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 paralela (com OpenMP), 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), 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, 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.

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 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.

3. Diferenças entre as Implementações

As diretivas de paralelismo foram escolhidas para:

- Aproveitar múltiplos núcleos da CPU.
- 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.

Aspecto	Sequencial	Paralela (OpenMP)
Execução	Apenas um núcleo processa todos os cálculos.	Divisão de cálculos entre múltiplos núcleos.
Inicialização	Loops simples para zerar as matrizes.	#pragma omp parallel for para acelerar.
Atualização	Um único núcleo percorre a matriz.	Paralelismo em dois níveis com `collapse`.
Erro Médio	Soma sequencial.	Soma paralela com `reduction`.
Tempo de Execução	Alto para matrizes grandes.	Reduzido proporcionalmente aos núcleos.

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

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. A tabela a seguir calcula o desempenho para diferentes configurações de threads.

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.

5. Gráficos Comparativos

5.1. Speedup, Speedup Linear, Eficiência

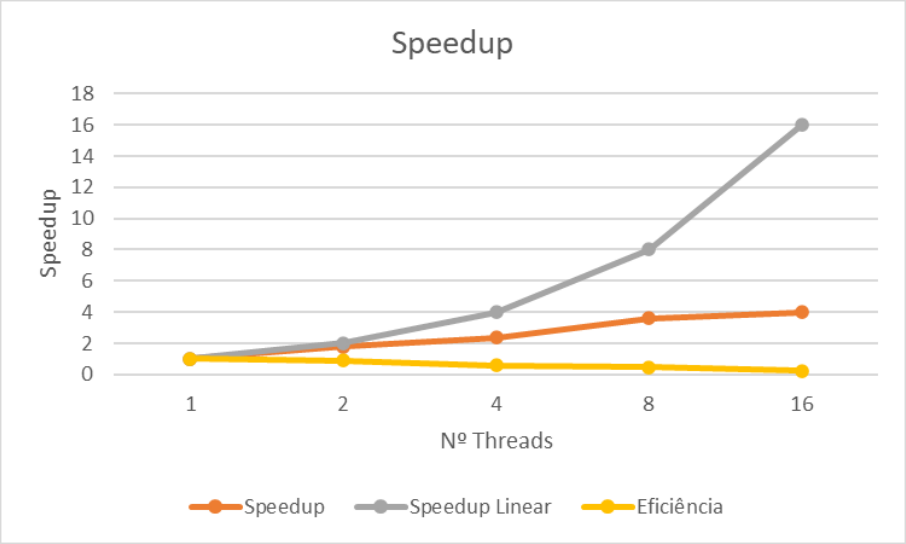

Figure 1. Gráfico Speedup, Speedup Linear, Eficiência

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 **Smax** 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.

Futuras etapas incluem a implementação em CUDA e MPI para explorar maior escalabilidade, especialmente para aproveitar o poder de processamento das GPUs e de clusters de máquinas, além de continuar a investigação sobre as limitações da Lei de Amdahl em sistemas de grande escala.

7. References

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.