Análise de Desempenho Paralelo de Modelos de Difusão de Contaminantes em Água

Eduardo Verissimo Faccio, Pedro Figueiredo Dias,
Pedro Henrique de Oliveira Masteguin

[1]Instituto de Ciência e Tecnologia – Universidade Federal de São Paulo (UNIFESP)
São José dos Campos – SP – Brasil

{verissimo.eduardo,pedro.figueiredo,p.masteguin}@unifesp.br

1. Introdução

A contaminação de corpos d'água, tais como lagos e rios, é um grande desafio ao meio ambiente e a saúde pública, sendo preciso entender a dispersão do contaminante no ambiente para poder realizar qualquer intervenção de mitigação. Dessa forma, este trabalho foca na modelagem numérica da difusão de poluentes em uma matriz bidimensional, utilizando o método de diferenças finitas para aproximar a equação de difusão discreta:

$$C_{i,j}^{t+1} = C_{i,j}^{t} + D \cdot \Delta t \cdot \left(\frac{C_{i+1,j}^{t} + C_{i-1,j}^{t} + C_{i,j+1}^{t} + C_{i,j-1}^{t} - 4 \cdot C_{i,j}^{t}}{\Delta x^2} \right) \tag{1}$$

Nesta equação, $C_{i,j}^{t}$ representa a concentração do contaminante na célula (i, j) no instante t, D é o coeficiente de difusão, Δt o intervalo de tempo discreto e Δx o espaçamento espacial. O objetivo principal é desenvolver uma simulação que modele a difusão de contaminantes aplicando programação paralela para acelerar os cálculos e analisar o comportamento dos poluentes ao longo do tempo. Serão comparadas as versões sequencial e paralela do algoritmo, utilizando OpenMP, CUDA e MPI para explorar o processamento simultâneo em múltiplos núcleos e dispositivos. Os resultados serão validados por meio de mapas de calor, gráficos de speedup e eficiência, além da comparação das matrizes geradas. Este estudo demonstra como técnicas de programação concorrente e distribuída podem otimizar simulações numéricas complexas, reforçando os conceitos aprendidos na disciplina e demonstrando sua aplicação prática no desenvolvimento de soluções eficientes.

2. Implementação do Algorítimo

2.1. Código Sequencial

O código sequencial implementa a solução numérica da equação de difusão usando uma abordagem serial. Utilizando-se do método de diferenças finitas, é simulado a dispersão de uma substância em uma matriz bidimensional. Cada célula da matriz representa a concentração de uma substância em um ponto do espaço.

O cálculo é realizado em um laço de repetição que itera sobre todas as células da matriz. A atualização de cada célula depende da média das concentrações dos seus vizinhos imediatos e de parâmetros físicos como coeficiente de difusão, o intervalo de tempo Δt e o espaçamento espacial Δx.

2.2. Código Paralelo

O código paralelo implementa uma versão desenvolvida em OpenMP do mesmo algorítimo sequencial, para distribuir o trabalho entre múltiplos núcleos do processador. Isso é alcançado utilizando-se das diretivas específicas da biblioteca na estrutura do código sequencial.

- #pragma omp parallel for collapse(2): É a diretiva que divide automaticamente os loops de cálculo entre múltiplos threads. Cada thread é responsável por atualizar uma parte distinta da matriz, permitindo que várias partes do cálculo sejam realizadas simultaneamente. O uso de collapse(2) indica que o compilador deve tratar múltiplas laços aninhados com um único para fins de paralelismo. No caso do código de difusão, os dois laços de repetição aninhados serão linearizados.
- #pragma omp parallel for reduction(+:difmedio) collapse(2): Essa diretiva assim como a anterior executa as iterações do laço de repetição de maneira paralela, com o adicional de reduções, que garantem que a operação de somatório que monitora a convergência do algorítimo seja realizada de maneira segura entre as diferentes threads que acessam as variáveis dessa operação.
- omp_set_num_threads(int num_threads): Permite configurar dinamicamente o número de threads a serem utilizados pelo código, sendo extremamente útil para realizar as verificações de desempenho.

2.3. Interface Python e ferramenta CMake

O projeto utiliza o CMake como sistema de compilação, definindo processos e dependências (como a biblioteca OpenMP) por meio de um arquivo de configuração. Para evidenciar as diferenças de desempenho entre as implementações sequencial e paralela, desativamos otimizações do compilador usando flags específicas, pois otimizações como a vetorização automática no código sequencial reduziram inicialmente essa diferença. Assim, desabilitar essas otimizações permitiu uma execução mais direta, destacando as diferenças de desempenho entre as abordagens.

Além disso, implementamos uma interface Python usando o módulo ctypes para carregar a biblioteca de difusão em C e mapear suas funções e, com isso, permite executar os códigos sequencial e paralelo de forma flexível, facilitando a integração com métodos de análise desenvolvidos em notebooks Jupyter, onde os resultados são visualizados. Criar essa interface segue a abordagem de outras grandes bibliotecas da linguagem, combinando a facilidade de uso de um ambiente de alto nível com a eficiência de soluções implementadas em linguagens como C. Um exemplo de biblioteca que utiliza esse método é o NumPy, que oferece operações matemáticas de alto desempenho por meio de códigos otimizados em C.

3. Resultados

Nesta seção, apresentamos os resultados obtidos de nossa implementação. Inicialmente, analisamos a equivalência lógica entre os códigos sequencial e paralelo, considerando possíveis erros que podem surgir na paralelização, como condições de corrida ou inconsistências de sincronização. Em seguida, ilustramos, por meio de mapas de calor, a atualização dos valores da matriz ao longo do tempo. Por fim, realizamos uma análise comparativa dos tempos médios de execução, *speedup* e eficiência entre as duas versões.

3.1. Validação da Implementação

Para assegurar a correção das duas implementações, verificamos em cada iteração se os valores presentes em cada célula da matriz são idênticos. Dessa forma, o resultado na última iteração deve ser o mesmo em ambas as versões.

Por meio desse procedimento, utilizando a interface Python em conjunto com um Jupyter Notebook, comprovamos que as duas soluções produzem resultados idênticos. Isso era esperado, pois no código paralelo não ocorrem condições de corrida, uma vez que a escrita não é realizada na mesma região de memória das leituras, tornando o processamento de cada célula pelas *threads* independente.

3.2. Validação da Implementação

Para ilustrar o funcionamento da implementação, foram gerados mapas de calor, representado pela Figura 1, nos quais cada ponto de uma matriz 50x50 é representado por uma cor distinta. Cores escuras correspondem a valores próximos de um, indicando alta concentração do contaminante, enquanto cores claras representam valores próximos de zero, indicando baixa presença de contaminação.

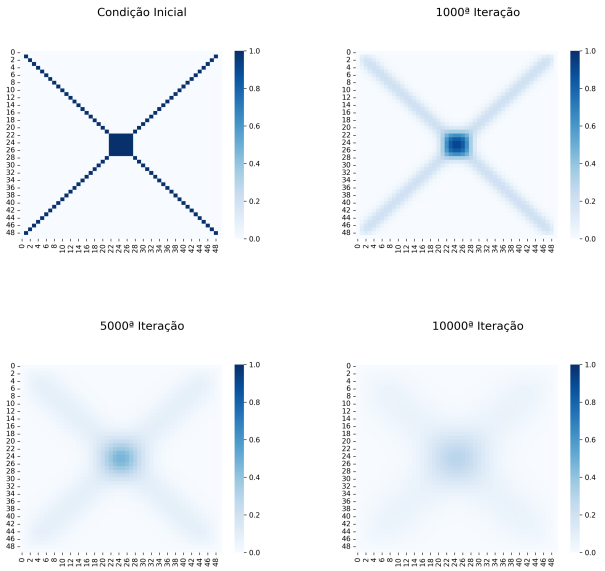

Figura 1. Mapa de calor em quatro instantes distintos da simulação.

Analisando a progressão dos mapas de calor, observamos que o comportamento faz sentido no contexto da solução proposta. Inicialmente, o contaminante é adicionado com alta concentração nas diagonais e no centro da matriz, evidenciado pelas regiões azuis escuros. Com o avanço das iterações, o contaminante começa a se difundir para as regiões adjacentes, aumentando gradativamente a luminosidade nessas áreas e diminuindo nos pontos de concentração inicial. Na última iteração, a concentração se distribui uniformemente pela matriz, com valores próximos entre si.

3.3. Análise de Desempenho

A análise de desempenho foi realizada em um computador *desktop* com as especificações apresentadas na Tabela 1. Ademais, as especificações dos parâmetros do problema foram incluídas na Tabela 2.

Tabela 1. Tabela de especificação de Hardware

Especificações	Detalhes
Processador	Intel i7–4790 @ 3.60GHz
Núcleos / Lógicos	4 / 8
Memória RAM	8 GB
Sistema Operacional	Ubuntu 22.04.05 (via WSL)

Tabela 2. Tabela de especificação da Simulação

Especificações	Detalhes
Dimensão da Matriz (N x N)	2000 x 2000
Número de Iterações	500
Distribuição Inicial	Alta concentração no centro
Coeficiente de Difusão	0.1
Δt	0.01
Δx	1.0

Para obter valores mais consistentes e minimizar influências externas, como outros programas em execução, cada teste foi executado quinze vezes e, assim, calculamos o tempo médio gasto e seu desvio padrão. O *speedup* é calculado dividindo-se o tempo de execução sequencial pelo tempo de execução paralelo correspondente, enquanto a eficiência é determinada ao dividir o *speedup* pelo número de *threads* utilizados.

Tabela 3. Tabela de comparação de desempenho entre o código sequencial e o paralelo utilizando OpenMP.

Nº Threads	Tempo	Speedup	Eficiência
1	26.22 ± 2.09	1.0	100%
2	14.59 ± 1.49	1.80	89,88%
4	10.60 ± 1.43	2.47	61.82%
8	8.52 ± 1.32	3.08	38.48%
16	8.29 ± 1.29	3.16	19.77%
32	8.67 ± 1.43	3.03	9.45%

O gráfico de *speedup* (Figura 2) mostra uma tendência de estabilização em torno do valor três, enquanto o *speedup* linear ideal apresenta valores significativamente maiores, o que pode sugerir que a escalabilidade da aplicação é limitada, possivelmente devido a *overheads* de sincronização ou ao fato de que o problema não é suficientemente grande para aproveitar eficientemente um número maior de *threads*. Além disso, o desempenho

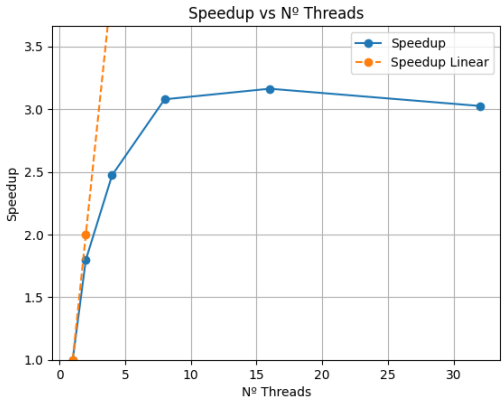

Figura 2. Gráfico do *speedup* por número de *threads*

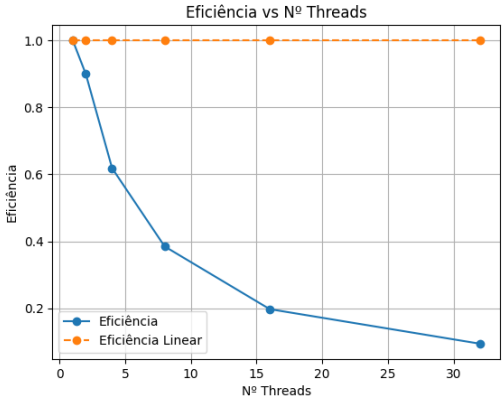

Figura 3. Gráfico da eficiência por número de *threads*

pode estar sendo afetado pela latência de memória ou pela arquitetura do processador utilizado e, consequentemente, não há grandes vantagens em utilizar um número elevado de *threads* para resolver este problema específico.

4. Conclusão

Por meio de implementações sequenciais e paralelas utilizando OpenMP, foi validada a correção dos resultados e analisou-se o impacto do paralelismo no tempo de execução. Os resultados obtidos mostraram que, embora o paralelismo traga ganhos significativos em relação ao código sequencial, há limitações na escalabilidade, possivelmente devido a fatores como *overheads* de sincronização e restrições arquiteturais do hardware utilizado. Assim, este trabalho reforça os conceitos fundamentais aprendidos em programação concorrente e distribuída, evidenciando as vantagens práticas da paralelização em termos de desempenho e eficiência em simulações numéricas.

Referências

Crank, J. (1979). *The Mathematics of Diffusion*. Oxford science publications. Clarendon Press.

Rauber, T. and Runger, G. (2013). *Parallel Programming*. Springer.