

# Projeto PCD: Simulação e Análise de Modelos de Difusão de Contaminantes em Água

Bruno Kenji Sato, Luiz Eduardo Casella, Nicolas Serrat

1

**Resumo.** *Nesse projeto criou-se uma simulação que modela a difusão de contaminantes em um corpo d'água (como um lago ou rio), aplicando conceitos de paralelismo para acelerar o cálculo e observou-se o comportamento de poluentes ao longo do tempo. O projeto investigou o impacto de OpenMP, CUDA e MPI no tempo de execução e na precisão do modelo.*

## 1. Características do Código OpenMP

O código realizado implementa uma simulação e análise de modelos de difusão de contaminantes em água de forma paralelizada com o uso de threads, a implementação dessas foi através da biblioteca OpenMP, que segundo [Herlihy and Shavit 2008] essa biblioteca é uma ferramenta que facilita o processo de paralelização de código em C, C++ e Fortran.

Dado o código serializado fornecido pelo professor, implementamos o uso de funções da biblioteca OpenMP, como por exemplo a função apresentada no Código 1, que define o número de threads a serem usadas nos trechos paralelizados como 4.

### Código 1. Definição do número de threads

```
1 int main() {  
2     //printf("%i", omp_get_max_threads());  
3     omp_set_num_threads(4);  
4     // Inicializar as matrizes
```

Usou-se também da diretiva pragma omp parallel for: e suas variações, essa diretiva informa o compilador que o loop seguinte deve ser executado em paralelo, criando uma região paralela onde as iterações do loop são distribuídas entre as threads criadas. Como o código fornecido apresentava muitos loops for aninhados, utilizamos principalmente da diretiva pragma omp parallel for collapse(n), que tem como função paralelizar n loops aninhados permitindo que múltiplas iterações dos loops sejam executadas simultaneamente por diferentes threads como visto no Código 2.

### Código 2. Paralelização de loops aninhados

```
1 #pragma omp parallel for collapse(2)  
2     for (int i = 0; i < N; i++) {  
3         for (int j = 0; j < N; j++) {  
4             C_new[i][j] = 0.;  
5         }  
6     }
```

Além disso, usou-se também da diretiva pragma omp parallel for reduction(+:var), que vai paralelizar um loop e realizar uma operação de redução em uma variável compartilhada entre as threads como podemos observar no Código 3, neste trecho de código a

variável `difmedio` será usada em uma operação de redução, ou seja, cada thread terá uma cópia local da variável `difmedio` e ao fim essas cópias serão somadas, em razão do `+`.

### Código 3. Paralelização de um loop com operação de redução

```
1 #pragma omp parallel for reduction(+:difmedio)
2   for (int i = 1; i < N - 1; i++) {
3       for (int j = 1; j < N - 1; j++) {
4           difmedio += fabs(C_new[i][j] - C[i][j]);
5           C[i][j] = C_new[i][j];
6       }
7   }
```

## 2. Características do Código CUDA

O programa apresentado realiza uma simulação de difusão bidimensional utilizando CUDA para aproveitar o processamento paralelo da GPU. A simulação é executada em uma grade de  $2000 \times 2000$  elementos ao longo de 500 iterações, com parâmetros configurados para um coeficiente de difusão de 0,1, um passo de tempo de 0,01 e uma resolução espacial de 1,0.

A lógica principal está no kernel CUDA `diff_eq_kernel`, que calcula o Laplaciano em cada ponto interno da grade, atualizando os valores de concentração conforme a equação discreta da difusão. Cada thread na GPU processa uma célula específica, otimizando o desempenho ao distribuir os cálculos entre milhares de núcleos.

A memória é gerida tanto no host quanto no device. No host, dois vetores são alocados para armazenar as concentrações atual e futura, ambos inicializados com zeros, exceto por uma concentração elevada no centro da grade. Na GPU, blocos de memória equivalentes são alocados, e os dados são copiados do host para o device no início da execução.

A configuração de execução em CUDA utiliza blocos de  $16 \times 16$  threads, enquanto o número de blocos é calculado dinamicamente para cobrir a área total da grade. Durante o loop de iterações, o kernel é chamado para atualizar os valores de concentração, e a cada 100 iterações os resultados intermediários são copiados de volta para o host para avaliar a diferença média entre os estados.

Ao término da simulação, os dados finais são copiados da GPU para a CPU e a concentração central da grade é exibida. O código também garante que toda a memória alocada seja devidamente liberada, evitando vazamentos. A implementação é eficiente e demonstra o uso correto das ferramentas CUDA para resolver problemas de difusão em larga escala.

## 3. Descrição da Avaliação de Desempenho

O código serial representa a execução sequencial de tarefas, comando por comando, sem o uso de múltiplos threads ou processos paralelos. Já o código paralelo divide essas tarefas entre diferentes threads ou núcleos, com o objetivo de reduzir o tempo total de execução. Nesse tópico, aborda-se a diferença entre o tempo de execução do código da forma sequencial e da forma em paralelo, utilizando métricas como tempo total, speedup e eficiência.

Tempo total: O tempo total de execução é a duração completa de um programa, medida desde o momento em que é iniciado até a sua finalização. Ele reflete o tempo necessário para que todas as instruções sejam processadas, incluindo possíveis atrasos causados por fatores externos, como o uso compartilhado de recursos do sistema. Para medir o tempo total de execução dos códigos, utilizou-se o comando !time no Google Colab. Este comando fornece três métricas distintas:

- Tempo real: Representa o tempo decorrido entre o início e o fim da execução, incluindo pausas e interferências externas.
- Tempo de usuário: Mede o tempo efetivamente gasto pelo processador executando as instruções do programa.
- Tempo de sistema: Refere-se ao tempo gasto pelo sistema operacional gerenciando operações necessárias para a execução do programa.

Entre essas métricas, foi considerado o tempo real como base para a análise, pois ele reflete de forma mais ampla a experiência do usuário com o tempo total de execução do programa. Para obter resultados mais consistentes, foram realizadas cinco execuções de cada código, reduzindo a influência de variações pontuais. A média dos tempos reais dessas execuções foi então utilizada para representar o tempo total de execução.

Speedup: Mede o fator de melhoria ao executar o código paralelo em relação ao serial. É calculado como:

$$Speedup = \frac{TempoMédioSerial}{TempoMédioParalelo}$$

Eficiência: Avalia o aproveitamento das threads utilizadas. É calculada como:

$$Eficiência = \frac{Speedup}{NúmeroDeThreads}$$

#### 4. Resultados OpenMP

A análise dos resultados obtidos a partir da execução do código serial e das versões paralelas (utilizando 2 e 4 threads) demonstrou os seguintes tempos médios, speedup e eficiência:

Numéro de Threads	Tempo Médio (s)	Speedup	Eficiência
Código Sequencial	36,14	100%	1
Código com 2 Threads	34,85	103,7%	0,519
Código com 4 Threads	34,14	105,9%	0,265

**Table 1. Tabela com Tempo Médio, Speedup e Eficiência dos códigos**

O tempo médio do código serial foi 36,14 segundos, enquanto as versões paralelas obtiveram reduções para 34,85 segundos (2 threads) e 34,14 segundos (4 threads). Embora o paralelismo tenha reduzido o tempo de execução, os ganhos foram modestos.

O speedup, que mede a melhoria relativa em comparação com o código serial, apresentou valores baixos:

- 1,037 com 2 threads (3,7% mais rápido).
- 1,059 com 4 threads (5,9% mais rápido).

A eficiência, que avalia o aproveitamento das threads, foi de 51,9% para 2 threads e caiu para 26,5% para 4 threads. Essa queda reflete um aumento no custo associado à coordenação entre threads e/ou redução na escala paralelizável.

Conclusão: Embora o tempo total tenha diminuído com o aumento no número de threads, os ganhos não foram proporcionais.

## 5. Resultados CUDA

O código foi executado através da plataforma do Google Collab, os resultados obtidos partir do comando "time" e foram de:

1,56s	1,58s	1,58s	1,59s	1,63s
-------	-------	-------	-------	-------

**Table 2. Tempos de execução obtidos com o código CUDA**

Com os cinco tempos de execução, obtivemos o tempo médio de: 1,588 segundos.

A análise dos resultados obtidos a partir da execução do código serial, das versão paralela OpenMP (4 threads) e do código utilizando CUDA demonstrou os seguintes tempos médios, speedup e eficiência:

Tipo de código	Tempo Médio (s)	Speedup
Código Sequencial	36,14	100%
Código CUDA	1,59	2148%

**Table 3. Tabela com Tempo Médio, Speedup e Eficiência dos códigos, comparando o código sequencial com código CUDA**

Tipo de código	Tempo Médio (s)	Speedup
Código OpenMP 4 threads	34,14	100%
Código CUDA	1,59	2205%

**Table 4. Tabela com Tempo Médio, Speedup e Eficiência dos códigos, comparando o código OpenMP com 4 threads com código CUDA**

A versão CUDA demonstrou ser significativamente mais rápida em comparação com as outras versões do código, evidenciando o poder da paralelização em GPUs. Enquanto a versão sequencial com OpenMP (usando 4 threads) e a versão sequencial pura apresentaram tempos de execução consideráveis, a versão CUDA executou a tarefa em apenas 1,58 segundos, resultando em uma redução drástica no tempo de execução. O speedup de 22,05 vezes em relação ao OpenMP e 21,48 vezes em relação à versão sequencial pura reforça o impacto da paralelização em hardware especializado. Isso mostra que, quando bem implementada, a execução em GPU pode levar a ganhos de desempenho extraordinários, superando amplamente as versões paralelizadas em CPU. A versão CUDA foi, sem dúvida, muito mais eficiente e rápida, mostrando o potencial de otimização que as GPUs oferecem para problemas altamente paralelizáveis.

## 6. Características do Código MPI

Utilizamos um código híbrido, que usa MPI e OpenMP para paralelizar a simulação da equação de difusão em uma matriz 2000x2000. O código divide as linhas da matriz entre os processos MPI. Ele distribui as linhas da matriz entre os processos MPI, onde cada um executa a atualização da concentração em sua região. A inicialização ocorre no processo de rank 0, que define uma condição inicial e a compartilha com os demais processos via MPI bcast. Utilizamos o OpenMP para paralelizar os loopings.

## 7. Resultados MPI

O código foi executado através da plataforma do Google Collab, os resultados obtidos partir do comando "time" e foram de:

27,86s	27,92s	27,79s	29,43s	29,48s
--------	--------	--------	--------	--------

**Table 5. Tempos de execução obtidos com o código MPI**

Com os cinco tempos de execução, obtivemos o tempo médio de: 28,49 segundos.

Tipo de código	Tempo Médio (s)	Speedup
Código Sequencial	36,14	100%
Código MPI	28,49	127%

**Table 6. Tabela com Tempo Médio, Speedup do código, comparando o código sequencial com código MPI**

Tipo de código	Tempo Médio (s)	Speedup
Código OpenMP 4 threads	34,14	100%
Código MPI	28,49	120%

**Table 7. Tabela com Tempo Médio, Speedup dos códigos, comparando o código OpenMP com 4 threads com código MPI**

A execução do código sequencial resultou em um tempo de 36,14 segundos, enquanto a versão paralelizada com MPI apresentou um tempo reduzido de 28,49 segundos. Essa redução no tempo de execução indica um ganho de desempenho devido à distribuição das tarefas entre múltiplos processadores.

## References

Herlihy, M. and Shavit, N. (2008). *The Art of Multiprocessor Programming*. Elsevier, Burlington, 1st edition.

Tipo de código	Tempo Médio (s)	Speedup
Código CUDA 4 threads	1,59	100%
Código MPI	28,49	5.6%

**Table 8. Tabela com Tempo Médio, Speedup dos códigos, comparando o código CUDA com código MPI**