# Otimização na Simulação e Análise de Modelos de Difusão de Contaminantes em Água

**Pedro Paulo F. M. Vianna, Victor Jorge C. Chaves, João Guilherme Iwo D. L. Costa**

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

***Abstract.*** *This work aims to create a simulation that models the diffusion of contaminants in a body of water (such as a lake or river), applying concepts of parallelism to speed up the calculation and observe the behavior of pollutants over time. The project will investigate the impact of OpenMP, CUDA and MPI on model runtime and accuracy. In this document we will address the implementation in OpenMP, parallelizing the code provided in the project proposal.*

***Resumo.*** *Este trabalho tem como objetivo criar uma simulação que modele a difusão de contaminantes em um corpo d'água (como um lago ou rio), aplicando conceitos de paralelismo para acelerar o cálculo e observar o comportamento de poluentes ao longo do tempo. O projeto investigará o impacto de OpenMP, CUDA e MPI no tempo de execução e na precisão do modelo.*

## 1. Introdução

### 1.1. Motivação

A difusão de contaminantes em corpos d'água, como lagos, rios e oceanos, é um tema de grande relevância ambiental e social, dada a crescente preocupação com a qualidade da água e o impacto de atividades humanas no meio ambiente. A contaminação por substâncias químicas, resíduos industriais, efluentes domésticos e outros poluentes pode comprometer ecossistemas aquáticos e a saúde pública. Assim, entender e prever a propagação desses contaminantes é fundamental para a formulação de políticas de controle e mitigação.

Uma abordagem eficaz para estudar a difusão de contaminantes é através de simulações computacionais. As simulações permitem a análise de diferentes cenários e variáveis, oferecendo uma visão detalhada sobre o comportamento dos poluentes ao longo do tempo e em diferentes condições ambientais. Essas ferramentas proporcionam aos pesquisadores a capacidade de modelar e prever eventos de contaminação, auxiliando na tomada de decisões mais informadas e eficazes.

No entanto, a simulação de processos de difusão em grande escala pode ser computacionalmente intensiva, especialmente quando se considera a necessidade de alta precisão e a complexidade dos modelos envolvidos. Para superar esses desafios, a otimização do código e a utilização de técnicas de paralelismo se tornam essenciais. Tecnologias como OpenMP (Open Multi-Processing), CUDA (Compute Unified Device Architecture) e MPI (Message Passing Interface) oferecem soluções poderosas para acelerar os cálculos, permitindo o processamento simultâneo de múltiplas operações e a distribuição de tarefas entre diferentes unidades de processamento.

Este estudo se concentra na implementação e comparação dessas técnicas de otimização em simulações de difusão de contaminantes, avaliando seu impacto no desempenho e na precisão dos modelos. Através da utilização de OpenMP, CUDA e MPI, será buscado a redução do tempo de execução de um algoritmo existente de simulação do processo de difusão em corpos d'água.

### 1.2. Modelo de Difusão Utilizado

$$\frac{\partial C}{\partial t} = D \cdot (\nabla)^2 C$$

Onde:

- $C$ é a concentração do contaminante,
- $t$ é o tempo,
- $D$ é o coeficiente de difusão, e
- $(\nabla)^2 C$ representa a taxa de variação de concentração no espaço.

Este modelo descreve a difusão de um contaminante em um meio homogêneo e isotrópico. A equação baseia-se na Segunda Lei de Fick, que afirma que a taxa de variação temporal da concentração em um ponto é proporcional à divergência do fluxo de difusão nesse ponto.

Para resolver esta equação, utiliza-se o método de diferenças finitas, discretizando o espaço e o tempo. A discretização leva à equação de diferença:

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

Onde:

- $C_{i,j}^{n}$ é a concentração no ponto $(i, j)$ no tempo $n$,
- $\Delta t$ é o passo temporal, e
- $\Delta x$ é o tamanho da célula no espaço discreto.

Este esquema permite calcular a evolução temporal da concentração em uma grade 2D, sendo adequado para implementação em sistemas paralelos, como o modelo estudado neste trabalho.

## 2. Objetivo

O objetivo deste estudo é desenvolver uma simulação eficiente para modelar a difusão de contaminantes em corpos d'água, como lagos e rios, utilizando técnicas avançadas de paralelismo e otimização de código. A simulação visa não apenas prever o comportamento dos poluentes ao longo do tempo, mas também avaliar o desempenho computacional de diferentes abordagens de paralelização.

As principais metas incluem:

- Implementar e comparar técnicas de paralelismo utilizando OpenMP, CUDA e MPI para acelerar a simulação.

- Avaliar o impacto da otimização -O3 do GCC no desempenho da simulação.
- Analisar a eficiência e a escalabilidade de cada técnica em diferentes cenários de simulação, variando o tamanho da matriz e o número de iterações.
- Fornecer insights sobre a melhor abordagem para otimizar a execução de simulações de difusão de contaminantes em ambientes computacionais diversos.

Ao alcançar esses objetivos, o estudo pretende contribuir para a compreensão dos benefícios e limitações das diferentes técnicas de paralelismo e otimização, oferecendo uma base sólida para futuros trabalhos na área de simulação ambiental e modelagem computacional.

## 3. Metodologia

Para otimizar a simulação da difusão de contaminantes em corpos d'água, diversas técnicas de paralelismo e otimização de código foram empregadas. Nesta seção, são descritas as metodologias relacionadas ao uso de CUDA, OpenMP, MPI e a otimização -O3 do GCC, com o objetivo de melhorar o desempenho computacional da simulação.

### 3.1. CUDA (Compute Unified Device Architecture)

CUDA é uma plataforma de computação paralela desenvolvida pela NVIDIA, que permite o uso de GPUs (Graphics Processing Units) para realizar cálculos computacionais intensivos. Ao distribuir as operações de cálculo em milhares de núcleos da GPU, CUDA possibilita uma aceleração significativa em tarefas que envolvem grandes volumes de dados ou computações repetitivas, como na simulação de difusão de contaminantes. A implementação em CUDA envolve a escrita de kernels, que são funções executadas em paralelo pela GPU, e a gestão eficiente da memória entre a CPU e a GPU.

### 3.2. OpenMP (Open Multi-Processing)

OpenMP é uma API que suporta a programação paralela em ambientes compartilhados de memória. Com uma sintaxe baseada em diretivas, OpenMP facilita a paralelização de loops e outras estruturas de controle em linguagens como C, C++ e Fortran. A utilização de OpenMP nesta simulação permite a execução paralela em múltiplos threads, aproveitando melhor os recursos de CPUs multicore. A abordagem é especialmente útil para dividir o trabalho computacional entre os núcleos do processador, reduzindo o tempo total de execução.

### 3.3. MPI (Message Passing Interface)

MPI é uma biblioteca padrão para programação paralela em sistemas de memória distribuída. Ao contrário do OpenMP, que opera em memória compartilhada, MPI permite a comunicação entre diferentes processos que podem estar executando em máquinas distintas. Na simulação de difusão de contaminantes, o MPI é utilizado para distribuir partes do domínio de cálculo entre múltiplos nós de um cluster de computadores, sincronizando e agregando os resultados. Isso permite o escalonamento eficiente da simulação para conjuntos de dados muito grandes.

### 3.4. Otimização -O3 do GCC

A flag -O3 do compilador GCC (GNU Compiler Collection) ativa uma série de otimizações agressivas de código que visam maximizar o desempenho de programas compilados. Entre as otimizações realizadas estão a unrolling de loops, a substituição de expressões comuns, a eliminação de código redundante, e a inlining de funções. Essas otimizações são aplicadas automaticamente pelo GCC durante a compilação do código, reduzindo o tempo de execução e melhorando a eficiência do programa sem a necessidade de modificações manuais no código-fonte.

## 4. Desenvolvimento e Resultados

Nesta seção, são apresentados os resultados obtidos para cada técnica de otimização aplicada, assim como os resultados do algoritmo sem otimização. A análise considera diferentes configurações de simulações, avaliando o impacto de cada técnica no desempenho computacional e na eficiência do modelo.

Para cada técnica, foi conduzida uma série de simulações utilizando combinações variadas de tamanhos de matriz e quantidades de iterações, conforme descrito a seguir:

- **Tamanho da Matriz:** 400, 800, 1200, 1600, 2000
- **Quantidade de Iterações:** 200, 400, 600, 800, 1000

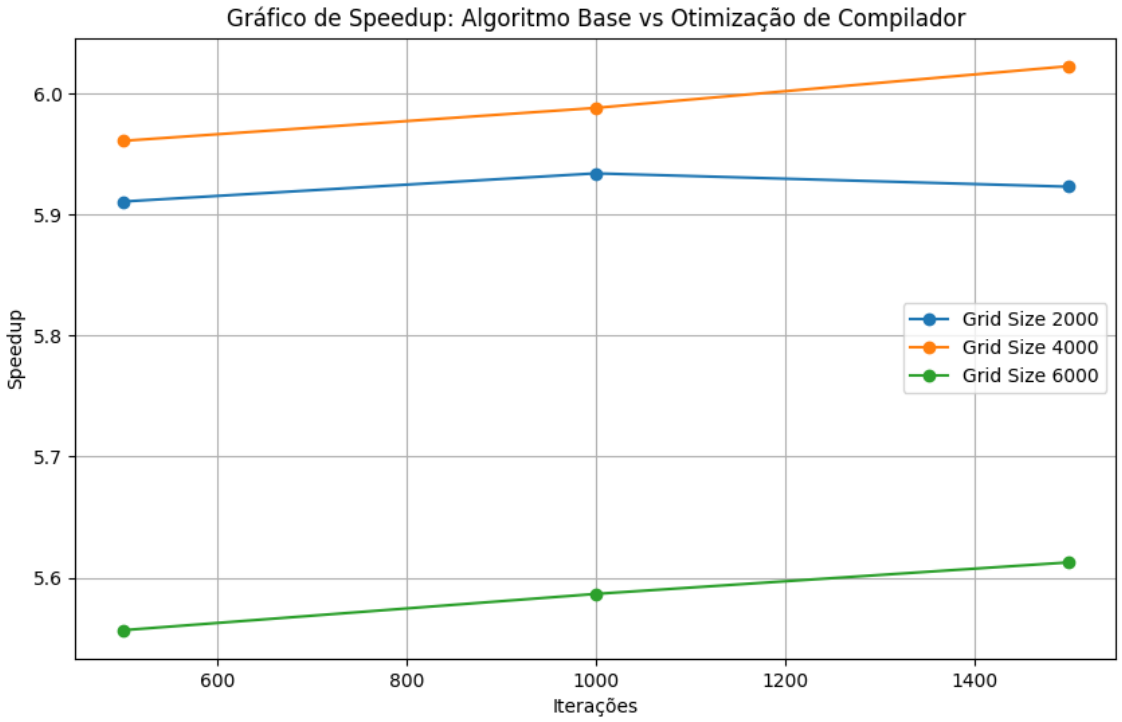
Essas combinações foram escolhidas para simular diferentes cenários de complexidade computacional, permitindo uma análise detalhada do desempenho de cada técnica em condições variadas. Os resultados focam nas métricas de tempo de execução.

### 4.1. Execução do Algoritmo Compilado com a Flag -O3

#### 4.1.1. Configuração do Ambiente de Execução

- Máquina: Notebook IdeaPad Lenovo
- CPU: Ryzen 5 5600G
- Memória RAM: 12 GB
- Compilador: GCC

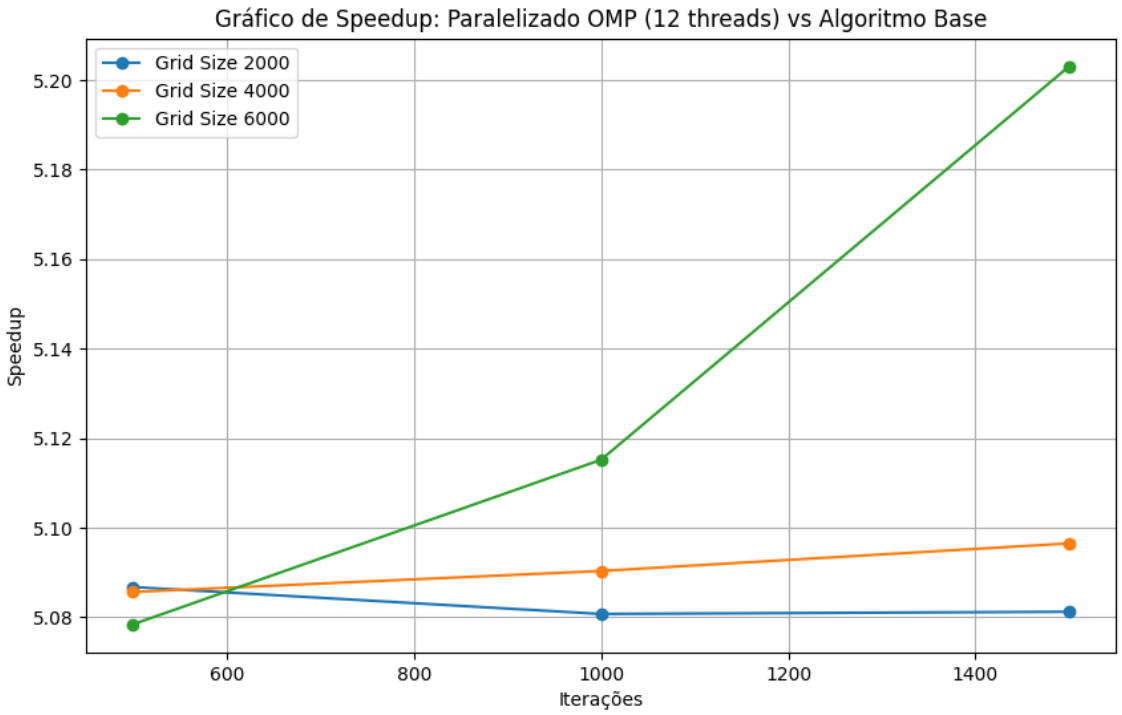
### 4.1.2. Resultados

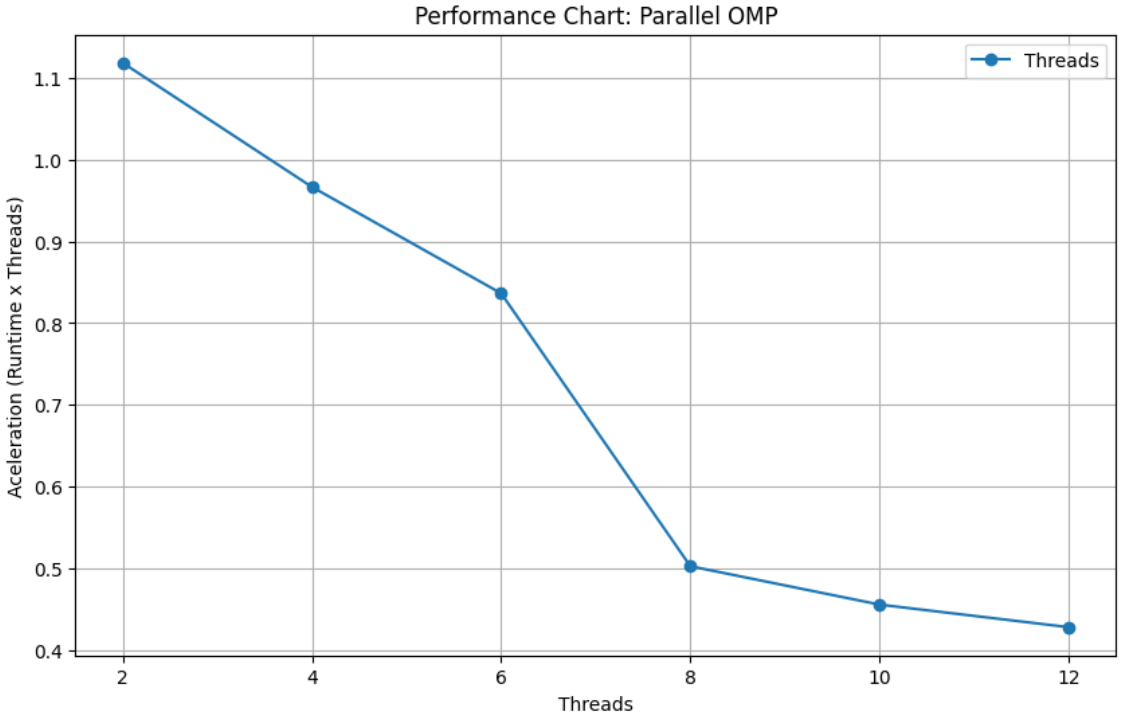



## 4.2. Execução do Algoritmo Paralelizado com OpenMP

### 4.2.1. Configuração do Ambiente de Execução

- Máquina: Notebook IdeaPad Lenovo
- CPU: Ryzen 5 5600G
- Memória RAM: 12 GB
- Compilador: GCC



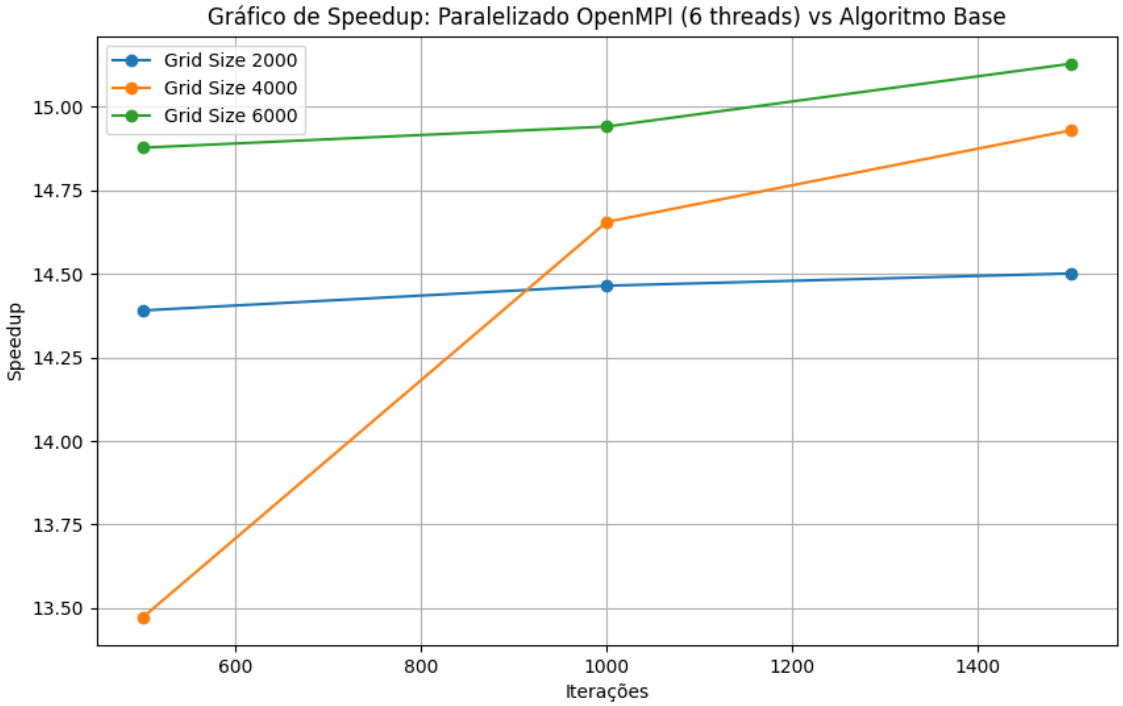



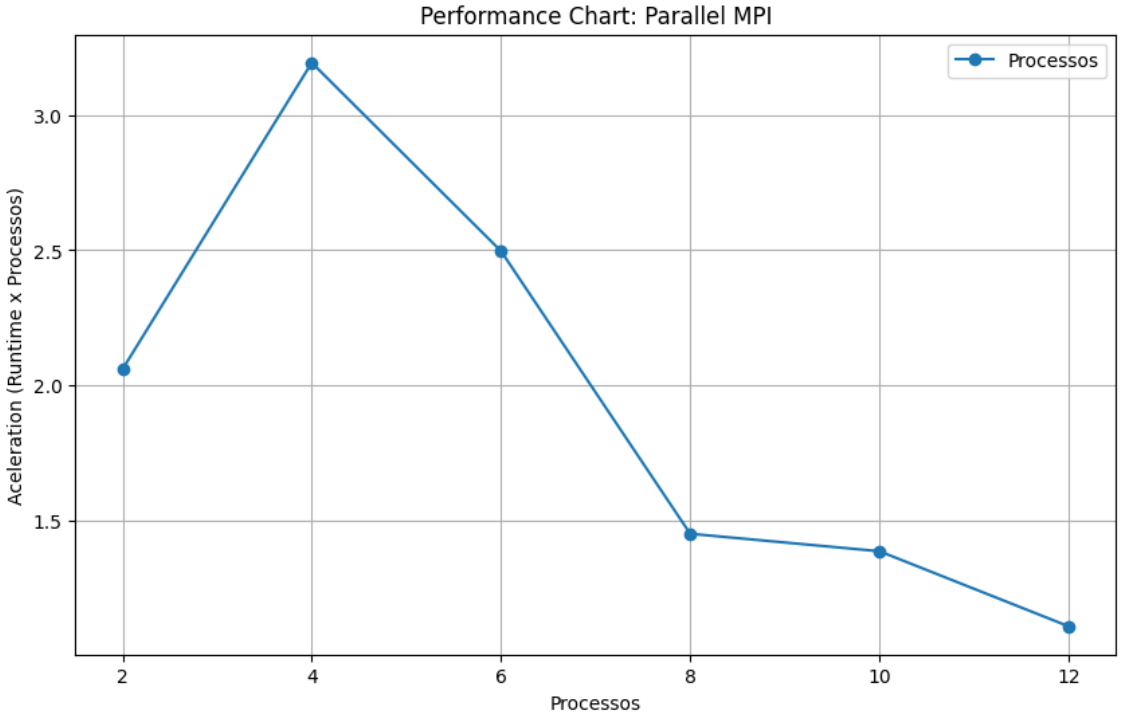

### 4.3. Execução do Algoritmo Paralelizado com OpenMPI

#### 4.3.1. Configuração do Ambiente de Execução

- Máquina: Notebook IdeaPad Lenovo

- CPU: Ryzen 5 5600G
- Memória RAM: 12 GB
- Compilador: GCC

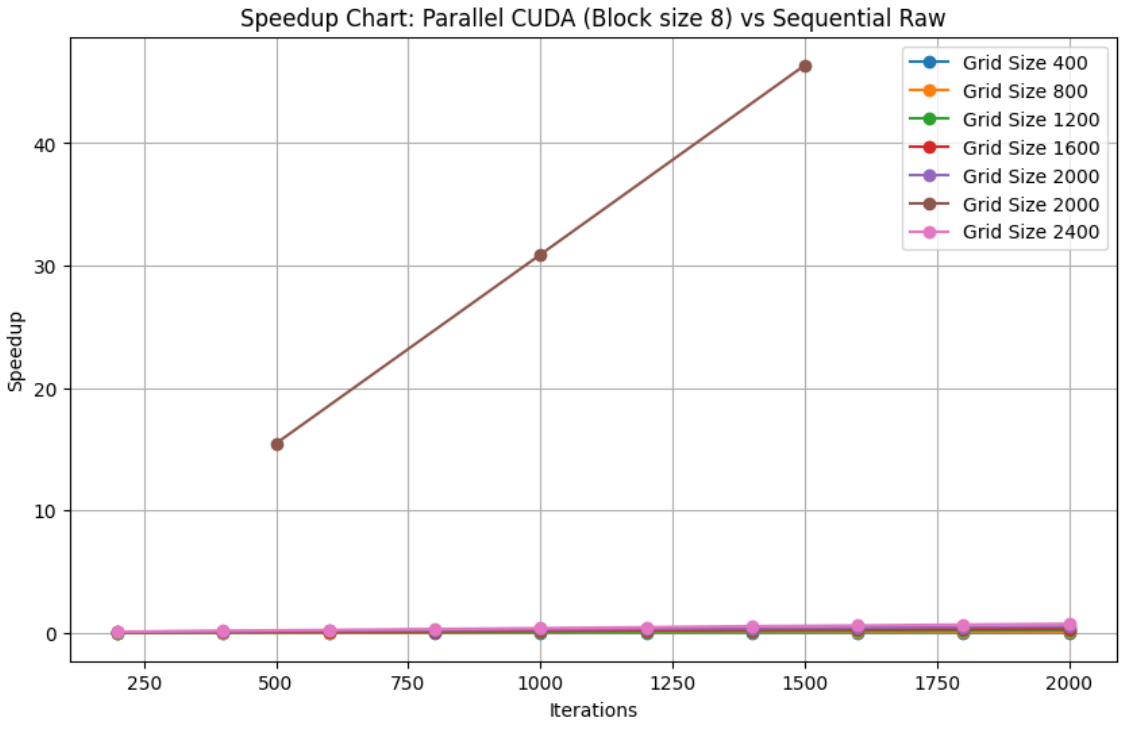


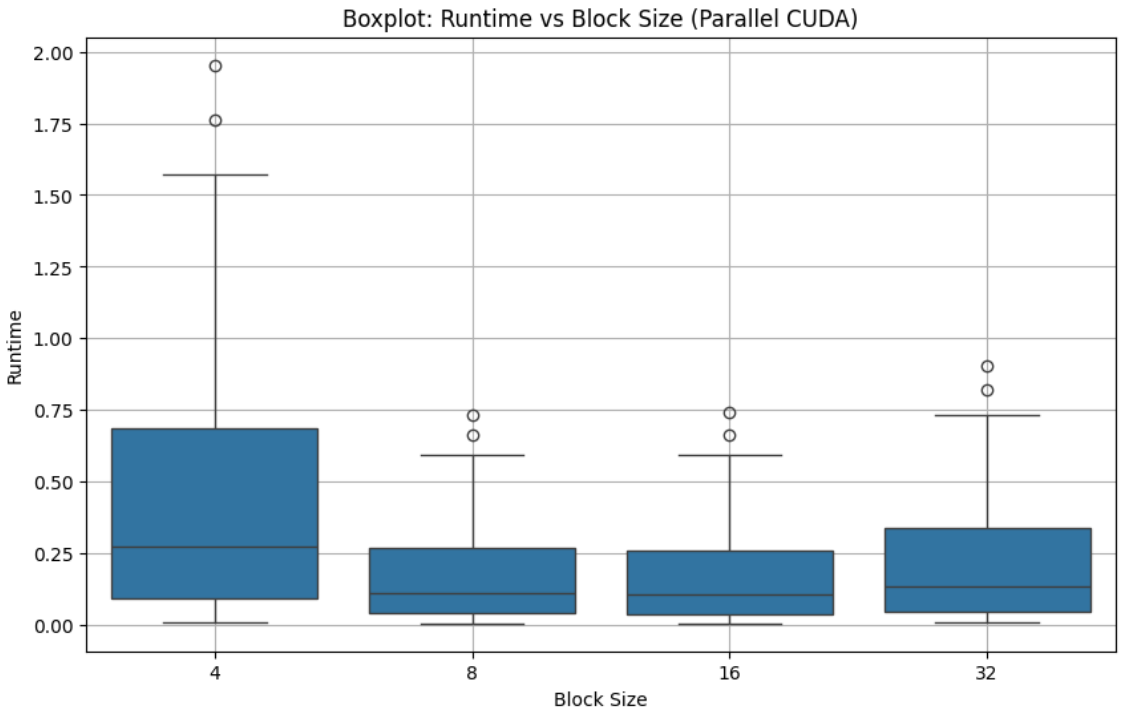



## 4.4. Execução do Algoritmo Adaptado para CUDA

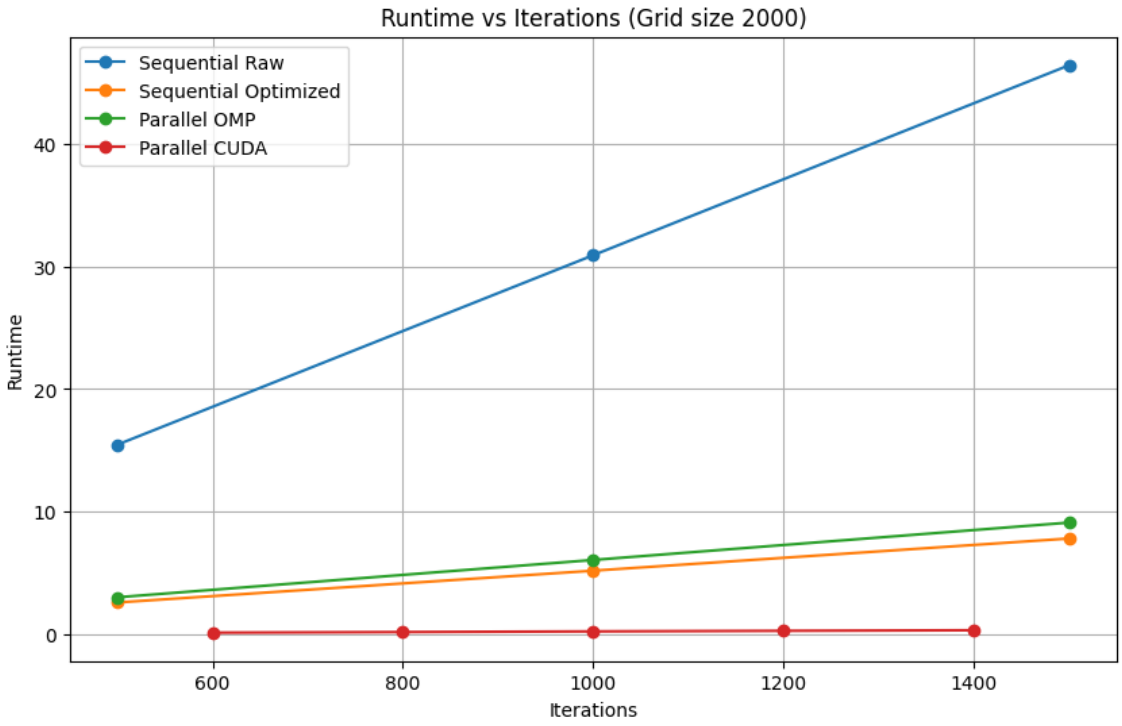
### 4.4.1. Configuração do Ambiente de Execução

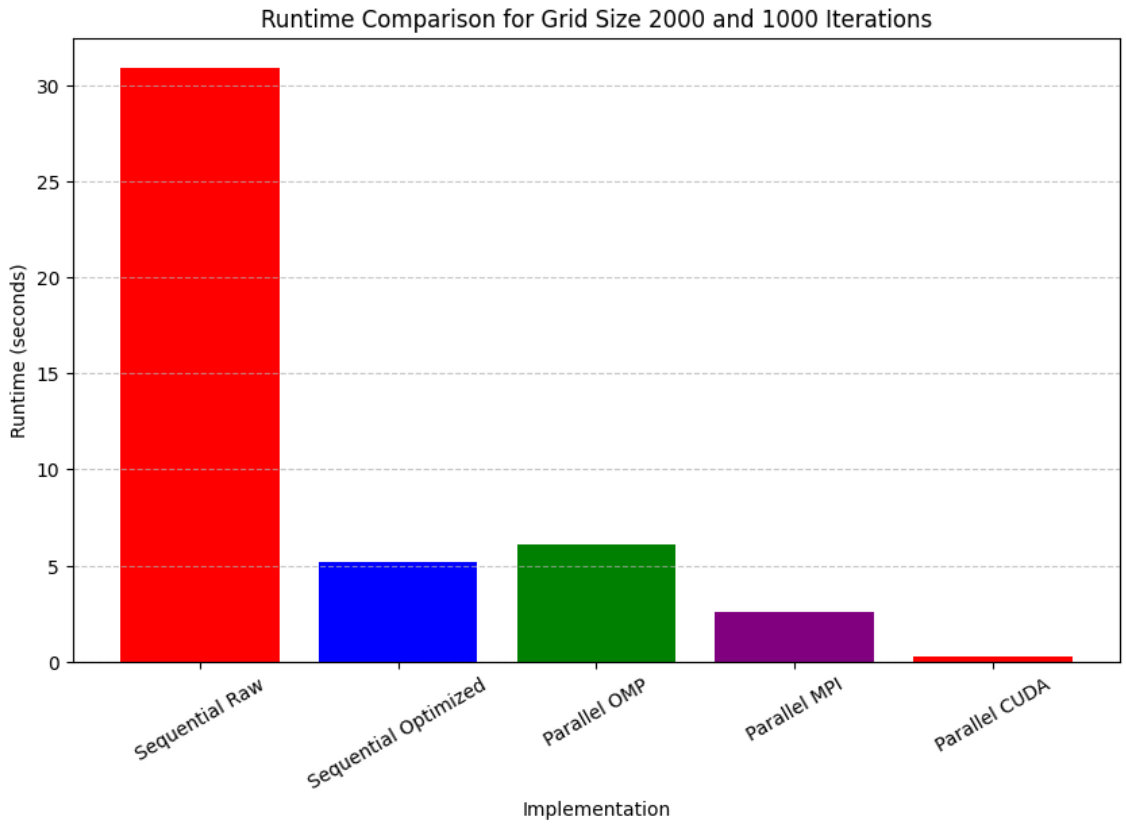
- Máquina: Ambiente Virtual do Google Colab
- CPU:
- Memória RAM: 16 GB
- GPU: T4
- Compilador: NVCC (NVIDIA CUDA Compiler)






### 4.5. Comparativo com todos os Métodos






## 5. Discussão

Os resultados obtidos evidenciam que a implementação CUDA apresentou o melhor desempenho, mas houve uma falha na obtenção de mais dados, o que limitou a análise mais aprofundada da escalabilidade da solução. No entanto, dentro dos experimentos realizados, verificou-se que um tamanho de bloco igual a 8 proporcionou o melhor desempenho. Esse comportamento pode ser explicado pelo balanceamento entre a ocupação da GPU e a minimização da sobrecarga associada à troca de contexto entre warps. Blocos muito pequenos podem não explorar eficientemente a capacidade dos multiprocessadores da GPU, enquanto blocos muito grandes podem causar contenção de recursos e maior latência na comunicação entre threads.

No caso do MPI, a configuração com 6 processos demonstrou o melhor desempenho entre as implementações paralelas CPU. Isso pode ser atribuído ao fato de que um número adequado de processos permite um balanceamento eficiente da carga de trabalho entre os núcleos disponíveis, minimizando a sobrecarga de comunicação e sincronização. Por outro lado, observou-se uma instabilidade nos tempos de execução do MPI, o que pode estar relacionado a flutuações na comunicação entre processos e à alocação dinâmica de recursos do sistema operacional, especialmente em clusters compartilhados.

Outro ponto relevante foi a queda de desempenho tanto no MPI quanto no OpenMP conforme o número de processos ou threads aumentava. Esse fenômeno pode ser explicado pelo aumento da contenção de memória compartilhada e pelo overhead gerado pela sincronização entre as threads/processos. No OpenMP, à medida que mais threads são adicionadas, o

benefício do paralelismo é progressivamente reduzido pela latência no acesso à memória e pela competição por cache L3. No MPI, um número excessivo de processos pode aumentar o custo da comunicação interprocessos, tornando a execução menos eficiente.

Curiosamente, a versão sequencial otimizada com a flag -O3 do GCC superou a implementação OpenMP com 12 threads. Isso sugere que a otimização agressiva aplicada pelo compilador, incluindo vectorization e reordering de instruções, foi suficiente para melhorar o desempenho sem a necessidade do overhead de gerenciamento de múltiplas threads. Esse resultado reforça a importância de considerar otimizações a nível de compilação antes de recorrer a técnicas de paralelismo explícito, uma vez que a simples ativação de OpenMP não garante automaticamente um ganho de desempenho significativo.

Diante dessas observações, futuros trabalhos podem explorar ajustes mais refinados na granularidade do paralelismo, estratégias de escalonamento de threads e otimizações específicas para minimizar a sobrecarga de comunicação em implementações MPI e OpenMP. Além disso, uma análise mais detalhada da configuração da memória na GPU poderia permitir um entendimento mais preciso sobre os limites da implementação CUDA e sua escalabilidade em diferentes tamanhos de problemas.

## 6. Conclusão

Os resultados obtidos evidenciam a importância das técnicas de paralelização para a redução do tempo de execução em simulações computacionais intensivas. A implementação sequencial não otimizada apresentou um tempo de execução significativamente superior às demais abordagens, demonstrando a ineficiência de métodos não paralelos para problemas de grande escala.

A versão sequencial otimizada reduziu o tempo de execução, mas ainda permaneceu consideravelmente mais lenta do que as abordagens paralelas. Entre as técnicas de paralelização analisadas, a implementação via CUDA apresentou o melhor desempenho, alcançando uma redução expressiva no tempo de execução em comparação com todas as outras abordagens. O uso de OpenMP e MPI também demonstrou ganhos significativos de desempenho, mas ainda superiores ao tempo registrado pela abordagem CUDA.

Portanto, os resultados indicam que a utilização de GPUs com CUDA se mostra particularmente eficiente para problemas do tipo stencil 2D, onde o processamento de células pode ser altamente paralelizado devido à sua natureza estruturada e ao alto grau de reutilização de dados vizinhos. Essa característica permite um melhor aproveitamento da hierarquia de memória da GPU, reduzindo a latência e aumentando a largura de banda efetiva. Trabalhos futuros podem explorar diferentes estratégias de acesso à memória global e compartilhada para melhorar ainda mais a eficiência da computação stencil em arquiteturas CUDA.