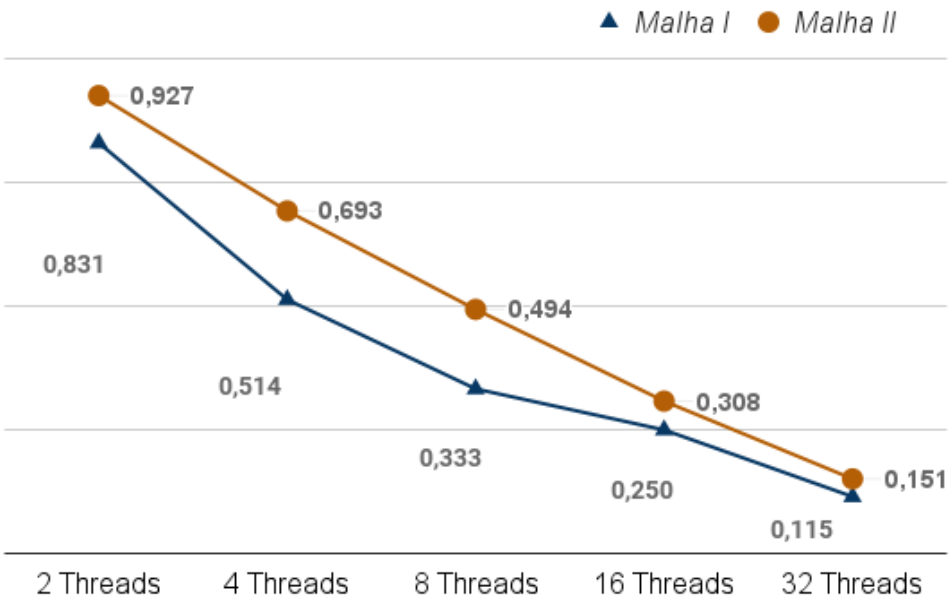
A Tabela 9 apresenta a média de 30 execuções do tempo total de execução de cada simulação usando 100 iterações do algoritmo para à *Malha I* e 200 iterações do algoritmo para a *Malha II*. Este número de iterações representa 1 segundo de simulação da câmara de mistura em ambos os casos. O número de threads avaliado foi de de 2, 4, 8, 16 e 32.

Tabela 9 – Tempo Total etapa Iterativa, versão OpenMP

Modo Execução	**Tempo Total (Segundos)** ***Malha I***	**Tempo Total (Segundos)** ***Malha II***
Sequencial	89,787s	1117,227s
2 Threads	54,023s	602,687s
4 Threads	43,686s	402,871s
8 Threads	33,702s	282,748s
16 Threads	22,452s	226,573s
32 Threads	24,441s	231,509s

Fonte: O Autor.

Figura 16 – Eficiência de cada thread, versão OpenMP



Fonte: O Autor.

6.4 Eficiência

O cálculo da eficiência é baseado na razão entre o speedup (S) e o número de threads utilizadas no experimento. Desta forma, pode-se saber o quão impactante cada thread contribuiu para o experimento, conforme apresentado na Figura 16.

Como o speedup(S) para o caso da *Malha II* foi maior, consequentemente a eficiência também foi mais elevada.

6.5 Tempo Total de Simulação para um experimento representando 30 segundos dentro da Câmara de Mistura

Como os dados necessários para calcular o tempo total de um experimento completo é conhecido, isto é, o número de iterações, é possível inferir o tempo total de execução da aplicação em cada caso.

Tabela 10 – Tempo Total para um experimento real, versão OpenMP

Modo Execução	**Tempo Total (Segundos)** (3000 **iterações) *Malha I***	**Tempo Total (Segundos)** (6000 **iterações *Malha II*)**
Sequencial	2694s	33517s
2 Threads	1621s	18081s
4 Threads	1311s	12086s
8 Threads	1011s	8482s
16 Threads	674s	6797s
32 Threads	733ss	6945s

Fonte: O Autor.

Assim, considerando as 3000 iterações para o caso da *Malha I* e 6000 para o caso da *Malha II* pode-se calcular o tempo de processamento que representam 30 segundos de simulação da câmara de mistura. Com isso, e baseado na média do custo por cada iteração para as duas entradas, torna-se viável mensurar a redução de tempo para um experimento real.

A redução de tempo é expressiva para o melhor caso de OpenMP com à *Malha I* e com a *Malha II*. Para o primeiro caso, reduz-se o tempo de processamento de aproximadamente 45 minutos da versão sequencial original, para aproximadamente 11 minutos para o melhor caso ao se utilizar 16 threads. Já para a *Malha II* os resultados foram ainda mais impactantes. Para o melhor caso, o tempo foi reduzido de aproximadamente 9 horas e 18 minutos da versão sequencial original, para aproximadamente 1 hora 50 minutos. Observa-se que a Tabela 10 apresenta o tempo gasto para cada caso de um experimento real.

7 RESULTADOS EXPERIMENTAIS COM A VERSÃO OPENACC

Este capítulo apresenta os desafios encontrados na implementação da API OpenACC da aplicação.

Como mencionado anteriormente, o código é extenso e, de certa forma, o que dificulta o estudo das variáveis com dependência para realizar a troca entre *host* e GPU, que são necessárias nos blocos paralelizáveis (*fork-join*), como mencionados anteriormente na Seção 4.3.1.

Algumas versões em OpenACC foram descontinuadas e desconsideradas para este trabalho pois não cumpriam com o pilar de verificação do ciclo de otimização definido na Seção 5.1. Ao final da execução era executado o comando `diff` ou `vimdiff` para comparar a saída sequencial original com a versão em GPU, a fim de comprovar se elas eram exatamente iguais. Como elas não eram 100% fiéis os testes eram desconsiderados.

Porém essas imprecisões numéricas são insignificantes visto que as mesmas acontecem somente em casas decimais muito grandes, por exemplo: uma mesma cédula da saída sequencial original apresenta o valor $-3.0699766169811789E-004$ enquanto está mesma cédula de um mesmo experimento com a versão em OpenACC apresenta o valor $-3.0699766172447902E-004$. A imprecisão nesta cédula é na casa decimal -9. Outro exemplo é o valor em outra cédula na versão sequencial que apresenta o valor de $-4.0451172400611840E-006$ e na versão OpenACC $-4.0451172400611552E-006$ onde a imprecisão numérica é na casa decimal -14. Estas pequenas imprecisões podem acontecer por alguns fatores como:

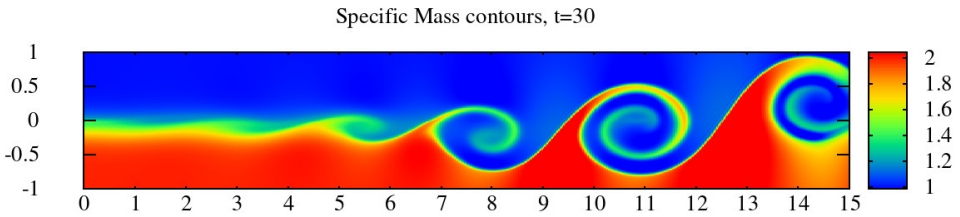
- Realizar os cálculos em ordem diferente da versão sequencial.
- Movimentação entre *host* e GPU e como a API e o compilador interpreta e lê o ponto flutuante nessas movimentações.
- Os valores *float* de cópia entre *host* e GPU podem não ser binariamente os mesmos.

Para confirmar que esses erros numéricos podem ser desconsiderados, foi executado um experimento com a *Malha II* com 6000 iterações para uma versão OpenACC e a versão sequencial original. Observa-se nas Figuras 17 e 18 no instante de 30 segundos da execução da aplicação.

Ambas as figuras geradas de cada versão são idênticas mesmo apresentando as pequenas imprecisões de valores. Nota-se que também foi comparado saídas para outras propriedades físicas como vorticidade, velocidade em `x` e em `y` e pressão.

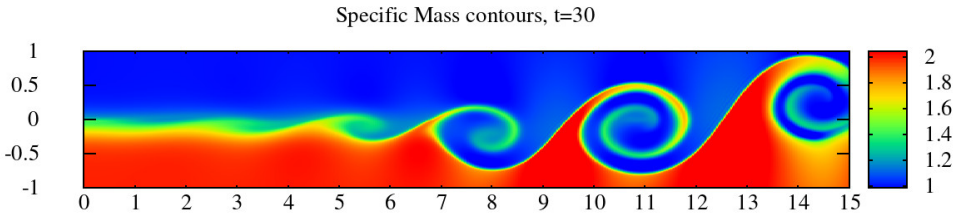
Mesmo com esses dados, é necessário realizar e avaliar mais experimentos para confirmar que ao invés de comparar os arquivos `.dat` da saídas das aplicações de cada versão, seja possível comparar as imagens de saída das execuções. Para isso seria necessário alterar o pilar de verificação do ciclo de otimização definido na Seção 5.1.

Figura 17 – Massa específica no instante de 30 segundos versão OpenACC



Fonte: O Autor.

Figura 18 – Massa específica no instante de 30 segundos versão Sequencial Original



Fonte: O Autor.

8 CONSIDERAÇÕES FINAIS

As simulações são importantes em várias áreas de conhecimento, como na meteorologia, prevendo situações climáticas, e na engenharia, na criação de projetos. Essas simulações auxiliam na prevenção de erros, falhas e, principalmente, na parte econômica de um dado projeto, visto que não é necessário a criação de um ambiente real para realizar os experimentos.

A importância de se realizar otimizações em aplicações de simulações é notável, o que provê a capacidade de realizar um número considerável a mais de testes em um intervalo de tempo menor. Em nosso trabalho aceleramos uma aplicação de simulação de câmera de mistura. A versão em OpenMP obteve um ganho expressivo para todos os experimentos do estudo de caso, obtendo um menor tempo de execução para as duas malhas avaliadas. Ao se utilizar 16 threads, alcançou-se o speedup foi de 4,9 para a *Malha II* e 4 para a *Malha I*. Esse speedup significa uma redução muito expressiva em cada iteração da aplicação, o que possibilita diminuir consideravelmente o tempo de execução para cada um dos casos de teste. No caso de teste *Malha II* foi economizado um tempo de aproximadamente 7 horas e 28 minutos com a implementação paralela em OpenMP. No caso de teste *Malha I*, uma redução de tempo de aproximadamente 34 minutos foi alcançada. A Figura 19 representa o tempo gasto para cada uma das malhas utilizadas como caso de uso para a versão sequencial e a versão OpenMP que obteve melhor resultado, ou seja, utilizando 16 threads.

Comparado com os trabalhos relacionados no Capítulo 2, o speedup de 4 e 4,9 é considerado pequeno, porém sabe-se que o custo das operações (computação) que são realizadas dentro das threads é baixo. Mas vale salientar que cada aplicação é um caso diferente e cada uma possui seus limites de paralelização e ápice de ganho de desempenho. O que gera de fato uma otimização expressiva em uma aplicação é executar tarefas com

Figura 19 – Tempo gasto para cada caso de teste (malhas) com a versão sequencial e OpenMP com 16 threads

Fonte: O Autor.

Tabela 11 – Rotinas mais custosas com a versão OpenMP

Rotina	**Arquivo**	**Porcentagem de tempo da execução total**
output	outt.f90	69,18%
initialize	init.f90	19,03%
deropm	diff.f90	4,28%
rk_euler	rk.f90	3,54%
lddfilterx	filltering.f90	2,86%

Fonte: O Autor.

uma boa carga de computação em paralelo. Por isso que se obteve um ganho de desempenho superior para a *Malha II*, visto que quanto maior for malha, mais pontos existem dentro das dimensões da câmara e, consequentemente, maior o nível de paralelismo.

O ganho ao aumentar o número de threads não se mantém linear com um ganho constante ao utilizar 32 threads (*hyper-threading*) devido ao fato da solução chegar cada vez mais próxima do ápice de paralelismo. Isto é comprovado, pois as rotinas que demandavam mais tempo de execução na versão sequencial (de acordo com a seção 5.4), após a implementação com a versão OpenMP foram reduzidas drasticamente, e as mais custosas na versão paralela são funções não paralelizáveis. Observa-se na Tabela 11 as funções que demandam mais porcentagem do tempo total de execução na versão OpenMP.

A versão em OpenACC não obteve resultados satisfatórios, visto que a saída do programa original e sequencial não foi 100% fiel com a versão em GPU apresentando pequenas imprecisões. A transição das variáveis entre *host* e GPU ocasionam muitas dependências de dados, o que é provavelmente a causa desta incompatibilidade com os resultados finais. Mesmo com a relação esquemática realiza no inicio do projeto, fica difícil prever e localizar estas dependências em um código com inúmeras linhas. Também é necessário estudar e avaliar como o compilador e a API realizam a leitura deses valores em ponto flutuante binariamente, visto que o erro pode não ser causado pelo programador e sim pelo fato de uma leitura diferente entre *host* (compilador) e GPU (API) nas cópias e movimentações de variáveis.

Neste trabalho e experimentos realizados seguiu-se um passo a passo padrão que pode ser utilizado em outras aplicações similares. É necessário entender o código a ser paralelizado/otimizado. Para isso, de primeira mão, é interessante navegar pelo código e executá-lo algumas vezes observando-o.

Em seguida a organização do projeto deve ser criada, ou seja, um mapeamento de funções e variáveis. Isso vai ser muito importante para os próximos passos. Para criar esta relação esquemática utilizou-se o software Drawio (ALDER, 2021). Após esta relação ter sido criada, fica fácil entender a ordem de chamadas e execução da aplicação, facilitando a identificação de trechos descontinuados e funções que não estão sendo utilizadas.

Neste trabalho a implementação começou de fato pelas rotinas mais custosas de acordo com a execução da ferramenta gprof (GRAHAM; KESSLER; MCKUSICK, 1982),

mas uma segunda forma também muito eficaz é encontrar a `main` da aplicação e ir avançando com a implementação da otimização de acordo com a ordem da aplicação.

A implementação deve ser realizada em cada bloco de código seguindo o conceito de *fork-join* das APIs. Desta forma, a cada pequena alteração a mesma deve ser verificada na relação esquemática criada anteriormente a fim de evitar e corrigir erros devido a dependências de dados. Também é necessário a cada alteração no código realizar uma comparação entre a saída da aplicação entre a versão original sequencial e a atual em processo de paralelização/otimização, a fim de conferir se a saída de ambas são de fato, iguais.

Este trabalho em diferentes versões e experimentos rendeu publicações em alguns eventos nacionais e internacionais, segue as publicações:

- Publicação na XXI Escola Regional de Alto Desempenho da Região Sul (ERAD 2021) (PIZZOLATO; SCHEPKE, 2021).
- Publicação no Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD 2021) (PIZZOLATO; SCHEPKE; LUCCA, 2021).
- Publicação no Salão Internacional de Ensino, Pesquisa e Extensão (2020) (PIZZOLATO; SCHEPKE, 2020).
- Publicação no Salão Internacional de Ensino, Pesquisa e Extensão (2021) (PIZZOLATO; SCHEPKE, 2021).
- (**Aprovado**) Publicação na XXII Escola Regional de Alto Desempenho da Região Sul (ERAD 2022).
- (**Em andamento**) Publicação na revista "Concurrency and Computation".

8.1 Trabalhos Futuros

A aplicação possui um número significativo de linhas de código e tem grandes possibilidades de trabalhos futuros que podem dar continuidade a atividade de aceleração da aplicação.

O mais importante seria realizar uma refatoração no código, adicionando padrões para nomenclaturas e localidade das variáveis, além de fazer um projeto de software organizado e bem elaborado, a fim de deixar o código mais simples e legível, de preferência reescrevendo-o para uma linguagem mais popular, como C ou C++ que são compatíveis com a API OpenACC e OpenMP.

Um segundo passo seria uma implementação em OpenACC mais complexa e detalhada, que seja fiel a versão original quanto a saída da execução e retorne uma paralelização mais expressiva, tendo ganho de tempo que supere a versão em OpenMP em CPU. Visto

que a solução atual em OpenMP já se aproxima do ápice de ganho de desempenho com paralelismo, seria importante também a necessidade de aderir uma malha ainda mais precisa do que os dois casos de usos utilizados neste experimento.

Esta aplicação possui uma versão descontinuada em MPI. Também fica em aberto a reformulação desta implementação de paralelização de forma distribuída. Uma arquitetura heterogênea também pode ser proposta, misturando duas APIs, aproveitando dessa forma o máximo da CPU e o máximo da GPU, podendo alcançar uma solução perfeita para esta aplicação.

Com a atual versão desta aplicação, torna-se possível testar e realizar experimentos em mais casos de testes, com diferentes entradas de fluidos e malhas, aumentando a diversidade de problemas que podem ser resolvidos com esta aplicação.

Para prover uma usabilidade e melhor praticidade nesta aplicação e controle dos parâmetros de entrada, seria interessante construir uma interface gráfica. Esta auxiliaria os usuários quanto aos valores de entrada, número de iterações, passo no tempo, fluídos, entre outras variáveis que são necessárias e sensíveis neste programa.

REFERÊNCIAS

ALDER, G. drawio. In: JGRAPH LTD. 2021. [Online; acessado em janeiro, 26 2021]. Disponível em: <app.diagrams.net>. Citado na página 54.

ALDINUCCI, M. et al. Practical parallelization of scientific applications with openmp, openacc and mpi. **Journal of Parallel and Distributed Computing**, v. 157, 06 2021. Citado 2 vezes nas páginas 21 e 22.

CALISKAN, K. Gpu opencl vs cuda vs arbb. In: . [s.n.], 2011. Disponível em: <http://www.keremcaliskan.com/gpgpu-opencl-vs-cuda-vs-arbb>. Citado na página 33.

ÉTIENNE, E. Y. **Hyper-Threading**. [S.l.]: TurbsPublishing, 2012. ISBN 6135655345. Citado na página 47.

FOSTER, I. **Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering**. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201575949. Citado 2 vezes nas páginas 21 e 44.

GOULD, H.; TOBOCHNIK, J.; CHRISTIAN, W. **An Introduction to Computer Simulation Methods**. Third edition. [S.l.]: Open Source Physics, 2016. ISBN 978-1974427475. Citado na página 19.

GRAHAM, S. L.; KESSLER, P. B.; MCKUSICK, M. K. Gprof: A Call Graph Execution Profiler. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 17, n. 6, p. 120–126, jun. 1982. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/872726.806987>. Citado 2 vezes nas páginas 45 e 54.

HERNANDEZ, E.; PALACIOS, G.; MARíN, C. Parallel programming languages on heterogeneous architectures using openmpc, ompss, openacc and openmp. **Tecnura**, v. 18, 08 2015. Citado 2 vezes nas páginas 22 e 23.

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, p. 012056, jan 2021. Disponível em: <https://doi.org/10.1088/1742-6596/1740/1/012056>. Citado 2 vezes nas páginas 21 e 22.

KIRK, D. B.; HWU, W.-m. W. **Programming Massively Parallel Processors: A Hands-on Approach**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN 0123814723. Citado na página 33.

LARREA, V. G. V. et al. Experiences in porting mini-applications to openacc and openmp on heterogeneous systems. **Concurrency and Computation. Practice and Experience**, v. 32, n. 20, 4 2020. ISSN 1532-0626. Disponível em: <https://www.osti.gov/biblio/1649533>. Citado na página 22.

LIPATNIKOV, A. **Numerical Simulations of Turbulent Combustion**. [S.l.]: MDPI, 2020. Citado na página 25.

LUCCA, N.; SCHEPKE, C. Programando aplicações com diretivas paralelas. In: _____. [S.l.: s.n.], 2020. p. 89–104. ISBN 9786587003009. Citado 2 vezes nas páginas 33 e 34.

MANCO, J. A. A. **Condições de contorno não reflexivas para simulação numérica de alta ordem de instabilidade de Kelvin-Helmholtz em escoamento compressível**. 134 p. Tese (Doutorado) — Instituto Nacional de Pesquisas Espaciais (INPE), 2014. Citado 4 vezes nas páginas 19, 25, 28 e 30.

MANCO, J. A. A. **Stability characteristic of subsonic binary axisymmetric coaxial jets**. 112 p. Tese (Doutorado) — Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, 2019-11-06 2020. Disponível em: <http://urlib.net/rep/8JMKD3MGP3W34R/3UFNTF2>. Citado na página 25.

MANCO, J. A. A.; MENDONCA, M. T. de. Comparative study of different non-reflecting boundary conditions for compressible flows. **Journal of the Brazilian Society of Mechanical Sciences and Engineering**, Springer, v. 41, n. 10, p. 1–16, 2019. Citado na página 26.

MOORE, G. E. Cramming more components onto integrated circuits. **Electronics**, v. 38, n. 8, April 1965. Citado na página 33.

OPENACC. **What is OpenACC?** 2022. [Online; acessado em 15 de fevereiro de 2022]. Disponível em: <https://www.openacc.org/>. Citado 4 vezes nas páginas 20, 36, 37 e 38.

OPENMP. **The OpenMP API specification for parallel programming**. 2022. [Online; acessado em fevereiro, 15 2022]. Disponível em: <https://www.openmp.org/>. Citado 5 vezes nas páginas 20, 22, 34, 35 e 36.

PAPERIN, M. **Kelvin-Helmholtz Instability Cloud Structure**. 2007. Disponível em: <https://www.brockmann-consult.de/CloudStructures/kelvin-helmholtz-instability-description.htm>. Citado na página 26.

PIZZOLATO, G.; SCHEPKE, C. Explorando paralelismo de laços em uma aplicação de simulação de câmara de combustão. In: **Anais da XXI Escola Regional de Alto Desempenho da Região Sul**. Porto Alegre, RS, Brasil: SBC, 2021. p. 37–40. ISSN 2595-4164. Disponível em: <https://sol.sbc.org.br/index.php/eradrs/article/view/14769>. Citado na página 55.

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 55.

PIZZOLATO, G. L.; SCHEPKE, C. Uso de operaÇÕes em gpu para acelerar uma aplicaÇÃo de simulaÇÃo de combustÃo. **Anais do Salão Internacional de Ensino, Pesquisa e Extensão**, v. 12, n. 2, dez. 2020. Disponível em: <https://periodicos.unipampa.edu.br/index.php/SIEPE/article/view/107302>. Citado na página 55.

PIZZOLATO, G. L.; SCHEPKE, C. Acelerando uma simulaÇÃo numÉrica de cÂmara de combustÃo. **Anais do Salão Internacional de Ensino, Pesquisa e Extensão**, v. 13, n. 3, nov. 2021. Disponível em: <https://periodicos.unipampa.edu.br/index.php/SIEPE/article/view/110857>. Citado na página 55.

SILVA, M. et al. Mixing layer stability analysis with strong temperature gradients. In: **17th Brazilian Congress of Thermal Sciences and Engineering (ENCIT 2018)**. [S.l.: s.n.], 2017. Citado na página 19.

SILVA, M. C. N. da. **The influence of kelvin-helmholtz instabilitty in a mixing layer: a simple model for the study of fire safety between two parallel walls**. 40 p. Tese (Trabalho de Conclusão de Curso (Bacharel em Engenharia civil)) — Universidade Federal do Pampa, Curso de Engenharia Civil, Alegrete/RS, 2020. Citado na página 27.

STONE ROGER L. DAVIS, D. Y. L. C. P. Concurrent parallel processing on graphics and multicore processors with openacc and openmp. **Accelerator Programming Using Directives**, Springer International Publishing, 2018. ISSN Print ISBN: 978-3-319-74895-5; Electronic ISBN: 978-3-319-74896-2. Disponível em: <https://www.springerprofessional.de/concurrent-parallel-processing-on-graphics-and-multicore-process/15456018>. Citado 2 vezes nas páginas 22 e 23.

ZHANG, S. et al. Parallel computation of a dam-break flow model using openacc applications. **Journal of Hydraulic Engineering**, v. 143, p. 04016070, 08 2016. Citado 2 vezes nas páginas 22 e 23.