Simulação e Análise de Modelos de Difusão de Contaminantes em Água
Projeto de Programação Concorrente e Distribuída

Camila Barretto Lins Paes, Harrison Caetano Candido, Laura Maria Cunha Lisbôa

[1]Instituto de Ciência e Tecnologia – Universidade Federal de São Paulo (UNIFESP)
Rua Talim, 330, São José dos Campos, São Paulo, CEP 12231-280.

Abstract. *This project develops a simulation to model the diffusion of contaminants in bodies of water, such as lakes or rivers, by applying parallelism concepts with OpenMP, CUDA, and MPI to analyze the behavior of pollutants over time. The approach implements the two-dimensional diffusion equation using finite differences on a 2D grid, where each cell represents the concentration of contaminants in a specific region.*

Resumo. *Este projeto desenvolve uma simulação para modelar a difusão de contaminantes em corpos d'água, como lagos ou rios, aplicando conceitos de paralelismo com OpenMP, CUDA e MPI, para analisar o comportamento dos poluentes ao longo do tempo. A abordagem implementa a equação de difusão bidimensional utilizando diferenças finitas em uma grade 2D, em que cada célula representa a concentração de contaminantes em uma região específica.*

1. Introdução

Com o passar dos séculos, a sociedade de consumo adquiriu um grande ritmo voltado para uma vida menos natural e mais dependente das indústrias. Com isso, remédios, cosméticos e produtos de higiene são lançados no esgoto. O resultado disso gera bastante preocupação, pois, mesmo em água tratada e potável, são apresentadas concentrações desses resíduos, chamados de "contaminantes emergentes".

Sabendo disso, este projeto tem como objetivo de realizar e verificar a simulação de difusão desses contaminantes e observar seu comportamento ao longo do tempo.

A difusão de contaminantes em corpos d'água é um fenômeno crucial em estudos ambientais e de engenharia, modelado frequentemente pela equação de difusão (ou equação do calor). Esta equação, em sua forma bidimensional, descreve a dispersão de um contaminante (C) em função do tempo (t) e da posição (x, y), sendo D o coeficiente de difusão. Devido à complexidade de soluções analíticas para condições reais, métodos numéricos como o método das diferenças finitas são empregados, discretizando o espaço e o tempo para simular a propagação do contaminante. A resolução de problemas em larga escala exige alta capacidade computacional, tornando o paralelismo (OpenMP, CUDA, MPI) essencial para acelerar simulações com grades de alta resolução. Simulações desta natureza são aplicáveis à gestão ambiental, engenharia hidráulica e ciências naturais, auxiliando na avaliação de impactos de poluentes e no planejamento de sistemas de contenção.

A implementação eficaz dessas simulações requer atenção à estabilidade numérica, precisão dos resultados e otimização computacional, balanceando o detalhe

$$\frac{\partial C}{\partial t} = D\left(\frac{\partial^2 C}{\partial x^2} + \frac{\partial^2 C}{\partial y^2}\right)$$

Figure 1. Equação de difusão

da simulação com o tempo de processamento. Este contexto justifica o uso de técnicas computacionais modernas para compreender a complexidade da difusão de contaminantes e suas implicações práticas.

1.1. Estudo do Modelo de Difusão

O estudo do modelo de difusão, baseado na segunda lei de Fick, concentra-se na propagação de substâncias (poluentes, nutrientes, calor) em um meio devido a gradientes de concentração, encontrando aplicações em diversas áreas científicas e de engenharia. A equação descreve a variação da concentração (C) no tempo (t) e espaço (x, y), sendo D o coeficiente de difusão.

Como a solução analítica é frequentemente inviável, métodos numéricos, como o de diferenças finitas, são utilizados, discretizando o domínio em uma grade. A estabilidade numérica é crucial, com o parâmetro alfa devendo ser menor ou igual 0.25 para evitar instabilidades. Em aplicações ambientais, o modelo prevê a dispersão de poluentes em corpos d'água, auxiliando na tomada de decisões.

Simulações em larga escala demandam alta capacidade computacional, em que o paralelismo e seus modelos (OpenMP, CUDA, MPI) se tornam essenciais para acelerar os cálculos, tornando as simulações mais acessíveis e detalhadas. E, assim, permitindo assim uma melhor compreensão e previsão de fenômenos de difusão e suas implicações práticas.

2. Implementações e testes

2.1. OPENMP

OPENMP (Open Multi-Processing) é uma API para programação paralela. Bastante utilizada nos dias de hoje, oferece contribuição com o multiprocessamento em memória compartilhada. Com isso, sua implementação é multithreading, ou seja, permite que diferentes partes do código sejam executadas simultaneamente em múltiplos núcleos da CPU. As threads são executadas de maneira coexistente em um ambiente que as distribui para diferentes processadores. Dessa maneira, para a simulação deste projeto, o OPENMP será utilizado para paralelizar o cálculo de difusão entre os núcleos da CPU.

Esta seção apresenta a resolução numérica da equação de difusão em duas dimensões utilizando o método explícito de diferenças finitas. A abordagem simula a propagação de uma concentração inicial ao longo do tempo em uma grade discreta, modelando o comportamento físico da difusão em um meio isotrópico.

O programa define constantes que determinam o tamanho da grade (N), o número de iterações temporais (T), o coeficiente de difusão (D), e os incrementos espaciais (Δx) e temporais (Δt). As matrizes C e C_{new} armazenam os valores de concentração na iteração

atual e na próxima, respectivamente, sendo alocadas dinamicamente para permitir escalabilidade. A cada iteração, a matriz C_{new} é usada para calcular os valores de concentração atualizados e, em seguida, os resultados são transferidos para C.

O cálculo da difusão é realizado por meio da fórmula explícita de diferenças finitas:

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

onde $C_{i,j}^{t}$ é a concentração no ponto (i, j) na iteração t, D é o coeficiente de difusão, e Δx e Δt são os incrementos espaciais e temporais, respectivamente. Apenas os pontos internos da grade são atualizados, enquanto as bordas permanecem fixas, representando condições de contorno estáticas.

Inicialmente, a matriz C é preenchida com zeros, exceto por uma alta concentração no centro da grade, representando uma fonte inicial. A propagação da concentração ao longo do tempo segue o perfil característico da difusão, com a matriz sendo atualizada iterativamente. Para facilitar o acompanhamento, o progresso da simulação é exibido no console a cada 100 iterações.

Ao final da simulação, os resultados são salvos em um arquivo (`output_final.txt`) utilizando a função `save_grid`, que formata os dados em uma estrutura legível. Esses resultados podem ser utilizados para gerar visualizações, como gráficos 2D ou mapas de calor, para interpretar o comportamento do processo de difusão.

Essa implementação, embora básica, é eficaz para resolver problemas de difusão e pode ser expandida para incluir condições de contorno dinâmicas, coeficientes não homogêneos ou paralelização para simulações de maior escala. Assim, ela oferece uma base sólida para aplicações em áreas científicas e de engenharia que necessitam de modelagem computacional de processos de transporte e dispersão.

2.2. CUDA

CUDA (Compute Unified Device Architecture), assim como OPENMP, é uma plataforma para desenvolvimento em computação paralela. Com a sua aplicação, há uma eficiência no paralelismo, otimização no tocante às GPUs. Assim, para esta parte do desenvolvimento, com CUDA, cada célula da grade é processada por uma thread independente na GPU, utilizando um esquema de diferenças finitas para calcular o laplaciano de (C).

Este programa implementa uma simulação de difusão em uma grade bidimensional utilizando CUDA para paralelização, onde cada célula da grade é processada por uma thread independente na GPU. A simulação resolve a equação do calor por meio de diferenças finitas, calculando o laplaciano para cada célula com base nos valores de seus vizinhos.

Inicialmente, uma grade de $N \times N$ é alocada na CPU e configurada com valores nulos, exceto no centro, que possui uma alta concentração inicial. A grade é então transferida para a memória da GPU para processamento. O coeficiente de difusão $\alpha = D \cdot \frac{\Delta T}{\Delta X^2}$ é calculado, e uma verificação de estabilidade é realizada ($\alpha \leq 0.25$) para evitar instabilidades numéricas. A execução paralela em CUDA é configurada com uma grade de blocos

2D, onde cada bloco contém `THREADS_PER_BLOCK` × `THREADS_PER_BLOCK` threads, garantindo que todas as células sejam processadas.

O ciclo de simulação é executado por `TIME_STEPS`, onde o kernel `update_grid` atualiza os valores da grade com base no esquema de diferenças finitas, e o kernel `apply_boundaries` aplica condições de contorno fixas, definindo as bordas da grade como zero. Após cada iteração, as grades são sincronizadas, e a cada 100 iterações os dados são copiados para a CPU e salvos em arquivos de texto para análise.

O desempenho da simulação é medido utilizando o tempo de execução total na CPU, destacando o ganho proporcionado pela paralelização em CUDA. No final da simulação, a grade final é copiada da GPU para a CPU e salva, e todos os recursos alocados na memória são liberados.

O uso de CUDA permite processar grades maiores de maneira eficiente, aproveitando o paralelismo da GPU para acelerar os cálculos e atender aos requisitos de processamento de cada célula por uma thread independente.

2.3. MPI

MPI (Message Passing Interface) permite que programas sejam dimensionados além dos processadores e da memória compartilhada de um único servidor de computação. Dessa maneira, é requerida algumas alterações do código serial, à medida que são adicionadas chamadas de função MPI para comunicar dados, e os dados devem de alguma forma ser divididos entre processos. Nesta seção, cada máquina processa uma seção do corpo d'água e troca informações nas bordas com as máquinas vizinhas para garantir a continuidade da difusão de contaminantes entre as regiões. Portanto, a implementação de MPI tem o papel de dividir a grade em sub-regiões e distribuir o processamento entre várias máquinas.

Para a simulação de difusão, o problema envolve dividir uma grade em sub-regiões e distribuir o processamento dessas regiões entre diferentes processos MPI. Cada processo é responsável por executar a difusão em uma parte do corpo d'água e deve trocar informações nas bordas com os processos vizinhos para garantir a continuidade da difusão de contaminantes entre as regiões.

Este trabalho deve demonstrar a implementação da difusão em um ambiente híbrido, utilizando MPI para a paralelização entre processos, CUDA para acelerar a computação de cada sub-região em uma GPU, e OpenMP para calcular as estatísticas da sub-região em paralelo no CPU. O objetivo é garantir a escalabilidade da solução e apresentar uma análise comparativa com as versões anteriores.

A abordagem central é dividir a grade global $N \times N$ em sub-regiões, que são atribuídas a diferentes processos MPI. Cada processo cuida de uma sub-região da grade e, ao final de cada iteração de tempo, envia e recebe as fronteiras da sua sub-região para garantir a troca de dados com os processos vizinhos.

Para cada processo MPI, a sub-região é transferida para a GPU, onde os cálculos da equação de difusão são realizados usando um kernel CUDA. O kernel utiliza memória compartilhada para armazenar os dados locais e das bordas de forma otimizada,

garantindo um acesso mais rápido aos valores necessários para o cálculo do laplaciano.

O bloco de threads é dimensionado em 32×32 para maximizar o uso dos recursos da GPU. Cada thread calcula a difusão para uma célula da sub-região. As fronteiras não são recalculadas diretamente, mas preservadas para garantir a continuidade do modelo de difusão.

Adicionalmente, o OpenMP é utilizado para paralelizar o cálculo das estatísticas da sub-região (mínimo, máximo e média) no CPU. O código aproveita múltiplos núcleos do processador para acelerar essa tarefa.

A comunicação entre os processos MPI é feita de forma assíncrona, utilizando as funções `MPI_Isend` e `MPI_Irecv`. Isso permite que o processo continue com o cálculo interno enquanto as trocas de fronteira estão sendo realizadas, minimizando o impacto da comunicação no tempo total da simulação.

3. Resultados e discussão

Para a análise de desempenho para este programa, implementado com a API OpenMP, com API CUDA e com MPI, foram levados em consideração, para cada número de threads/processos comparados, os seguintes parâmetros:

Tempo de Execução: quanto tempo a execução do programa leva para ser concluída. Espera-se que, conforme o número de threads/processos aumenta, o tempo de execução diminua. Essa diminuição, contudo, não costuma ser linear, devido a dependências entre threads/processos, ao overhead de comunicação e sincronização, e às características do hardware utilizado.

Speedup: revela o ganho de desempenho com a paralelização, comparado com o modelo sequencial. Esse indicador é calculado como a razão entre o tempo de execução sequencial e o tempo de execução paralelo. Um speedup maior que 1 indica que a versão paralela é mais rápida que a sequencial.

Eficiência: comparação entre o speedup e o número de threads/processos. Tende a diminuir conforme o número de threads/processos aumenta, por questões de sobrecarga do sistema, contenção de recursos e limitações inerentes ao paralelismo da aplicação.[1]

3.1. Resultados com OpenMP

Para o presente estudo, com OpenMP, os resultados obtidos encontram-se na tabela abaixo.

N° threads	Tempo de execução (s)	*Speedup*	Eficiência
Sequencial	5.84	1	100%
2	5.23	1.12	55.83%
4	5.3	1.10	13.77%
8	6.16	0.94	11.87%
16	5.7	1.02	6.40%
32	6.15	0.96	2.97%

É possível observar uma melhora inicial no tempo de execução ao usar 2 threads, com um speedup de 1,12 e eficiência de 55,83%. No entanto, ao aumentar o número de threads, a eficiência cai drasticamente, chegando a apenas 2,97% com 32 threads. Isso

sugere que o programa não está escalando bem com mais threads, possivelmente devido à sobrecarga de gerenciamento de threads, à contenção de recursos de memória (como o acesso à memória cache compartilhada) ou à falta de paralelismo adequado na tarefa, o que leva a um aumento do tempo de execução em alguns casos, como com 8 e 32 threads.

A baixa escalabilidade com OpenMP pode ser causada por diversos fatores, incluindo:

- **Overhead de criação e sincronização de threads:** o custo de criar e sincronizar um grande número de threads pode ser significativo e dominar o tempo de computação;
- **Contenção de recursos:** threads competindo por acesso à memória compartilhada (especialmente memória cache) podem causar gargalos de desempenho;
- **Paralelismo limitado:** a aplicação pode ter seções críticas que não podem ser paralelizadas, limitando o speedup máximo alcançável.

3.2. Resultados com CUDA

Já para o estudo com CUDA, foram obtidos os seguintes resultados.

N° threads	Tempo de execução (s)	*Speedup*	Eficiência
Sequencial	1.87	1	100%
2	1.06	1.76	88.2%
4	1.39	1.34	33.63%
8	0.77	2.42	30.35%
16	1.38	1.35	8.47%
32	1.30	1.43	4.49%

Os resultados com CUDA evidenciam a eficiência da arquitetura de GPUs no processamento de tarefas paralelizáveis. O tempo de execução sequencial de 1,87 segundos é significativamente menor do que o obtido com OpenMP (5,84 segundos), o que destaca a vantagem da GPU no processamento massivamente paralelo. Com 2 threads, o tempo de execução reduz-se ainda mais para 1,06 segundos, apresentando um speedup de 1,76 e uma eficiência de 88,2%, o que demonstra uma boa utilização dos recursos paralelos na GPU. No entanto, à medida que o número de threads aumenta para 8 e 32, a eficiência diminui consideravelmente, caindo para 30,35% e 4,49%, respectivamente.

Esse comportamento sugere que o kernel da GPU enfrenta dificuldades para escalar a tarefa com um número elevado de threads, especialmente se a carga computacional de cada thread não for suficientemente alta. Além disso, a contenção no acesso à memória global (memória da GPU) e a sobrecarga de comunicação entre CPU e GPU podem contribuir para o desempenho limitado em configurações com muitos threads. A arquitetura CUDA é projetada para um grande número de threads leves, mas a eficiência depende fortemente da utilização da memória e da minimização da comunicação entre a CPU e a GPU.

Possíveis fatores que contribuem para a baixa escalabilidade com CUDA incluem:

- **Saturação da largura de banda da memória:** o acesso à memória global da GPU é um gargalo comum. Com muitas threads, a demanda por largura de banda pode exceder a capacidade da GPU;
- **Latência de comunicação CPU-GPU:** transferir dados entre a CPU e a GPU tem um custo significativo;

3.3. Resultados com MPI

Para a execução paralela utilizando MPI, os resultados são apresentados na tabela abaixo:

N° de processos	Tempo de execução (s)	*Speedup*	Eficiência
Sequencial	0.2670948	1	100%
2	0.5616876	0.4755	23.78%
4	12.0172393	0.0222	0.55%
8	17.7857122	0.0150	0.19%

Os resultados obtidos com MPI não demonstram um desempenho muito satisfatório na execução paralela. Dentro dessa ótica, é possível observar que, à medida que o número de processos aumenta, o tempo de execução se eleva drasticamente, resultando em um *speedup* inferior a 1 e uma eficiência próxima de zero. Esse comportamento indica que o overhead introduzido pela comunicação entre os processos supera os benefícios da computação paralela, o que torna a execução sequencial mais vantajosa.

A quantidade de trabalho realizada por cada processo é muito pequena em relação ao tempo gasto na comunicação. Além disso, problemas de balanceamento de carga, nos quais alguns processos podem permanecer ociosos enquanto outros estão sobrecarregados, contribuem para o desempenho inferior.

Ademais, outro fator importante é a latência da comunicação, que pode se tornar um gargalo à medida que mais processos são envolvidos na execução. A comunicação em MPI envolve a troca de mensagens entre processos, e essa troca tem um custo que pode se tornar proibitivo se a quantidade de dados a ser transferida for grande ou se a comunicação for muito frequente.

As possíveis causas para o baixo desempenho com MPI incluem:

- **Overhead de comunicação elevado:** o tempo gasto na comunicação entre os processos MPI (troca de mensagens) é significativo e domina o tempo de computação;
- **Granularidade fina:** a quantidade de computação realizada por cada processo entre as comunicações é muito pequena, tornando a comunicação proporcionalmente mais cara;
- **Latência de comunicação:** a latência de cada comunicação pode ser significativa, especialmente em redes com alta latência;
- **Topologia da rede:** como os processos estão conectados fisicamente pode influenciar o desempenho da comunicação.

4. Conclusão

Após todo o desenvolvimento e análises realizadas, é possível observar o quão diferente foi a atuação de cada modelo e seu paralelismo executado. Houve diferenças significativas em seus desempenhos a depender do número de threads/processos utilizados.

As imagens abaixo apresentam um resumo visual dos resultados obtidos. No primeiro gráfico, observa-se o tempo de execução sequencial para cada uma das implementações (OpenMP, CUDA e MPI). Nota-se que a implementação CUDA apresenta um tempo sequencial significativamente menor do que a implementação OpenMP, evidenciando a vantagem da GPU mesmo em execução sequencial. A implementação

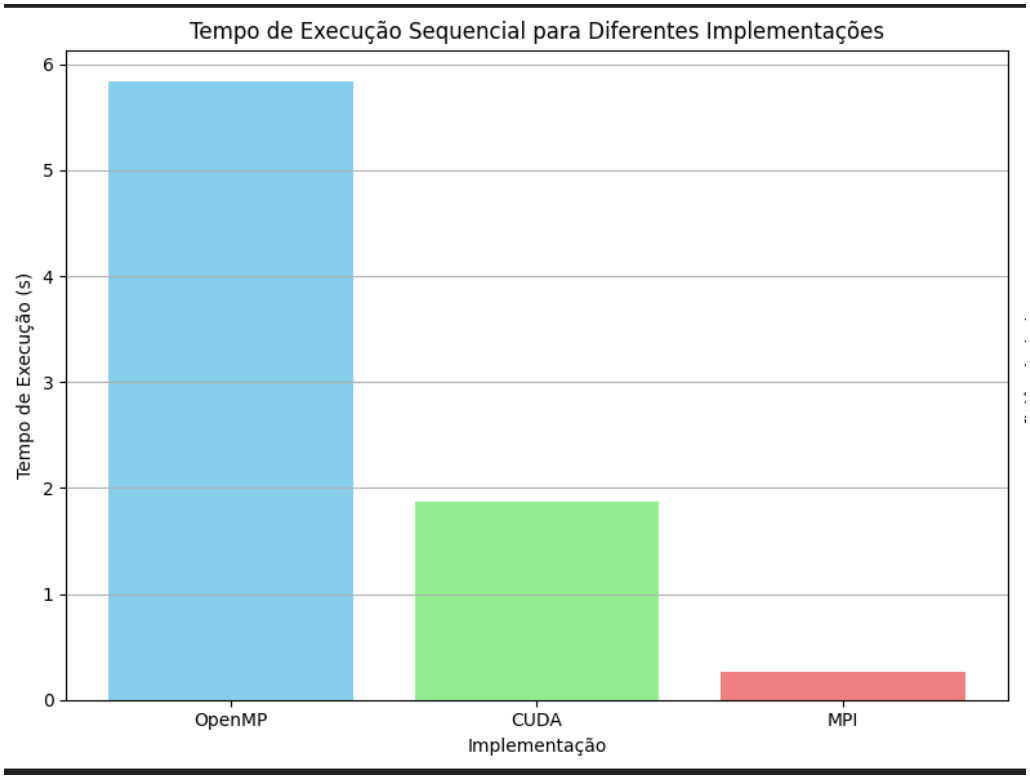

Figure 2. Comparação dos Tempos de Execução

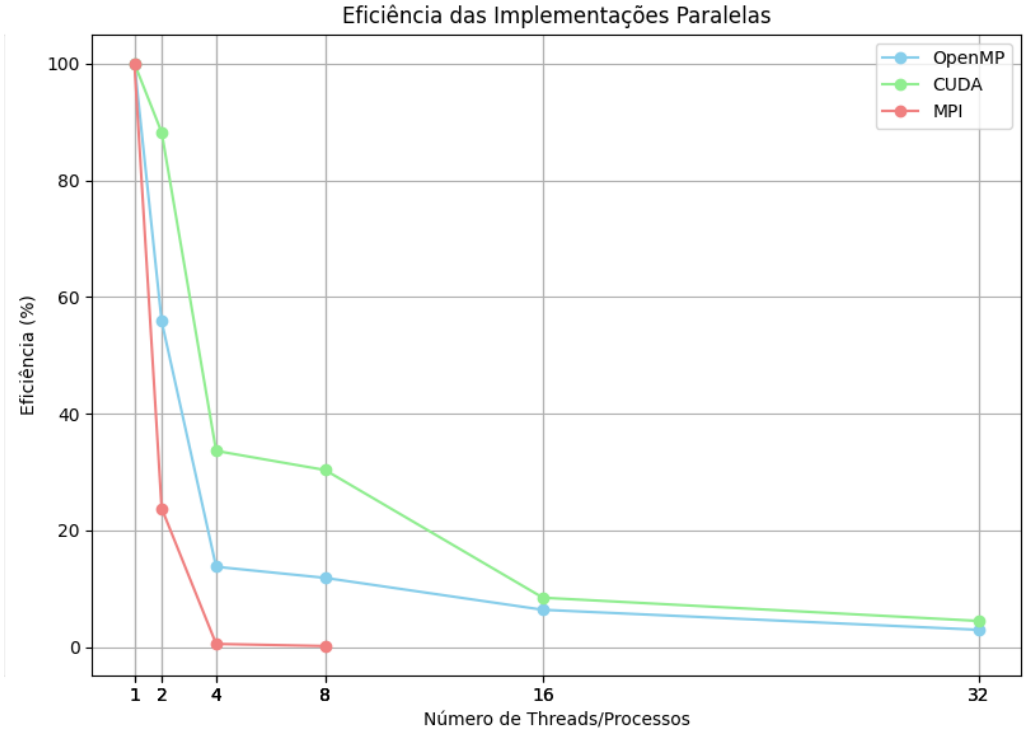

Figure 3. Eficiência das Implementações

MPI, por sua vez, apresenta o menor tempo de execução sequencial, o que pode indicar uma melhor otimização para a arquitetura em questão, ou um menor overhead em

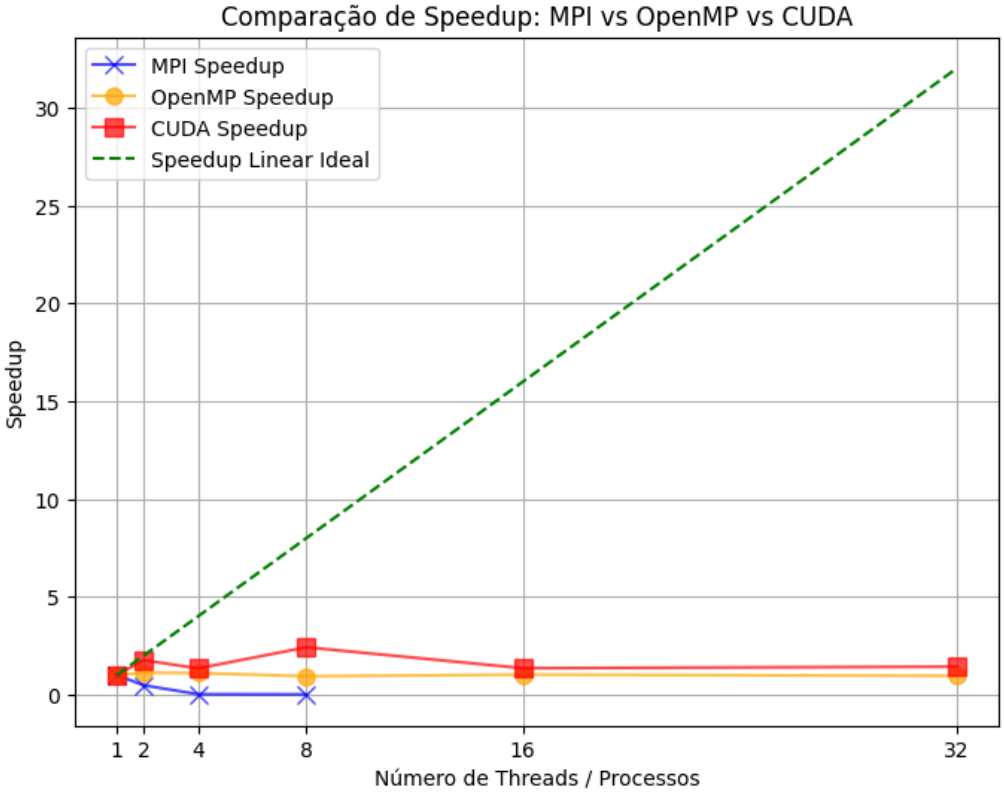

Figure 4. Comparação do Speedup: MPI vs OpenMP vs CUDA

operações básicas.

O gráfico em seguida, mostra a eficiência das implementações paralelas em função do número de threads/processos. Observa-se que a eficiência da implementação OpenMP diminui rapidamente à medida que o número de threads aumenta, indicando uma baixa escalabilidade devido a fatores como sobrecarga de gerenciamento de threads e contenção de recursos. A implementação CUDA apresenta uma eficiência relativamente alta com um pequeno número de threads, mas também sofre uma queda significativa à medida que o número de threads aumenta, sugerindo que a arquitetura da GPU pode estar sendo subutilizada com muitos threads, ou que a comunicação entre a CPU e a GPU se torna um gargalo.

A implementação MPI, por sua vez, apresenta o pior desempenho. A eficiência é baixa mesmo com um pequeno número de processos, e diminui drasticamente com o aumento do número de processos. Esses resultados indicam que o overhead de comunicação entre os processos MPI está dominando o tempo de execução, tornando a execução paralela menos eficiente do que a execução sequencial. Problemas de balanceamento de carga e a própria natureza do algoritmo podem contribuir para esse resultado.

A terceira imagem ilustra um comparativo do speedup dos modelos implementados. Observa-se que o CUDA apresenta o melhor desempenho, mantendo um speedup constante em torno de 5. Isso indica alta eficiência na execução paralela. Por outro lado, o OpenMP atinge um ganho inicial ao dobrar o número de threads, mas estabiliza próximo a 2, demonstrando limitações de escalabilidade. Por sua vez, o MPI apresenta um aumento

inicial, mas o desempenho declina à medida que mais processos são adicionados. Dessa maneira, aqui há a evidência de que a sobrecarga de comunicação impacta negativamente sua escalabilidade. O speedup linear, representando o ideal teórico, infelizmente, não é alcançado, destacando os desafios à implementação de paralelismo eficiente em diferentes métodos de aplicação.

Logo, a análise comparativa das implementações OpenMP, CUDA e MPI revela que a escolha da abordagem de paralelização ideal depende fortemente das características do problema e da arquitetura de hardware disponível. Embora a implementação CUDA mostre um bom desempenho inicial, a escalabilidade limitada e a necessidade de otimização cuidadosa do código para a GPU são desafios importantes. A implementação OpenMP, por sua vez, pode ser adequada para problemas com baixo overhead de comunicação e balanceamento de carga; enquanto a implementação MPI requer uma análise cuidadosa da granularidade da computação e da topologia da comunicação para evitar gargalos de desempenho.

5. Referências

References

[1] Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). **Introduction to Parallel Computing (2nd ed.)**. Addison-Wesley.

[2] SERPA, Matheus; PINTO, Vinícius; NAVAUX, Philippe. **INTRODUÇÃO À PROGRAMAÇÃO VETORIAL E PARALELA PARA O PROCESSADOR INTEL XEON PHI KNIGHTS LANDING**. Programação Paralela OpenMP – Aula 01, 2018. Disponível em: https://www.inf.ufrgs.br/erad2018/downloads/minicursos/eradrs2018-openmp.pdf. Acesso em: 09 jan. 2025.

[3] Introduction to OpenMP. **Research Computing Services**. Disponível em: https://carleton.ca/rcs/rcdc/introduction-to-openmp/.

[4] ALECRIM, Emerson; HIGA, Paulo. O que é a plataforma CUDA da Nvidia e qual sua importância?. **Tecnoblog**, 2023. Disponível em: https://tecnoblog.net/responde/o-que-e-cuda-nvidia/. Acesso em: 09 jan. 2025.

[5] Introduction to MPI. **Research Computing Services**. Disponível em: https://carleton.ca/rcs/rcdc/introduction-to-mpi/.

[6] BERNSTEIN, Any. Contaminantes emergentes na água. **Revista Educação Pública**, Rio de Janeiro, v. 22, nº 34, 13 de setembro de 2022. Disponível em: https://educacaopublica.cecierj.edu.br/artigos/22/34/contaminantes-emergentes-na-agua/.