Simulação de Modelos de Difusão de Contaminantes com Diferentes Métodos de Programação Distribuída Análise de Desempenho das Versões Seriais x Paralelas

Arthur L. A. Silva, Evandro K. Kayano, Yuri S. Bastos

[1]Instituto de Ciência e Tecnologia – Universidade Federal de São Paulo (UNIFESP)

1. Introdução

A velocidade de execução de cálculos é uma das maiores qualidades dos computadores. Tarefas como simulações que demorariam diversas horas para um humano podem ser executadas por uma máquina em poucos segundos. Entretanto, mesmo na execução de máquina, uma tarefa de complexidade muito grande ou com um número grande de amostragem pode ter um desempenho não desejado para tarefas que exijam respostas rápidas. Para isso, pode-se muitas vezes dividir a execução desta tarefa, paralelizando-a em *threads* ou processos.

Este trabalho aborda um problema de simulação de contaminantes em um corpo d'água e, devido ao alto custo computacional dessas análises, a computação paralela se torna uma alternativa eficiente, utilizando tecnologias como OpenMP, CUDA e OpenMPI.

Com isso, foram comparados os desempenhos dessas abordagens em termos de tempo de execução, escalabilidade e eficiência, visando orientar a escolha adequada para simulações ambientais de larga escala.

2. Desenvolvimento

Foi utilizado como base para este projeto o código de simulação em execução serial em C disponibilizado pelos professores. Este programa base discretiza uma equação diferencial de difusão/transporte de contaminantes em um copo d'água utilizando diferenças finitas em uma grade bidimensional. Para isso cada valor da matriz atualiza seu valor a cada iteração com base nos valores de posições vizinhas, e as iterações indicam o avanço do tempo.

Por ser uma estrutura de dados grande (matriz de 2000x2000), e serem feitas 500 iterações temporais, a execução serial do código pode não ter o desempenho desejado.

Para dividir o processamento desta tarefa entre múltiplos processos, o programa em MPI recebe como parâmetro o número de processos a serem utilizados com *MPI_Comm_size()*. Já na implementação em OpenMP, o programa recebe o número máximo de *threads*. Por fim, na versão de CUDA recebe a quantidade de *threads* por bloco. Estes valores são utilizados em cada versão para calcular a quantidade de linhas da matriz que cada processo ou thread é responsável por calcular.

Nas três implementações a cada 100 iterações temporais, é impresso o valor da difusão média. Este valor, caso seja igual na versão serial e paralelizada, pode comprovar a compatibilidade entre as versões serial e paralelizadas.

2.1. OpenMP

Para ser feita a paralelização em OpenMP, as iterações dos laços de linhas (e colunas, por estarem em um laço interno) foram divididas igualmente entre as *threads* a serem utilizadas. Para isso, foi utilizada a diretiva *#pragma omp parallel for collapse(2)* no primeiro conjunto de *for* e *#pragma omp parallel for collapse(2) reduction(+:difmedio)* no segundo conjunto. A cláusula *reduction(+:difmedio)* faz com que a variável *difmedio*, responsável por armazenar a média da difusão em uma iteração, combine os resultados parciais de cada thread em um único valor global para evitar condições de corrida na operação de somatório da difusão média.

2.2. CUDA

Em CUDA, foi necessário alocar no *host* um espaço suficiente para a matriz, para que seja guardada a matriz resultado. No device a alocação de uma matriz de *input*, outra de *output* e uma variável *float* (usada para guardar o valor de difusão média) com o *CudaMalloc*. São feitas então 500 iterações temporais, e dentro destas são chamadas as funções *update_matriz* que faz o cálculo da difusão sobre a matriz e *update_dif _matriz* que faz o cálculo da difusão média nesta iteração usando *atomicAdd*. Por fim os valores alocados são liberados usando *free* e *cudaFree*.

2.3. OpenMPI

Na implementação em MPI, como o cálculo de difusão é feito utilizando elementos vizinhos da matriz, então é necessário que haja troca de mensagens entre os processos para que as linhas calculadas em um processo possam ser lidas pelos seus processos de *rank* imediatos. Para que essa troca de mensagens seja possível, são utilizadas as funções não-bloqueantes *MPI_Isend()* para enviar dados e *MPI_Irecv()* para receber dados, e então é chamada a função *MPI_Waitall()* para garantir que haja sincronização dos dados das requisições (até dois pares de *MPI_Isend()* e *MPI_Irecv()*) feitas e que os cálculos de difusão média não utilizem valores não atualizados. Com isso, cada processo consegue calcular a difusão dos elementos de suas linhas correspondentes.

3. Especificações da máquina utilizada para testes

Para a realização dos testes deste trabalho, foi utilizado um processador *AMD Ryzen 7 5700u @ 1.8GHz* [8 núcleos reais e 16 *threads*] nas versões de OpenMP e OpenMPI. Uma NVIDIA Tesla V4 de arquitetura Turing, 2.560 CUDA Cores, 320 Tensor Cores e 16 GB de VRAM GDDR6 foi utilizada para a versão de CUDA utilizando o ambiente gratuito do Google Colab.

4. Resultados

4.1. Tempos de Execução Serial e Concorrente

As tabelas abaixo demonstram os tempos de execução, speedup e eficiência para cada cenário de execução em relação à quantidade de processos ou threads em cada implementação:

Threads	Tempo (s)	Speedup	Eficiencia
1	21,6584	1,000000	100,00%
2	17,0364	1,271301	63,57%
4	7,9932	2,709603	67,74%
8	6,9052	3,136535	39,21%
16	6,263	3,458151	21,61%

Figure 1. Threads X Tempo, Eficiência e SpeedUp - OpenMP

Threads	Tempo(s)	Speedup	Eficiencia
1	6,633510	1,000000	100,00%
2	4,784174	1,386553	69,33%
4	4,373969	1,516588	37,91%
8	4,286109	1,547677	19,35%
16	4,280073	1,549859	9,69%

Figure 2. Threads X Tempo, Eficiência e SpeedUp - CUDA

Processos	Tempo(s)	Speedup	Eficiencia
1	25,559583	1,000000	100,00%
2	16,006543	1,596821	79,84%
4	11,076580	2,307534	57,69%
8	7,694598	3,321757	41,52%
16	7,607713	3,359693	21,00%

Figure 3. Processos X Tempo, Eficiência e SpeedUp - OpenMPI

Também é possível traçar comparativos das métricas de tempo, speedup e eficiência com as soluções desenvolvidas entre as três implementações:

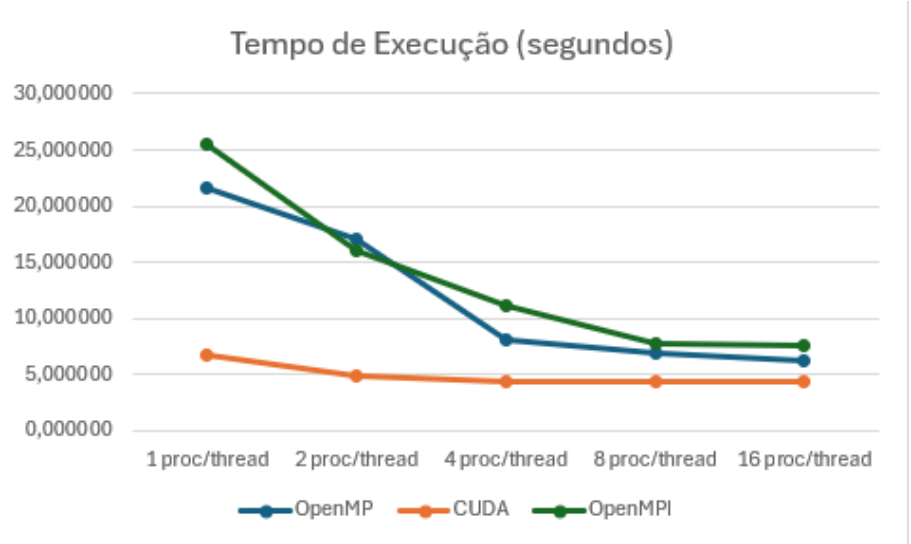

Figure 4. Tempo de Execução OpenMP X CUDA X OpenMPI

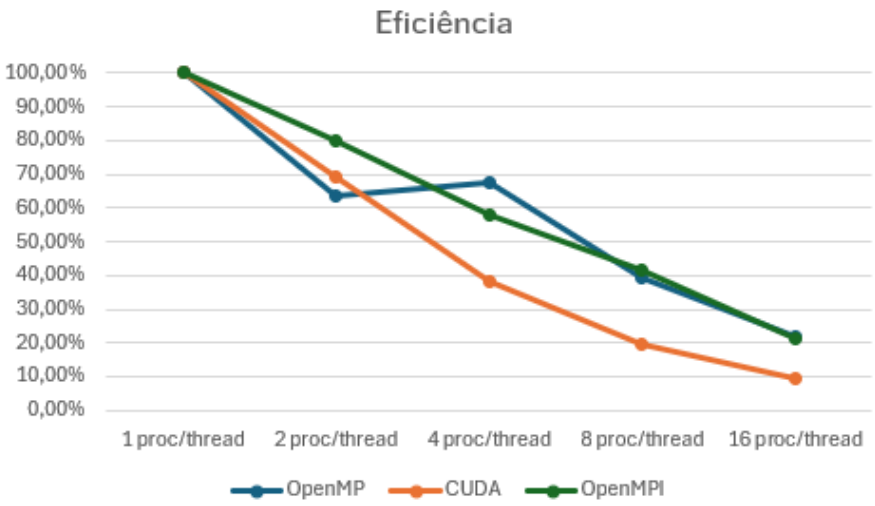

Figure 5. Eficiência OpenMP X CUDA X OpenMPI

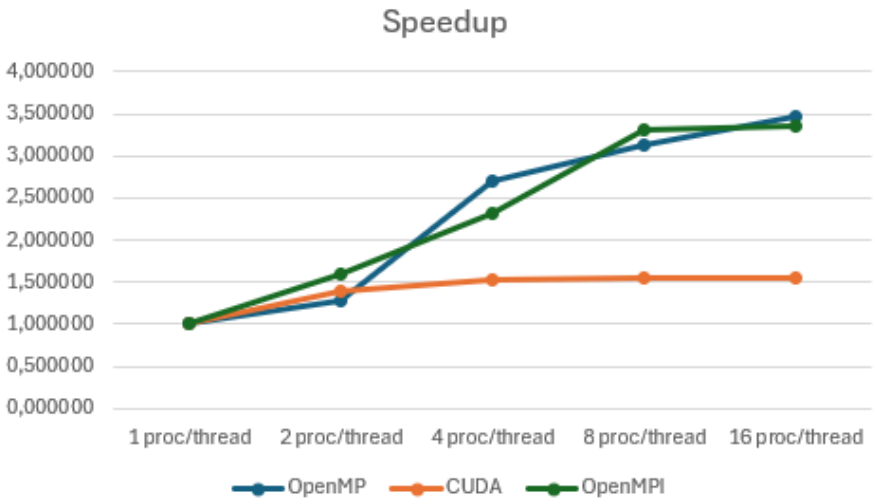

Figure 6. Speedup OpenMP X CUDA X OpenMPI

4.2. Compatibilidade

A compatibilidade do software é comprovada pelos valores de difusão média e concentração final serem equivalentes na versão serial e paralelizada das três implementações.

4.2.1. OpenMP

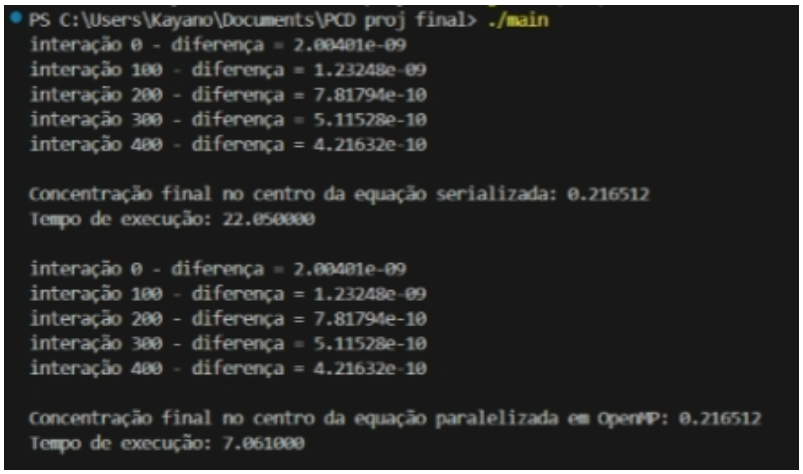

Figure 7. Comparação de diferença média na execução serial e paralela

4.2.2. CUDA

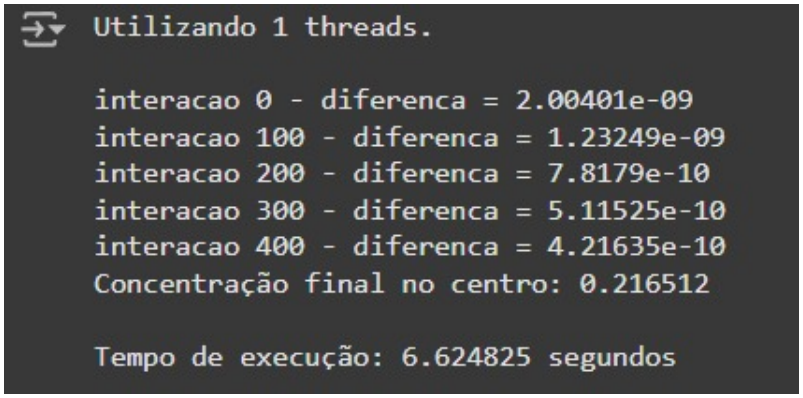

Figure 8. Diferença média na execução serial

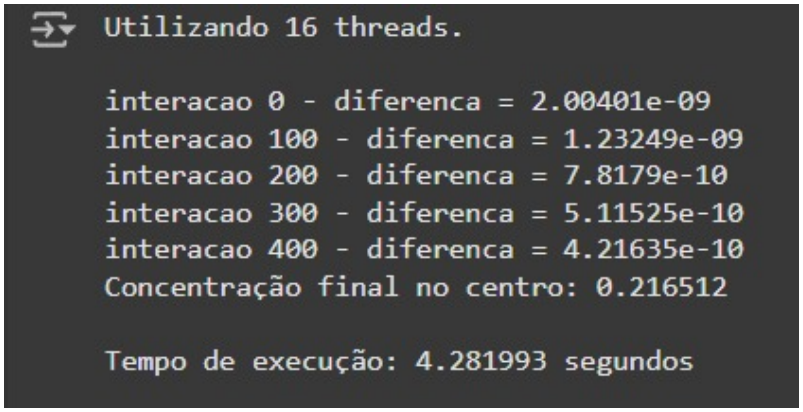

Figure 9. Diferença média na execução paralela

4.2.3. OpenMPI

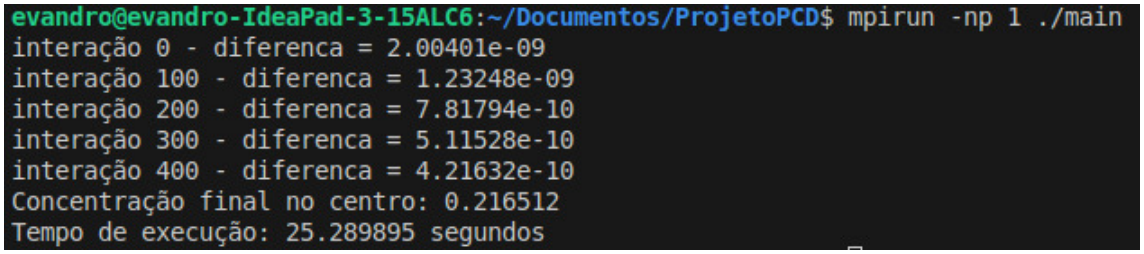

Figure 10. Diferença média e concentração final - 1 Processo

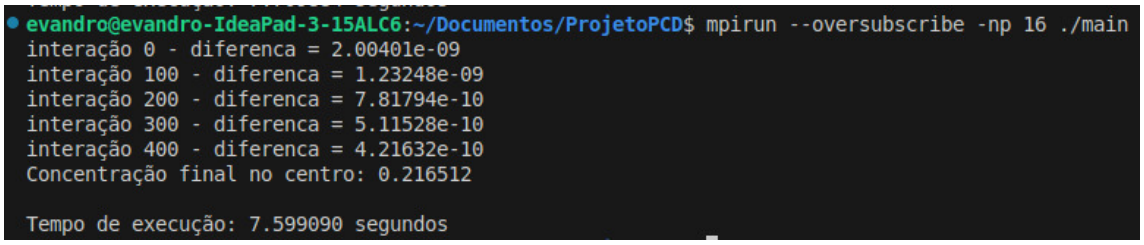

Figure 11. Diferença média e concentração final - 16 Processos

5. Conclusão

A análise comparativa entre OpenMP, CUDA e OpenMPI para a simulacão do modelo utilizado nesse estudo revela diferenças significativas na escalabilidade e eficiência de cada abordagem. Observamos que o CUDA apresenta o menor tempo de execução tanto serial quanto nas paralelizadas, fato que se dá devido ao uso de outra máquina e com sua parte mais custosa sendo rodada na GPU (NVIDIA Tesla T4). Apesar de seu menor tempo de execução, esta implementação teve a menor eficiência e speedup a partir de 4 threads.

Fazendo uma comparação entre as soluções com OpenMP e OpenMPI, é possível notar que de forma geral ambas têm um desempenho parecido, com o OpenMPI sendo levemente melhor com 2 processos que o OpenMP com 2 threads, e levemente pior que o OpenMP nas demais condições.

Para trabalhos posteriores seria interessante fazer os testes de OpenMP e OpenMPI com uma CPU com maior poder computacional à fim de conseguir menor tempo de execução serial (e, consequentemente, paralelizada). Desta forma, ao dividir a carga entre os processos/threads, talvez seria possível obter um tempo de execução menor que o CUDA, devido à baixa eficiência encontrada nesta solução.