

# Paralelização de uma Aplicação de Câmara de Mistura com OpenMP\*

Glener L. Pizzolato<sup>1</sup>, Claudio Schepke<sup>1</sup>

<sup>1</sup>Universidade Federal do Pampa (UNIPAMPA), Alegrete, Brazil

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

***Resumo.** Um programa para realizar simulações de uma câmara de mistura foi modelado e implementado de forma sequencial em Fortran 90. Conseqüentemente o programa possui um tempo computacional expressivo. Neste trabalho é proposto o uso da programação paralela para acelerar a execução do código. Foram utilizadas operações fornecidas pela interface de programação paralela OpenMP, usando duas malhas de tamanhos distintos como estudo de caso.*

## 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 et al. 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 algumas das principais características da realização de simulações computacionais. 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, o qual provê a capacidade de simulação de uma câmara de mistura, foi desenvolvido por [Manco 2014]. A aplicação tem como fim avaliar a mistura entre combustível e oxidante, por exemplo. Para tanto, o modelo físico é representado bidimensionalmente através das equações de Euler.

[Silva et al. 2017] adaptaram o programa original para que o mesmo usasse as equações de Navier-Stokes como modelo físico. Isso amplia a possibilidade de representação de fluidos com diferentes propriedades físicas. Em todos os casos, duas correntes paralelas em velocidades diferentes podem ser compostas de diferentes espécies químicas ou com grandes diferenças de temperatura, conforme a Figura 1. Essas simulações numéricas diretas de alta ordem são frequentemente usadas para resolver as escalas espaço-temporais de tais fluxos.

[Pizzolato and Schepke 2021] realizaram algumas paralelizações nesta aplicação, reduzindo o tempo de processamento. O presente artigo é uma evolução do mesmo, explorando diferentes parâmetros de entrada para a aplicação, conforme a Seção 2.1.

Aplicações como a da câmara de mistura requerem alta precisão numérica o que geralmente resulta em um custo de processamento computacional significativo. Um experimento típico demora em torno de 45 minutos. A implementação original da aplicação de simulação não explora o uso de multicore. Diante disso, o objetivo deste trabalho é otimizar e paralelizar a computação dos valores numéricos das propriedades físicas simuladas. Desta forma, a aplicação de simulação de mistura é executada sob forma de threads.

---

\*Trabalho parcialmente suportado pela FAPERGS

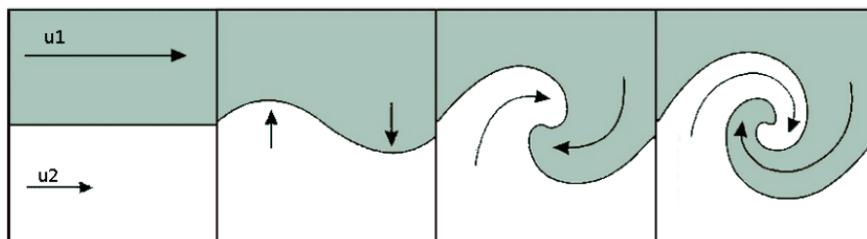


Figura 1. Instabilidade de Kelvin-Helmholtz [PAPERIN 2007]

## 2. Metodologia

OpenMP é uma API fundamentada no modelo *fork-join* [OpenMP 2020]. Esse modelo possui uma thread mestre que inicia a execução e gera threads de trabalho para executar as tarefas em paralelo. O *fork-join* é aplicado em segmentos do código que são informados pelo programador (ou usuário). A API OpenMP possui um conjunto de diretivas de compilação, funções e variáveis de ambiente para a programação paralela. Na linguagem Fortran 90, utilizada na codificação do código, uma diretiva é precedida obrigatoriamente por `!$omp` e seguida por `[atributos]`, sendo que os atributos são opcionais.

Para identificar os blocos mais custoso do código para inicializar a implementação utilizou-se da ferramenta *gprof* [Graham et al. 1982] que faz coletas estatísticas de tempo de cada rotina executada durante a aplicação. E para auxiliar na identificação dos blocos paralelizáveis foi adicionado a `flag -Minfo=all` da OpenMP, que ao compilar provê um *log* ao programador com possíveis soluções de paralelizações. Maiores informações dos arquivos do código fonte podem ser obtidos em [Pizzolato et al. 2021]. O código fonte está disponível no Github<sup>1</sup>.

### 2.1. Parâmetros de Entrada

Duas entradas foram escolhidas como estudo de caso, conforme a Tabela 1. Dessa forma foi possível calcular o ganho de desempenho com diferentes tamanhos de entrada para o problema, podendo mensurar o quão eficaz tornou-se a versão paralela.

Tabela 1. *Malha I e Malha II*

Descrição	Variável	Valor <i>Malha I</i>	Valor <i>Malha II</i>
Dimensão x da Câmara	<code>imax</code>	461	921
Dimensão y da Câmara	<code>jmax</code>	381	761

### 2.2. Ambiente de Validação

Todos os testes foram realizados em uma *workstation* da Universidade Federal do Pampa (UNIPAMPA), campus Alegrete, possui um processador *Xeon E5-2650 (x2)* com frequência de 2.0 GHz, 8 (x2) Núcleos e 16 (x2) Threads, *Cache L1* de 32 KB, *Cache L2* de 256 KB e *Cache L3* de 20 MB e uma memória RAM de 128 GB. Como sistema operacional foi utilizado Ubuntu na versão 20.04 de 64 bits e *kernel 5.11.0-27-generic*.

<sup>1</sup><https://github.com/glener10/MixingChamber-OpenMPVersion>

O código-fonte foi compilado com `pgf90` versão 20.9 do pacote `HPC_SDK` da NVIDIA. Foram utilizadas as diretivas `-O3 -mp` para os experimentos. Outros testes também foram feitos com o compilador `gfortran` com diretivas similares. No entanto, o tempo de execução sequencial e paralelo foram maiores neste caso. Para os testes, o ambiente foi reservado somente para os experimentos, a fim de evitar quaisquer interferência de algum processo, e conseqüentemente, imprecisões nos resultados.

### 3. Desenvolvimento e Resultados

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

**Tabela 2. Custo por Iteração *Malha I* e *Malha II***

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

Ao se utilizar duas threads, nota-se que o custo por iteração foi reduzido consideravelmente para as duas entradas avaliadas. Nota-se que o *Hyper-Threading* (uso de 32 threads) não surtiu uma redução de tempo mais expressiva para nenhuma das entradas. Essa tecnologia permite que cada núcleo de um processador possa executar duas threads de uma única vez [Étienne 2012].

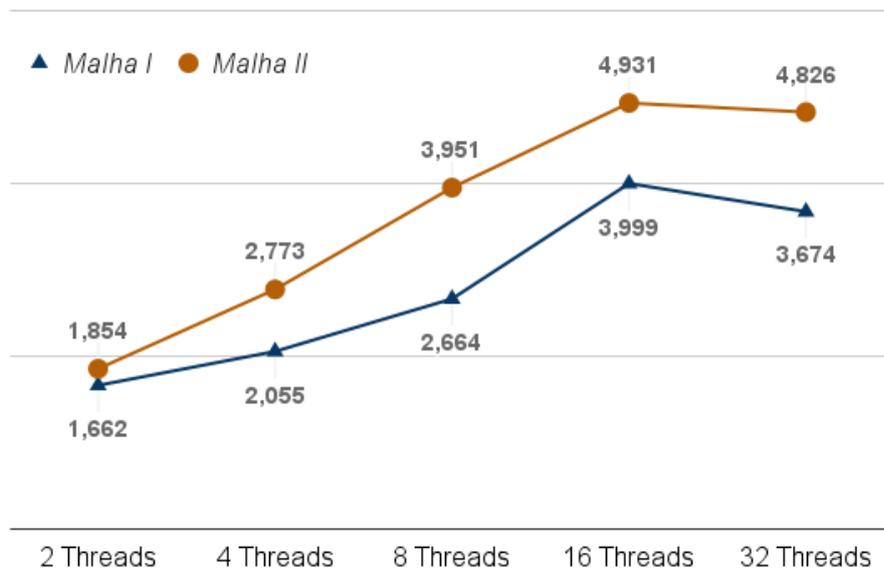
O speedup é mostrado na Figura 2. O melhor resultado para ambas as entradas foi obtido ao usar 16 threads. Neste caso, obteve-se o speedup de aproximadamente 4 para a *Malha I* e de aproximadamente 5 para a *Malha II* para o custo por cada iteração.

Ao comparar as duas entradas, a *Malha II* em relação à *Malha I* obteve melhores resultados para todos os casos. Assim, é possível afirmar que quanto mais precisa a malha utilizada como caso de teste, maior é o número de pontos a serem computados e maior o nível de paralelismo e ganho de desempenho para a versão paralela implementada.

### 4. Considerações Finais e Trabalhos Futuros

A redução do tempo de processamento é expressiva e expressiva com a paralelização usando OpenMP. Para a *Malha I*, o tempo de processamento foi reduzido de aproximadamente 45 minutos da versão sequencial original, para aproximadamente 11 minutos ao utilizar-se 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 da versão sequencial original para aproximadamente 1 hora e 50 minutos.

A próxima etapa a ser realizada na aplicação é uma implementação com OpenACC a fim de explorar o paralelismo utilizando GPU. Também fica em aberto com a atual versão desta aplicação, 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.



**Figura 2. Speedup para Malha I e Malha II**

## Referências

- Étienne, E. Y. (2012). *Hyper-Threading*. TurbsPublishing.
- Gould, H., Tobochnik, J., and Christian, W. (2016). *An Introduction to Computer Simulation Methods*. Open Source Physics, Third edition.
- Graham, S. L., Kessler, P. B., and Mckusick, M. K. (1982). Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 17(6):120–126.
- 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).
- OpenMP (2020). The OpenMP API specification for parallel programming. [Online; acessado em novembro, 15 2021. <https://www.openmp.org>].
- PAPERIN, M. (2007). Kelvin-helmholtz instability cloud structure. [Online; acessado em novembro, 15 2021. <https://www.brockmann-consult.de/CloudStructures/kelvin-helmholtz-instability-description.htm>].
- 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.
- Pizzolato, G., Schepke, C., and Lucca, N. (2021). Aceleração de uma aplicação de simulação de câmara de combustão em multi-core. In *Anais do XXII Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 36–47, Porto Alegre, RS, Brasil. SBC.
- 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)*.