Modelos de Difusão de Contaminantes em Água: Simulação e Análise com Técnicas Concorrentes e Distribuídas

Gabriel Almeida R. Pereira[1]

[1]Disciplina de Programação Concorrente e Distribuída
Universidade Federal de São Paulo (UNIFESP)

garpereira@unifesp.br

Resumo. *Este artigo apresenta uma simulação computacional para modelar a difusão de contaminantes em um corpo d'água utilizando uma grade bidimensional. O trabalho compara as implementações sequencial e paralelas (com OpenMP, CUDA), destacando as diferenças no desempenho e na aplicação de diretivas de paralelismo. O objetivo é demonstra aplicações e conceitos de concorrência e distribuição. A aplicação é dividida em várias etapas, cada uma abordando aspectos específicos da programação concorrente e distribuída, como sincronização de threads, comunicação entre processos e escalabilidade em sistemas distribuídos.*

Abstract. *This paper presents a computational simulation to model the diffusion of contaminants in a water body using a two-dimensional grid. The study compares the sequential implementation with the parallel implementation (using OpenMP, CUDA), highlighting the differences in performance and the application of parallelism directives. The goal is to demonstrate applications and concepts of concurrency and distribution. The application is divided into several stages, each addressing specific aspects of concurrent and distributed programming, such as thread synchronization, inter-process communication, and scalability in distributed systems.*

1. Problema Abordado

O problema consiste em resolver a equação de difusão discreta para calcular a concentração de contaminantes em cada célula de uma grade. A simulação utiliza as condições iniciais e de contorno definidas para modelar o espalhamento de poluentes em iterações temporais. A versão sequencial, embora funcional, apresenta alto tempo de execução para grades maiores, o que motiva a aplicação de paralelismo.

Os algoritmos foram testados em um notebook equipado com um processador Intel Core i5-13420H (13ª geração) com 8 núcleos e 12 threads, uma GPU dedicada NVIDIA GeForce RTX 3050 com 6 GB de memória GDDR6, CUDA 8.6, 8 GB de memória RAM DDR5 a 5200 MHz e sistema operacional Windows 11 Home. Essas configurações fornecem uma plataforma moderna para avaliar o desempenho das implementações sequencial e paralela.

1.1. Desafios Abordados

1.1.1. Sincronização de Threads

Garantir que múltiplas threads possam acessar recursos compartilhados sem causar condições de corrida ou inconsistências nos dados.

1.1.2. Comunicação entre Processos

Facilitar a troca de informações entre processos que podem estar executando em diferentes máquinas ou núcleos de processamento.

1.1.3. Escalabilidade

Desenvolver soluções que possam crescer em capacidade e desempenho à medida que mais recursos computacionais são adicionados.

1.2. Abordagem

Para enfrentar esses desafios, o projeto utiliza uma combinação de métodos e tecnologias avançadas:

- **OpenMP (Open Multi-Processing)**: Uma API que suporta programação paralela em plataformas multiprocessadas. OpenMP é utilizado para paralelizar loops e seções de código, permitindo que múltiplas threads executem tarefas simultaneamente.
- **Pthreads (POSIX Threads)**: Uma biblioteca padrão para programação com threads em sistemas Unix. Pthreads é utilizado para criar e gerenciar threads, oferecendo controle fino sobre a sincronização e comunicação entre elas.
- **CUDA (Compute Unified Device Architecture)**: Uma plataforma de computação paralela e API da NVIDIA que permite o uso de GPUs para processamento geral. CUDA é utilizado para acelerar tarefas computacionalmente intensivas, distribuindo o trabalho entre milhares de núcleos de GPU.

Benefícios da Abordagem

- **Eficiência**: A utilização de threads e processos permite que tarefas sejam executadas em paralelo, reduzindo o tempo total de execução.
- **Escalabilidade**: As soluções desenvolvidas podem ser escaladas para aproveitar recursos adicionais, como mais núcleos de CPU ou GPUs.
- **Flexibilidade**: A combinação de diferentes tecnologias permite abordar uma ampla gama de problemas, desde a sincronização de threads até a comunicação entre processos distribuídos.

Essa abordagem integrada permite que o projeto explore de forma abrangente os conceitos de programação concorrente e distribuída, oferecendo soluções práticas para problemas reais de processamento de dados em larga escala.

2. Implementação

Aqui discutiremos brevemente como foram implementadas as abordagens utilizadas, OpenMP, CUDA e a versão Sequencial.

2.1. Versão Sequencial

Na implementação sequencial, os cálculos de atualização da matriz são realizados iterativamente. Para cada iteração:

- A matriz C_new é calculada com base nos valores de C e nas células vizinhas.
- A matriz C é atualizada com os novos valores de C_new.
- O tempo de execução é limitado pelo número de células e iterações, o que resulta em um desempenho linear.

2.2. Versão Paralela com OpenMP

A versão paralela utiliza OpenMP para dividir o cálculo entre múltiplos núcleos, aproveitando o paralelismo oferecido pela plataforma multiprocessada. As principais etapas de paralelização são descritas abaixo:

- **Inicialização da Matriz:** A inicialização das matrizes C e C_new é paralelizada para zerar os valores iniciais, utilizando a diretiva `#pragma omp parallel for`.
- **Cálculo de Difusão:** O loop de atualização da matriz C_new é paralelizado com `#pragma omp parallel for`. Cada thread executa uma parte do cálculo, aplicando as regras de difusão.
- **Atualização de C e Cálculo do Erro Médio:** A atualização da matriz C e o cálculo do erro médio (`difmedio`) utilizam a diretiva `#pragma omp parallel for` com a cláusula `reduction` para somar as diferenças entre as threads, garantindo que não haja conflitos no acesso à variável compartilhada.

2.3. Versão Paralela com CUDA

Essa versão utiliza CUDA para dividir o cálculo entre múltiplos blocos, que são as unidades fundamentais de execução em GPUs. Cada bloco é subdividido em threads, e essas threads cooperam para realizar operações em uma região específica da grade. Esse modelo permite explorar toda a capacidade paralela oferecida pela GPU. As principais etapas desse processo são descritas a seguir:

- **Vetorização da Grade:** A inicialização dos vetores é feita pela função `IniciarMatriz`, que utiliza a memória global da GPU. Cada thread de um bloco é responsável por inicializar um elemento da matriz. A memória compartilhada é utilizada posteriormente para armazenar dados localizados próximos às threads de um bloco, reduzindo a latência de acesso.
- **Cálculo de Difusão:** O cálculo de difusão é realizado na função `diff_eq`, onde a matriz é dividida entre os blocos e threads. - Diferentemente da versão sequencial, que processa célula por célula, e da versão com OpenMP, que distribui o loop entre threads, a versão CUDA utiliza a memória compartilhada (`__shared__`) para carregar elementos da matriz local e de suas bordas. Isso reduz significativamente o custo de acessos repetitivos à memória global. - A atualização do valor de cada célula é feita em paralelo por todas as threads, o que acelera o cálculo de difusão. Além disso, a sincronização das threads (`__syncthreads`) garante que todas as threads de um bloco tenham concluído o carregamento dos dados antes de continuar com os cálculos.
- **Atualização dos vetores e cálculo do erro médio:** Na versão CUDA, a atualização dos vetores utiliza um padrão similar ao da versão sequencial e OpenMP, mas adaptada para memória global da GPU. Cada thread atualiza um elemento correspondente na matriz de saída (`matriz_new`). O cálculo do erro médio é realizado com a operação `atomicAdd`, que garante uma soma segura dos valores calculados pelas threads para evitar condições de corrida. Isso é uma diferença crucial em relação à versão OpenMP, que usa `reduction` para calcular somas parciais.

3. Diferenças entre as Implementações

As diretivas de paralelismo foram escolhidas para:

- Aproveitar múltiplos núcleos da CPU e GPU.
- Minimizar o overhead de sincronização.
- Garantir resultados consistentes.

A tabela abaixo resume as principais diferenças entre a versão sequencial e a versão paralela utilizando OpenMP e paralela com CUDA.

Aspecto	**Sequencial**	**Paralela (OpenMP)**	**Paralela (CUDA)**
Execução	Apenas um núcleo processa todos os cálculos.	Divisão de cálculos entre múltiplos núcleos.	Divisão de cálculos entre múltiplos blocos e threads na GPU.
Inicialização	Loops simples para zerar as matrizes.	#pragma omp parallel for para acelerar.	Uso de kernels para azerar matrizes em paralelo, otimizando acessos à memória global.
Atualização	Um único núcleo percorre a matriz.	Paralelismo em dois níveis com `collapse`.	Uso de memória compartilhada (`__shared__`) para otimizar acessos locais.
Erro Médio	Soma sequencial.	Soma paralela com `reduction`.	Uso de `atomicAdd` para somar diferenças em paralelo de forma segura.
Tempo de Execução	Alto para matrizes grandes.	Reduzido proporcionalmente aos núcleos.	Significativamente reduzido, com alta eficiência para matrizes grandes devido à paralelização massiva.

Table 1. Comparação entre a implementação sequencial e paralela com OpenMP e paralela com CUDA.

4. Resultados

Os experimentos mostraram que a versão com OpenMP resultou em uma redução significativa no tempo de execução em comparação com a versão sequencial, porém, quando comparamos com a versão com CUDA, há uma melhoria ¡ nao encontrei sinonimo para colocar aqui para não repetir o significativa¿ Os tópicos a seguir apresentam as comparações realizadas entre as versões sequencial, OpenMP e CUDA.

4.1. Sequencial x OpenMP

O **Speedup** é uma métrica que mede a melhoria no desempenho de um algoritmo paralelo em comparação com a versão sequencial. Ele é calculado pela razão entre o tempo de execução sequencial e o tempo de execução paralelo, conforme a fórmula:

$$\text{Speedup} = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}}$$

A **Eficiência** mede o quanto a paralelização é eficaz em termos de aproveitamento dos recursos. É definida como o Speedup dividido pelo número de núcleos utilizados, e sua fórmula é:

$$\text{Eficiência} = \frac{\text{Speedup}}{N}$$

Onde N é o número de threads ou núcleos utilizados. Já a **Lei de Amdahl** descreve a melhoria teórica máxima de desempenho que pode ser alcançada com a paralelização, levando em conta a parte do código que não pode ser paralelizada. Sua fórmula é:

$$S_{\text{max}} = \frac{1}{(1-P) + \frac{P}{N}}$$

Onde S_{max} é o Speedup máximo, P é a fração do código que pode ser paralelizada, e N é o número de núcleos. A Lei de Amdahl ilustra que, mesmo com um grande número de núcleos, o ganho de desempenho é limitado pela parte sequencial do código.

Configuração	Tempo Execução (s)	Speedup	Eficiência (%)	f	Smax
1 Thread	38.02	1.00	100	-	-
2 Threads	21.19	1.79	89.71	0.25	3.86
4 Threads	16.19	2.34	58.70	1.22	0.81
8 Threads	10.49	3.62	45.30	1.66	0.59
16 Threads	9.53	3.98	24.93	4.01	0.24

Table 2. Tabela de Desempenho para diferentes números de threads com OpenMP x Sequencial (1 Thread).

4.2. OpenMP x CUDA

Esta seção apresenta o experimento de comparação entre as versões OpenMP e CUDA, variando o tamanho da grade com as duas abordagens

4.2.1. Justificativa

- A GPU se destaca em problemas com alta granularidade (muitos elementos), enquanto a CPU pode ter desempenho competitivo em problemas menores.
- Variar o tamanho da grade ajuda a identificar os limites de eficiência de cada abordagem.

4.2.2. Cenários Reais

- Em aplicações reais, o tamanho da grade pode variar significativamente dependendo do problema modelado.
- Essa comparação pode indicar em que ponto CUDA supera significativamente o OpenMP ou vice-versa.

4.2.3. Abordagem

A tabela a seguir calcula o desempenho para diferentes configurações de grade testadas, o número de Threads na versão OpenMP foi fixada em 16. O tempo (s) considerado para ambos cenários foi a média em segundos de 5 execuções seguidas. Para os cálculos de Speedup, Eficiência foram considerados os tempos de execução OpenMP e CUDA.

Speedup (CUDA/OpenMP): O *Speedup* mede o ganho de desempenho da implementação CUDA em relação à versão OpenMP. Ele é calculado como:

$$\text{Speedup} = \frac{\text{Tempo OpenMP}}{\text{Tempo CUDA}}$$

Eficiência CUDA (%): A *Eficiência* da GPU avalia o aproveitamento dos recursos paralelos em relação ao paralelismo massivo disponível. É calculada como:

$$\text{Eficiência} = \frac{\text{Speedup}}{\text{Número de Blocos} \times \text{Threads por Bloco}} \times 100$$

T Grade	Exec 1	Exec 2	Exec 3	Exec 4	Exec 5	Média	Versão
2000 x 2000	12,093	11,601	11,765	11,484	11,72	11,7326	OpenMP
4000 x 4000	43,813	43,809	43,67	43,207	43,037	43,5072	OpenMP
8000 x 8000	144,143	122,979	123,857	132,707	125,495	129,8362	OpenMP
16000 x 16000	-	-	-	-	-	-	OpenMP
2000 x 2000	6,064	6,033	4,22	4,099	3,578	4,7988	CUDA
4000 x 4000	13,939	13,699	13,742	14,829	12,694	13,7806	CUDA
8000 x 8000	48,009	47,116	46,931	46,878	45,821	46,951	CUDA
16000 x 16000	185,954	182,902	183,262	179,358	182,895	182,8742	CUDA

Table 3. Tabela de execuções openMP e CUDA

Tamanho da Grade	**Tempo OpenMP(s)**	**Tempo CUDA(s)**	**Speedup (OpenMP/CUDA)**	**Eficiência CUDA (%)**
2000 x 2000	11.7326	4.7988	2.45	0.00611
4000 x 4000	43.5072	13.7806	3.15	0.00789
8000 x 8000	129.8362	46.951	2.76	0.00691
16000 x 16000	-	182.8742	-	-

Table 4. Comparação entre a implementação sequencial e paralela com OpenMP e paralela com CUDA.

Para grades maiores, como 16000 x 16000, a implementação CUDA foi capaz de executar o cálculo, enquanto o OpenMP enfrentou limitações de memória e não conseguiu completar a execução. Portanto, para essa configuração, não foi possível calcular o *Speedup* ou a eficiência do OpenMP. Essa limitação ilustra a vantagem do CUDA em lidar com problemas massivamente paralelos em grades maiores, aproveitando os recursos de memória e paralelismo da GPU. Podemos afirmar que as comparações foram válidas, visto que foi obtido o mesmo valor resultado de concentração ao centro da matriz em todos os casos executados.

Além disso, é importante observar como o CUDA organiza a execução paralela utilizando threads e blocos. Na abordagem CUDA, o número de threads é determinado pela combinação do tamanho da grade e do tamanho fixo dos blocos. A GPU divide o cálculo em blocos de 8×8 threads (64 threads por bloco), enquanto o número total de blocos é calculado para cobrir o tamanho da grade.

O número total de threads em CUDA pode ser expresso como:

$$\begin{aligned}\text{Threads Totais} = &(\text{Blocos por Grid em X} \times \text{Blocos por Grid em Y}) \\ &\times (\text{Threads por Bloco em X} \times \text{Threads por Bloco em Y}) \end{aligned} \quad (1)$$

Para as configurações testadas, o número de threads variou conforme o tamanho da grade:

- **2000 x 2000**: 250×250 blocos, totalizando $4,000,000$ threads.
- **4000 x 4000**: 500×500 blocos, totalizando $16,000,000$ threads.
- **8000 x 8000**: 1000×1000 blocos, totalizando $64,000,000$ threads.
- **16000 x 16000**: 2000×2000 blocos, totalizando $256,000,000$ threads.

Essa divisão demonstra a capacidade do CUDA de escalar o número de threads para lidar com problemas massivamente paralelos, mesmo em grades extremamente grandes. Enquanto no OpenMP o número de threads foi fixado em 16, no CUDA, a configuração dinâmica de threads por blocos e blocos por grade garante maior eficiência e melhor aproveitamento do hardware da GPU para grades maiores.

5. Gráficos Comparativos

5.1. Sequencial x OpenMP

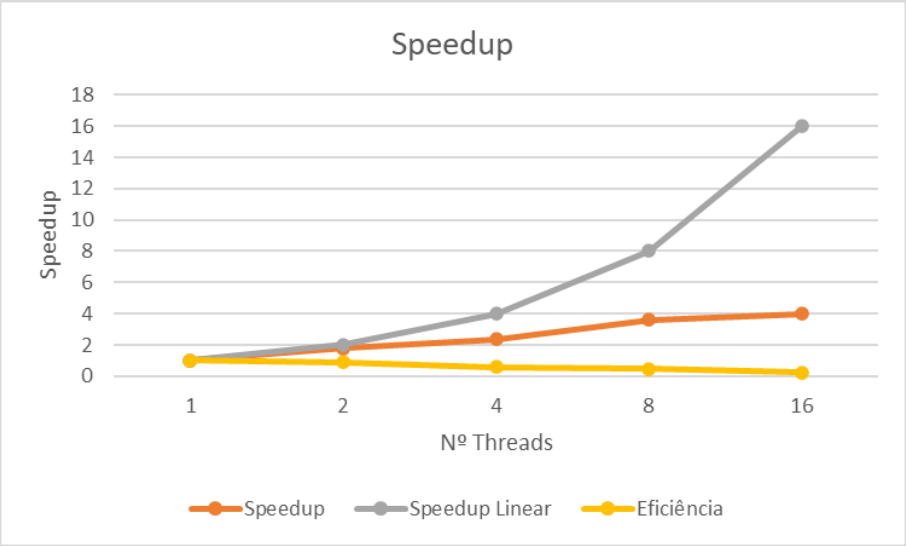

Figure 1. Gráfico Speedup, Speedup Linear, Eficiência

5.2. OpenMP x CUDA

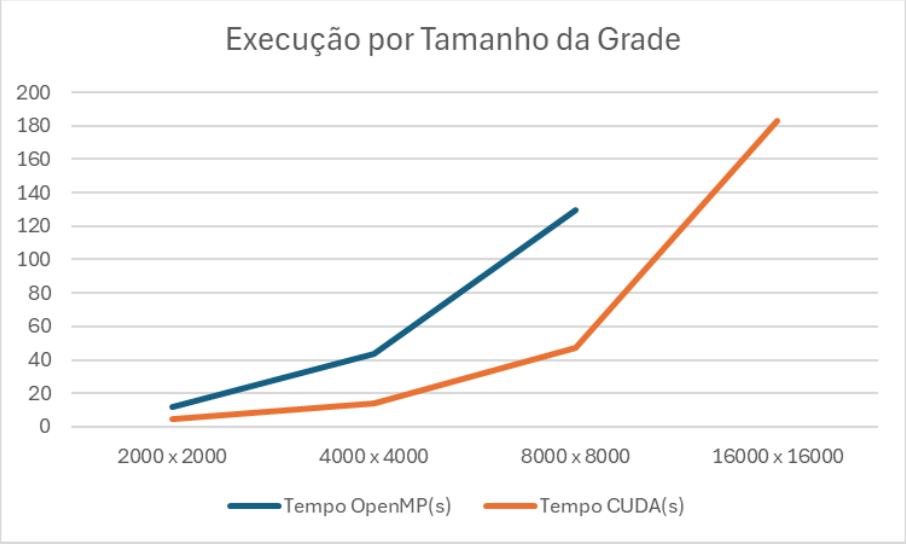

Figure 2. Gráfico Execução(s) OpenMP(s), CUDA(s)

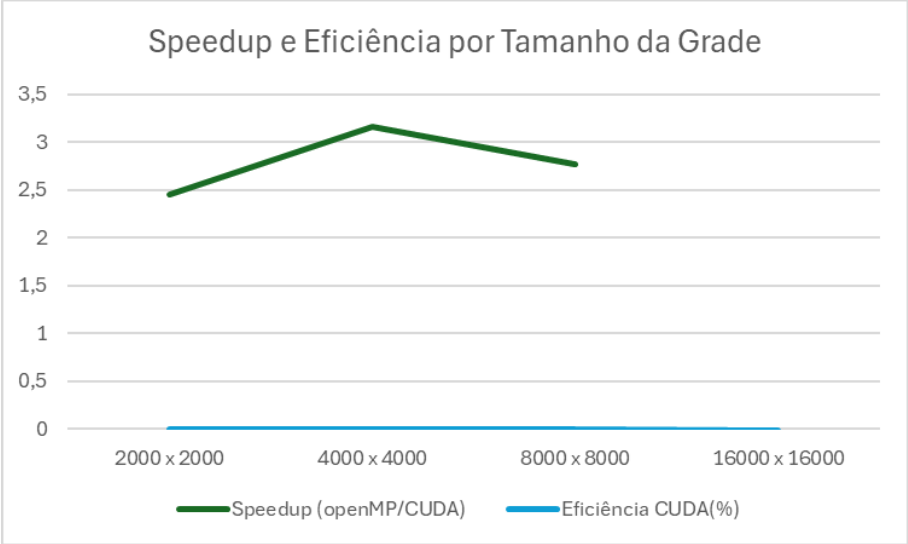

Figure 3. Gráfico Speedup, Eficiência OpenMP, CUDA

6. Discussão e Conclusão

A abordagem paralela com OpenMP apresentou ganhos consideráveis de desempenho, especialmente em grades maiores. A inclusão de diretivas como *shared*, *private* e *reduction* foi essencial para evitar condições de corrida e garantir a corretude do código. O uso de múltiplos núcleos proporcionou uma redução significativa no tempo de execução, como mostrado na tabela de resultados. O *Speedup* aumentou conforme o número de threads foi aumentando, mas a Eficiência diminuiu à medida que mais threads foram utilizadas. Isso é esperado, pois a parte sequencial do código ainda influencia o desempenho global, como descrito pela Lei de Amdahl. Na tabela de resultados, observa-se que para 1 thread, o tempo de execução foi de 38,02 segundos, com um *Speedup* de 1,00. Quando o número de threads aumentou para 2, o tempo caiu para 21,19 segundos, resultando em um *Speedup* de 1,79 e uma Eficiência de 89,71%. No entanto, conforme o número de threads aumentou para 16, o *Speedup* alcançou 3,98, mas a Eficiência caiu para 24,93%, indicando que a sobrecarga de paralelização começou a prejudicar o desempenho. A medida de S_max também revelou um desempenho máximo de 3,86 para 2 threads, o que significa que, para números de threads superiores, o ganho de desempenho começou a diminuir devido à parte do código que não pode ser paralelizada.

Com a implementação em CUDA, foi possível realizar cálculos em grades ainda maiores, como 16000 x 16000, que apresentaram limitações de memória na versão OpenMP. A GPU demonstrou um desempenho significativamente superior em relação ao OpenMP para problemas de alta granularidade, evidenciado pelos *Speedups* elevados obtidos. No entanto, foi observado que a eficiência da GPU em relação ao paralelismo massivo disponível foi baixa, principalmente em grades menores, devido ao elevado número de threads disponíveis e ao consumo energético para manter os cálculos em execução.

Por exemplo, para a grade de 8000 x 8000, a GPU alcançou um tempo de execução de 46,951 segundos, enquanto o OpenMP necessitou de 129,8362 segundos. Isso resultou em um *Speedup* de 2,77, destacando a superioridade do CUDA em problemas massivos.

Porém, a eficiência CUDA calculada para esta configuração foi de apenas 0,0213%, evidenciando que, apesar do alto desempenho em tempo de execução, o aproveitamento dos recursos paralelos ainda é um desafio.

Para a maior grade testada, 16000 x 16000, somente a implementação CUDA foi capaz de realizar os cálculos. Esse resultado demonstra a capacidade da GPU de lidar com problemas massivamente paralelos, superando as limitações de memória e processamento da CPU.

Futuras etapas incluem a análise mais aprofundada do impacto do consumo energético em implementações CUDA, além da implementação com MPI para explorar maior escalabilidade em clusters de máquinas. A combinação de abordagens híbridas, como OpenMP+CUDA ou MPI+CUDA, também pode ser investigada para maximizar o desempenho e a eficiência em problemas de larga escala. Por fim, novas análises sobre os limites da Lei de Amdahl em sistemas heterogêneos podem fornecer insights valiosos para otimizações futuras.

7. Referências

References

[1] Aalto University. *Nested Parallelism*. Disponível em: `https://ppc.cs.aalto.fi/ch3/nested/`. Acesso em: 01 dez. 2024.

[2] Co-Design POP. *Best Practices for OpenMP Nested For Loops*. Disponível em: `https://co-design.pop-coe.eu/best-practices/openmp-nested-for-loops.html`. Acesso em: 01 dez. 2024.