Aceleração de uma Aplicação de Simulação de Câmara de Combustão em Multi-Core*

Glener Lanes Pizzolato, Claudio Schepke, Natiele Lucca

[1]Laboratório de Estudos Avançados em Computação (LEA)
Universidade Federal do Pampa (UNIPAMPA) – Alegrete – RS – Brasil

{glenerpizzolato.aluno,claudioschepke,natielelucca.aluno}@unipampa.edu.br

Abstract. *Developing new propulsive engines, evaluating fuel combustion or proposing new catalysis substances can be computer simulated without the need to build a real environment. For that, an application was developed that allows representing the functioning of a combustion chamber. However, a discrete representation of a simulation has an expressive processing time (hours). In this article, parallelization techniques for an application are proposed and evaluated, in order to run as simulations in multicore architecture. The results found show that an application was faster in a multicore architecture.*

Resumo. *Desenvolver novos motores propulsivos, avaliar a queima de combustíveis ou propor novas substâncias de catálise podem ser simulados computacionalmente sem a necessidade de construção de um ambiente real. Para tanto, foi desenvolvido uma aplicação que possibilita representar o funcionamento de uma câmera de combustão. No entanto, a representação discreta de uma simulação possui um expressivo tempo de processamento (horas). Neste artigo são propostas e avaliadas técnicas de paralelização para a aplicação, a fim de executar as simulações em arquitetura multicore. Os resultados obtidos mostram que a aplicação foi mais rápida em uma arquitetura multicore.*

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. 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].

O desempenho dos sistemas de combustão em turbinas de motores de foguete depende fortemente na injeção adequada e mistura entre combustível e oxidante [Manco 2020]. 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). Uma visão seccional deste tipo de injetor também é apresentado na figura, que é usado para misturar oxigênio líquido (LOX) e hidrogênio gasoso (GH2).

*O trabalho foi desenvolvido com apoio da bolsa de Iniciação Científica PIBIC/CNPq 2020 e PROBIC/FAPERGS 2021.

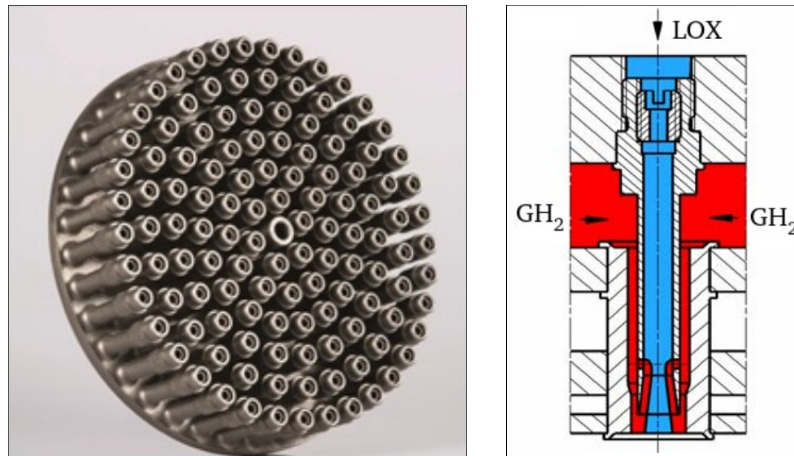


Figura 1. (a) Injetores coaxiais do motor principal do ARIANE 6. (b) Visão esquemática de um injetor [Manco 2020].

A fim de prover a modelagem de diferentes tipos de combustíveis em uma câmera de combustão foi desenvolvido uma aplicação computacional [Manco 2014]. Para tanto, o modelo é representado bidimensionalmente através das equações de Euler. Posteriormente a aplicação foi modificada, modelando a câmara de combustão através das equações de Navier-Stokes [Silva et al. 2017]. Isso possibilita uma representação mais ampla de diferentes fluídos com diferentes características físicas. A camada de mistura compreensível 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 [da Silva 2020]. Em todos os casos, duas correntes paralelas em velocidades diferentes podem ser compostas por diferentes espécies químicas ou com grandes diferenças de temperatura, conforme Figura 2. Essas simulações numéricas diretas de alta ordem são frequentemente usadas para resolver as escalas espaço-temporais de tais fluxos.

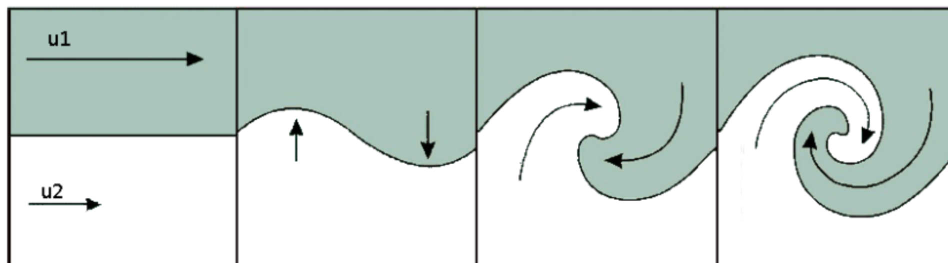


Figura 2. Região de mistura [PAPERIN 2007].

Simulações como o da câmera de combustão requerem alta precisão numérica, o que geralmente resulta em um custo de processamento computacional significativo. O código original foi escrito em Fortran 90 e foi implementado de forma sequencial, o que torna demorado, na escala de horas, realizar algumas simulações específicas. Diante disso, o objetivo deste artigo é propor técnicas de paralelização a fim de reduzir o tempo de processamento da computação das equações para que a aplicação possa ser executada em arquiteturas multi-core. Para tanto, são utilizadas operações paralelas oferecidas pela

interface de programação paralela OpenMP. A biblioteca provê a criação implícita de *threads* tanto em laços como em seções paralelas, através do uso de diretivas de compilação.

O artigo está organizado da seguinte forma. Na Seção 2 é apresentada a formulação matemática do modelo. A Seção 3 descreve os métodos e implementações utilizados para a realização deste trabalho. A Seção 4 mostra e discute os resultados obtidos. Por fim, na Seção 5, são destacadas as considerações finais sobre o artigo.

2. Representação Numérica da Câmara de Combustão

Esta pesquisa aborda a questão de como os gradientes de temperatura afetam o desenvolvimento da instabilidade de Kelvin-Helmholtz [da Silva 2020]. O trabalho contribui para o cálculo da estabilidade e acústica de escoamentos compressíveis relevantes para sistemas de propulsão.

O problema é executado 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 e radiação, conforme representação da Figura 3 e descrito na Equação 1.

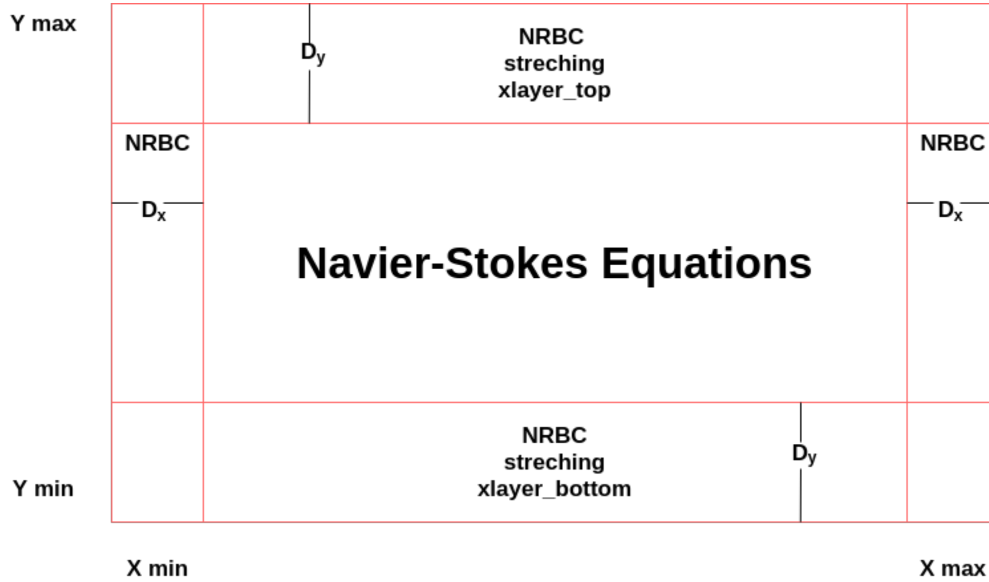


Figura 3. Representação esquemática do modelo (Adaptado de [Manco and de Mendonca 2019])

$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho e \end{bmatrix},\ E = \begin{bmatrix} \rho u \\ \rho u^2 + p - \tau_{xx} \\ \rho uv - \tau_{xy} \\ \rho eu + q_x - u\tau_{xx} - v\tau_{xy} \end{bmatrix},$$
$$F = \begin{bmatrix} \rho v \\ \rho uv - \tau_{xy} \\ \rho v^2 + p - \tau_{yy} \\ \rho ev + q_y - u\tau_{xy} - v\tau_{yy} \end{bmatrix} \tag{1}$$

Assim, o conjunto de equações é definido conforme apresentado na Equação 2.

$$\frac{\partial U}{\partial t} + \frac{\partial E}{\partial x} + \frac{\partial F}{\partial y} = 0. \tag{2}$$

Onde as tensões viscosas estão relacionadas ao tensor de deformação linearmente, conforme é mostrado na Equação 3.

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

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 4.

$$Re = \frac{\rho_1 U_1 \delta_w}{\mu_1}, Pr = \frac{\mu_1 cp_1}{k_1}, Ma = \frac{U_1}{a_1} \tag{4}$$

Onde, ρ é a densidade, U_1 é a velocidade do fluxo rápido, mi_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 (Runge-Kutta de sexta 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 algoritmo é executado 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 4.

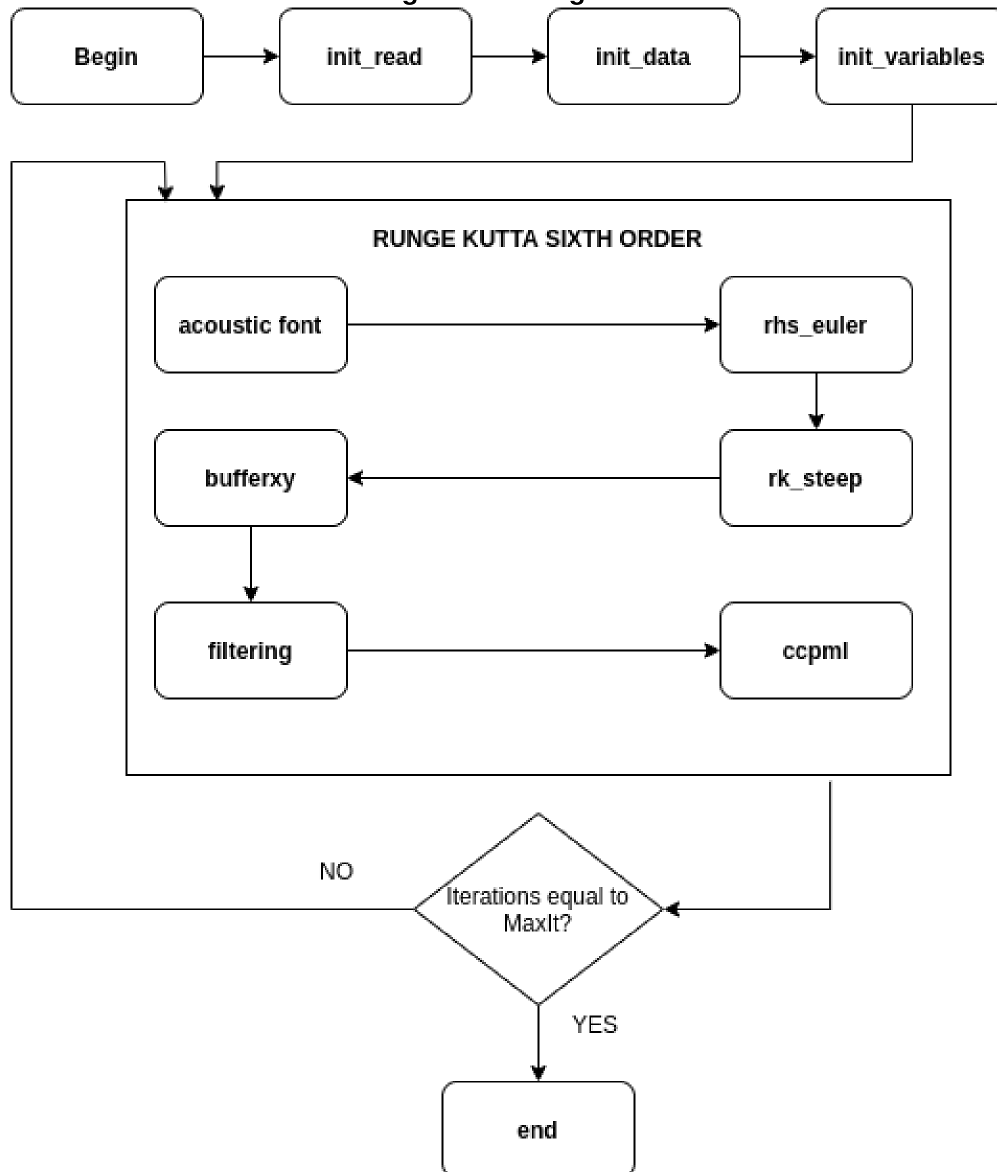
As operações realizadas dentro do laço principal que itera o *dt* de tempo são:

- *acoustic font*: 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 5 e Figura 6 apresentam uma saída com e sem filtro, respectivamente [Manco 2014].
- *ccpml*: Faz o tratamento das condições de contorno da região de interesse.

3. Aspectos Metodológicos

A versão sequencial original do código implementado em FORTRAN foi inicialmente avaliada. Foram feitos testes para mensurar o custo de inicialização (leitura de arquivo

Figura 4. Fluxograma



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 que serve de *baseline* para a avaliação das otimizações.

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

- *Observar:* eram definidas tarefas como mapeamento de variáveis e funções, verificação de trechos paralelizáveis e planejamento para o próximo pilar.
- *Otimizar:* era realizado o processo de otimização, tais como paralelizar trechos de código, exclusão de partes desnecessárias, modificação de funções e movimentação de blocos de código.
- *Avaliar:* o código era então compilado e executado. Caso nenhum erro fosse constatado, ainda sim, era verificada e comparada a saída da versão serial original do código com a versão atual em processo de otimização. Era obrigatório que as

Figura 5. Execução Com Filtro

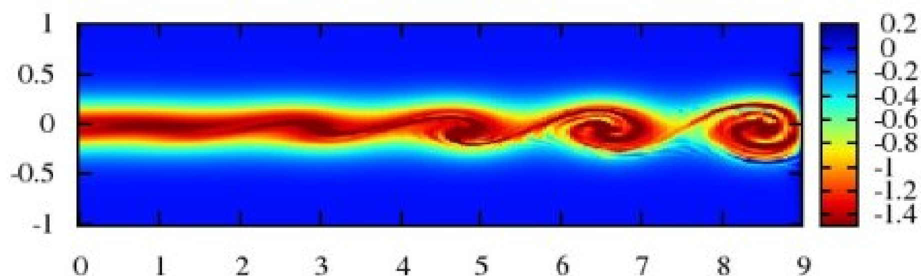
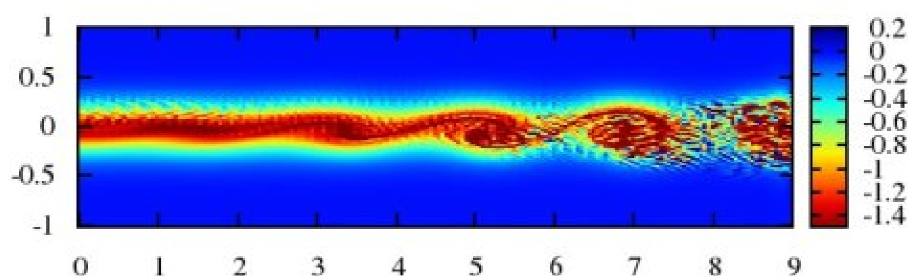


Figura 6. Execução Sem Filtro



duas saídas não tivessem nenhuma alteração e o código deveria ter sido executado em menos tempo que a versão antecessora, caso contrário, a otimização teve um impacto negativo no desempenho.

Desta forma, a medida que o código ia sendo alterado ou paralelizado, foram feitas avaliações de desempenho parciais. Incrementalmente essa etapa repetiu-se até não ser encontrado mais nenhum ganho de desempenho com novas alterações.

3.1. Avaliando a performance de trechos de código

Na sequência, o código foi então avaliado com a ferramenta `gprof` [Graham et al. 1982], que faz coletas estatísticas do tempo de execução demandado por cada rotina que compõe o código. A Tabela 1 apresenta os resultados obtidos com a execução da ferramenta `gprof`. Observa-se na tabela as rotinas que mais demandam tempo de execução.

Tabela 1. Principais rotinas que demandam tempo de Processamento

Rotina	Arquivo	Porcentagem de tempo da execução total
deropn	diff.f90	25,23%
deropm	diff.f90	15,26%
lddfilterx	filltering.f90	8,04%
lddfiltery	filltering.f90	7,22%
rhs_euler	filltering.f90	19,66%

As subrotinas `deropm` e `deropn` calculam a derivada primeira de uma função u utilizando diferenças centradas de quarta ordem nos pontos internos do domínio, diferenças centradas de segunda ordem nos pontos vizinhos à fronteira e diferenças unilateral

de segunda ordem nos pontos de fronteira. A primeira subrotina `deropm` opera em x e a segunda subrotina `deropn` em y.

As subrotinas `lddfilterx` e `lddfiltery` apresentam esquemas explícitos para calcular o ruído aerodinâmico. Já na rotina `rhs_euler` são feitas apenas alocações de memórias temporárias desnecessárias, as quais foram removidas. Porém, as chamadas de sub-rotinas necessárias para o processamento dos dados foram mantidas.

Com base nos resultados de *profile* obtidos, foram feitas inspeções nas rotinas, identificando as rotinas que demandavam mais tempo de execução. Nessa inspeção também foi possível identificar os laços aninhados nas rotinas. Estes laços operam sobre estruturas de dados bidimensionais que representam cada uma das propriedades físicas. Portanto, esse conjunto de operações foi paralelizado com as diretivas da API OpenMP.

Em um segundo momento foram feitas inspeções em outras rotinas com menor impacto no tempo total de execução (menor que 5%, de acordo com o *GProf*), e que previamente não foram investigadas. Essas rotinas por mais que sejam pequenas são numerosas e também puderam ser paralelizadas, a paralelização de tais rotinas também contribuiu para a redução do tempo total de execução da aplicação.

3.2. Experimentos

Todos os experimentos foram realizados considerando 100 iterações da etapa iterativa principal da aplicação e os resultados finais de cada teste são calculados a partir da média de 10 execuções. Embora em um experimento real o número de iterações necessárias seja maior (ex. 8000 iterações), para os testes constantes neste trabalho realizado, não se torna necessário manter o número de iterações tão alto, além do fato de ter sido observado que o tempo médio de cada iteração é computacionalmente (número de instruções/operações) e na prática o mesmo.

100 iterações é o número mínimo de iterações para que a execução passe por todas funcionalidades do código, visto que, caso seja um valor muito pequeno, ou seja, menor que 100, alguns trechos do código não são necessários na execução, o que pode gerar erros na otimização. 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 impacto em cada pequena alteração realizada no código.

A Tabela 2 apresenta testes distintos baseado em uma média de 3 experimentos para cada caso. O número total de iterações testado foi 100, 200, 400 e 600, visando medir o custo por iteração e o desvio padrão entre os experimentos em cada teste.

Tabela 2. Custo por iteração variando número total de Iterações

Número de iterações	Custo por Iteração
100	1,3739s
200	1,3792s
400	1,3755s
600	1,3811s

Para avaliar o custo de inicialização e pós processamento, reduziu-se o número de iterações para o valor 0. O valor obtido aqui corresponde a criação e ao preenchimento das estrutura de dados, a leitura do arquivo de entrada e a alocação de memória para a inicialização e de escritas de saída e liberação de memória na etapa de pós-processamento. Em geral, tais tarefas são seriais e não possuem muitos trechos que podem ser paralelizados.

Para avaliar o custo de iteração para cada teste realizado com um dado número de *threads*, utilizou-se da Equação 5, onde `M_Custo_Total` representa a média de 10 execuções do tempo total de execução com 100 iterações e `M_Custo_Inicialização` representa a média de 10 execuções de somente o tempo de inicialização e pós-processamento do experimento. Dessa forma é desconsiderado o tempo de inicialização/pós-processamento na fórmula. Em seguida o valor é dividido pelo valor `N` que representa o número de iterações, que no nosso caso é igual a 100. Com esse método, é encontrado o custo exato de tempo que cada iteração possui.

$$Custo_{\text{Iteração}} = \frac{M_{CustoTotal} - M_{CustoInicial}}{N} \quad (5)$$

3.3. Ambiente de Validação

Os testes foram feitos utilizando o compilador `pgf90` versão 20.9 do pacote HPC_SDK da NVIDIA, considerando as diretivas `-O3 -fastsse -mp` para os experimentos com a API OpenMP [OpenMP 2021] Outros testes também foram feitos com o compilador `gfortran` com diretivas similares [Pizzolato and Schepke 2021]. No entanto, o tempo de execução sequencial e paralelo foram maiores para este compilador.

Todos os testes foram realizados em uma *workstation* da Universidade Federal do Pampa (UNIPAMPA), campus Alegrete, que possui a configuração apresentada na Tabela 3 e como sistema operacional foi utilizado Ubuntu na versão 20.04 de 64 bits.

Tabela 3. Ambiente CPU

Características	Xeon E5-2650
Frequência	2.00 GHz
Núcleos	8 (x2)
Threads	16 (x2)
Cache L1	32 KB
Cache L2	256 KB
Cache L3	20 MB
Memória RAM	128 GB

4. Resultados

Para medir o desempenho das versões dos algoritmos paralelos foi utilizado o conceito de *speed up*(S). O *speed up* é definido como a razão entre o tempo de computação do algoritmo serial (T_{serial}) e o tempo de computação do algoritmo paralelo ($T_{paralelo}$) [Étienne 2012]. O cálculo da eficiência é baseado no tempo de execução e o número de *threads* utilizado.

A Figura 4 apresenta a média dos valores de cada uma das 10 execuções para os tempos de inicialização/finalização e custo por iteração em segundos. Além do caso sequencial, foram medidos os tempos usando 2, 4, 8, 16 e 32 *threads*.

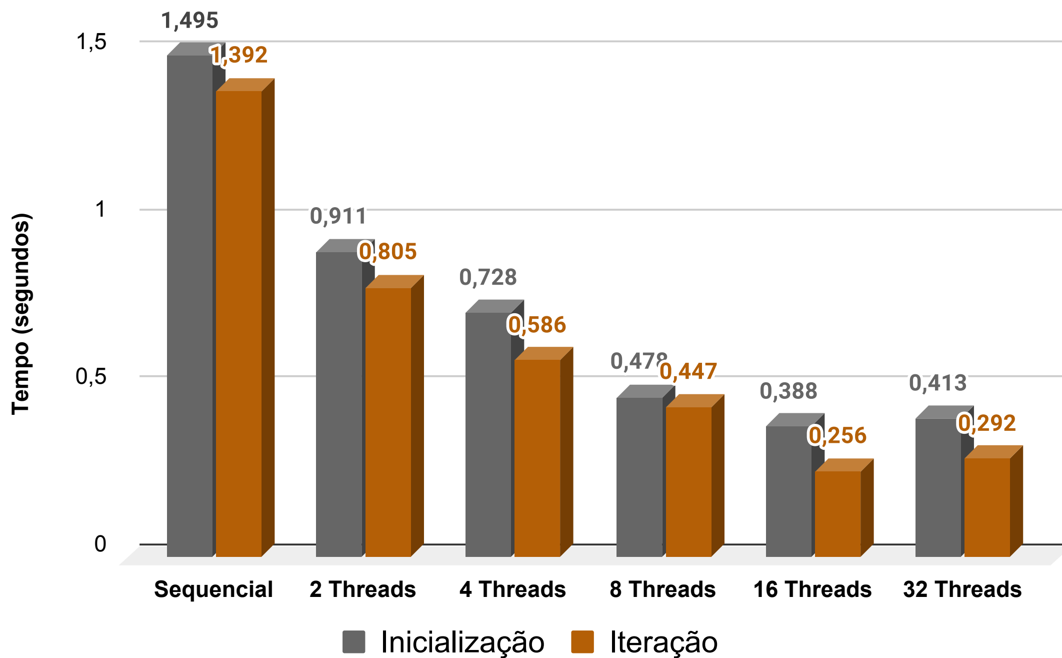


Figura 7. Tempo de inicialização/finalização e custo médio de cada iteração

Ao se utilizar duas *threads*, nota-se que o custo por iteração foi reduzido a aproximadamente metade. Utilizando-se de mais *threads*, o tempo de inicialização e custo por iteração continuou sendo reduzido expressivamente. Percebe-se também que o custo de inicialização não diminui tão significativamente, uma vez que os trechos paralelizáveis são menores.

A Tabela 4 apresenta o tempo total de execução de cada simulação usando 100 iterações do algoritmo, e os valores de *speed up*(S) separados para a etapa de inicialização/finalização e iterativa, da execução sequencial e paralelas, variando o número de *threads* de 2 a 32.

Tabela 4. Tempo Total e SpeedUP

Modo Execução	**Tempo**	***speed up***(S) **Inicialização**	***speed up***(S) **Iteração**
Sequencial	140,69s	-	-
2 Threads	81,39s	1,64	1,73
4 Threads	59,32s	2,05	2,38
8 Threads	45,18s	3,13	3,11
16 Threads	25,94s	3,85	5,45
32 Threads	29,62s	3,62	4,77

O melhor resultado para o custo por iteração foi obtido ao utilizar-se 16 *threads*. Neste caso, obteve-se um *speed up*(S) de 5,45 para o custo de cada iteração e 3,85 de *speed up*(S) para a inicialização. Vale salientar que quanto mais *threads* foram usadas, menor foi o tempo de execução. 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. Nota-se que o *Hyper-Threading* (uso de 32 *threads*) não surtiu uma redução de tempo mais expressiva. Essa tecnologia permite que cada núcleo de um processador possa executar duas *threads* de uma única vez [Étienne 2012].

O cálculo da eficiência, Figura 8, serviu para mensurar qual impacto teve cada *thread* no *speed up*(S) em cada experimento. É natural a queda de eficiência ao se aumentar o número de *threads*, visto que a solução se aproxima do ápice máximo de ganho de desempenho do problema.

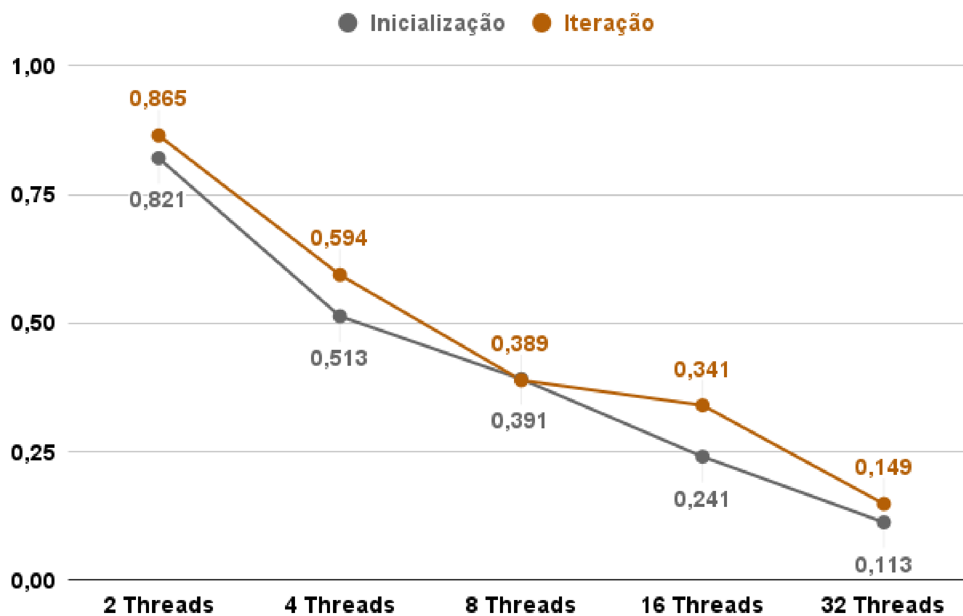


Figura 8. Eficiência de cada *thread* para o resultado do *speed up*(S)

Como possuímos os dados necessários para calcular o tempo total de um exemplo real, sendo o número de iterações (8000) e o custo por iteração, se torna viável mensurar a redução de tempo para um experimento real.

A redução de tempo é expressiva para o melhor caso da *OpenMP*, reduzindo o tempo de aproximadamente 3 horas da versão sequencial original, para aproximadamente 34 minutos para o melhor caso ao se utilizar *16* threads. Observa-se a tabela 5 que apresenta o tempo gasto para cada caso de um experimento real.

Tabela 5. Tempo Total e *speed up*(S) para um exemplo real (8000 iterações)

Modo Execução	Tempo total
Sequencial	11136s
2 Threads	6439s
4 Threads	4688s
8 Threads	3576s
16 Threads	2044s
32 Threads	2337s

5. Conclusão e Trabalhos Futuros

Aplicações que possibilitam a simulação de fenômenos e meios físicos correspondem a uma gama de estratégias eficientes para a solução de problemas de diversas áreas, inclusive problemas complexos, como a simulação de uma câmara de combustão.

O objetivo principal deste trabalho foi introduzir técnicas paralelas em uma aplicação de simulação de uma câmera de combustão. Foram obtidos ganhos de desempenho para o melhor caso de 5.45 de *speed up* em testes realizados em uma arquitetura multicore, usando 16 *threads*. O que permitiu reduzir o tempo do experimento de aproximadamente 3 horas para aproximadamente 34 minutos.

Como trabalhos futuros pretende-se otimizar ainda mais o código utilizando a API OpenACC [OpenACC 2021], utilizando *GPU* para aumentar o poder computacional e obter melhores ganhos de desempenho.

Referências

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

[Étienne 2012] Étienne, E. Y. (2012). *Hyper-Threading*. TurbsPublishing.

[Graham et al. 1982] Graham, S. L., Kessler, P. B., and Mckusick, M. K. (1982). Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 17(6):120–126.

[Lipatnikov 2020] Lipatnikov, A. (2020). *Numerical Simulations of Turbulent Combustion*. MDPI.

[Manco 2014] Manco, J. A. A. (2014). *Condições de contorno não reflexivas para simulação numérica de alta ordem de instabilidade de Kelvin-Helmholtz em escoamento compressível*. PhD thesis, Instituto Nacional de Pesquisas Espaciais (INPE).

[Manco 2020] Manco, J. A. A. (2020). *Stability characteristic of subsonic binary axisymmetric coaxial jets*. PhD thesis, Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos.

[Manco and de Mendonca 2019] Manco, J. A. A. and de Mendonca, M. T. (2019). Comparative study of different non-reflecting boundary conditions for compressible flows. *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, 41(10):1–16.

[OpenACC 2021] OpenACC (2021). What is openacc? [Online; acessado em maio, 28 2021].

[OpenMP 2021] OpenMP (2021). The openmp api specification for parallel programming. [Online; acessado em junho, 15 2021].

[PAPERIN 2007] PAPERIN, M. (2007). Kelvin-helmholtz instability cloud structure.

[Pizzolato and Schepke 2021] Pizzolato, G. and Schepke, C. (2021). 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*, pages 37–40, Porto Alegre, RS, Brasil. SBC.

[Silva et al. 2017] Silva, M., Cristaldo, C., Manco, J. A. A., Fachini, F., and de Mendonça, M. T. (2017). Mixing layer stability analysis with strong temperature gradients. In *17th Brazilian Congress of Thermal Sciences and Engineering (ENCIT 2018)*.