Explorando paralelismo de laços em uma Aplicação de Simulação de Câmara de Combustão*

Glener Lanes Pizzolato[1], **Claudio Schepke** [1]

[1]Ciência da Computação – Universidade Federal do Pampa (UNIPAMPA)
Alegrete – RS – Brazil

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

Resumo. *Neste trabalho é proposto a otimização e paralelização de um programa que realiza a simulação de uma câmera de combustão. O programa sequencial original possui um tempo computacional expressivo, levando horas para computar todos os dados. Para acelerar a execução do código foram utilizadas operações paralelas usando OpenMP, os quais foram avaliados em uma arquitetura multicore, reduzindo o tempo total de processamento do programa.*

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. Um exemplo de aplicação computacional foi desenvolvido por [Manco 2014], o qual provê a capacidade de simulação de uma câmera de combustão, a fim de avaliar a mistura entre combustível e oxidante. Para tanto, o modelo 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, o qual possibilita uma representação mais ampla de fluidos 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. 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.

Simulações como o da câmera de combustão requerem alta precisão numérica o que geralmente resulta em um custo de processamento computacional significativo. Diante disso, o objetivo deste trabalho é otimizar e paralelizar a computação das equações discretas de Navier-Stokes para que a aplicação possa ser executada em máquinas 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. A interface é simples e fácil de se utilizar, sendo compatível com a linguagem FORTRAN 90 utilizada no código.

*O presente trabalho foi desenvolvido no Laboratório de Estudos Avançados (LEA) com apoio da bolsa de Iniciação Científica PROBIC/FAPERGS 2020/2021.

O artigo está organizado da seguinte forma. A Seção 2 apresenta os métodos e implementações utilizados para a realização deste trabalho. A Seção 3 mostra os resultados obtidos. Por fim, na Seção 4, são destacadas as considerações finais sobre o trabalho.

2. Metodologia e Implementação

A versão sequencial original do código implementado em FORTRAN foi inicialmente avaliada. Foram feitos testes para mensurar o custo de inicialização (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, tem-se o tempo total sequencial da execução da aplicação.

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. Observou-se que as rotinas `deropm` (15,26%) e `deropn` (25,23%) do arquivo `diff.f90` e `lddfilterx` (8,04%) e `lddfiltery` (7,22%) do arquivo `filltering.f90` são as principais rotinas que demandam tempo de processamento. 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 opera em x e a segunda em y. Já as subrotinas `lddfilterx` e `lddfiltery` apresentam esquemas explícitos para calcular o ruído aerodinâmico.

Com base nos resultados de *profile* obtidos, foram feitas inspeções nas rotinas, identificando-se laços aninhados nas rotinas. Estes laços operam sobre estruturas de dados bidimensionais das propriedades físicas. Portanto, podem ser paralelizadas com diretivas `parallel for` de 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 também foram paralelizadas utilizando diretivas OpenMP. 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 ia sendo alterado ou paralelizado. Algumas dessas rotinas encontravam-se na etapa de inicialização do código.

A rotina `rhs_euler` do arquivo `rhs.f90` também possuía um tempo de execução expressivo. Observou-se que na rotina eram feitas apenas alocações de memórias temporárias desnecessárias, as quais foram removidas. Porém, as chamadas de subrotinas necessárias para o processamento dos dados foram mantidas.

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 ($T_{paralelo}$).

Para avaliar o custo de iteração para cada teste realizado com um dado número de threads, utilizou-se da fórmula abaixo (1), onde `M` representa a média de tempo e `N` o número de iterações utilizadas. Todos os experimentos foram realizados considerando 16 iterações da etapa iterativa principal da aplicação, embora em um experimento real o

número de iterações necessárias seja maior (ex. 8000 iterações).

Esta simplificação possibilita avaliar experimentalmente o paralelismo da aplicação, 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. Para fins de testes não é viável e necessário utilizar um número de iterações maior, visto que a carga de computação não muda. Tal simplificação possibilitou um grande número de testes, para verificar o impacto em cada alteração realizada no código.

$$Custo\_Iteração = \frac{M\_Custo\_Total - M\_Custo\_Inicialização}{N} \quad (1)$$

Todos os testes foram realizados em uma *workstation* da Universidade Federal do Pampa (UNIPAMPA), campus Alegrete, que possui a seguinte configuração: dois processadores Intel® Core™ Xeon CPU E5-2650, com 2,0 GHz de frequência, 8 núcleos e 16 *threads*, e 128 GB de memória RAM. Foi utilizado o sistema operacional Ubuntu na versão 20.04 de 64 bits. Os resultados são calculados a partir da média de 10 execuções. Os testes foram feitos considerando o compilador `pgf90` versão 20.9 do pacote HPC_SDK da NVIDIA, considerando as diretivas `-O3 -mp`. Outros testes também foram feitos com o compilador `gfortran` com diretivas similares. No entanto, o tempo de execução sequencial e paralelo foram maiores para este compilador.

3. Resultados

A Figura 1 apresenta a média dos valores de cada uma das 10 execuções para os tempos de inicialização e custo por iteração em segundos. Além do caso sequencial, foram medidos os tempos usando 2, 4, 8, 16 e 32 *threads*. O tempo de inicialização e o custo por iteração no caso sequencial é de em torno de 1,5 segundos em cada caso. Ao se utilizar duas *threads*, nota-se que o custo por iteração foi reduzido a metade. Utilizando-se 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 1 apresenta o tempo total de execução de cada simulação usando 16 iterações do algoritmo, e os ganhos de performance separados para a etapa de inicialização e iterativa, da execução sequencial e paralelas, variando o número de *threads* de 2 a 32. O tempo total de execução sequencial foi de aproximadamente 25s. O melhor resultado foi obtido ao utilizar 16 *threads*, ou seja, o máximo de cores físicos disponíveis na *workstation* utilizada. Assim, obteve-se o ganho de desempenho de 74,97% para a inicialização e 87,79% no custo por iteração. 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. Já em relação ao tempo de inicialização, o limite de aceleração é menor, uma vez que a leitura de arquivo de entrada e alocação de memória são tarefas seriais. Apenas alguns preenchimentos das estruturas de dados é que podem ser feitos em paralelo.

4. Considerações Finais

Os algoritmos que realizam simulações correspondem a uma gama de estratégias eficientes para a solução de problemas de diversas áreas, desde uma simples análise de dados até

Figura 1. Tempo de inicialização e custo médio de cada iteração

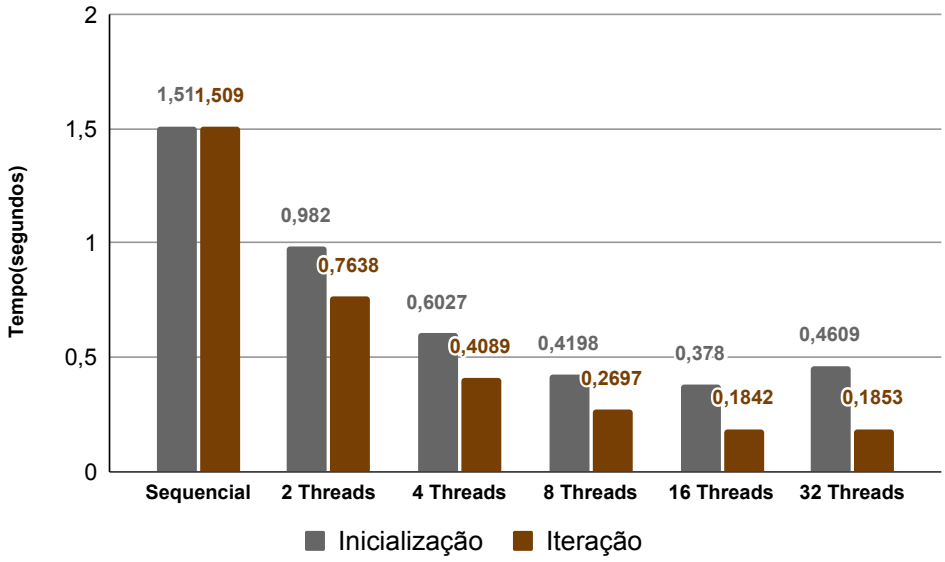

Tabela 1. Custo Total da Execução e Ganho de Desempenho

Modo Execução	Tempo	Inicialização	Iteração
Sequencial	25,67s	-	-
2 Threads	13,202s	34,92%	49,38%
4 Threads	7,146s	60,09%	72,90%
8 Threads	4,735s	72,18%	82,13%
16 Threads	3,326s	74,97%	87,79%
32 Threads	3,427s	69,48%	87,72%

problemas complexos como a simulação de uma câmara de combustão. O objetivo principal deste trabalho foi a paralelização de partes de um algoritmo de simulação de uma câmera de combustão, onde se obteve ganhos de desempenho expressivos nos diversos testes realizados do algoritmo, chegando a 87,79% de eficacia usando 16 *threads*.

Como trabalhos futuros buscaremos complementar o algoritmo com mais otimizações paralelas e também utilizar arquitetura GPU. Deseja-se aumentar o poder computacional para realizar testes mais complexos, visando ainda mais a otimização e ganho de desempenho do código.

Referências

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).

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)*.