Análise de Desempenho Paralelo de Modelos de Difusão de Contaminantes em Água

Eduardo Verissimo Faccio, Pedro Figueiredo Dias,
Pedro Henrique de Oliveira Masteguin

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

{verissimo.eduardo,pedro.figueiredo,p.masteguin}@unifesp.br

1. Introdução

A contaminação de corpos d'água, tais como lagos e rios, é um grande desafio ao meio ambiente e a saúde pública, sendo preciso entender a dispersão do contaminante no ambiente para poder realizar qualquer intervenção de mitigação. Dessa forma, este trabalho foca na modelagem numérica da difusão de poluentes em uma matriz bidimensional, utilizando o método de diferenças finitas para aproximar a equação de difusão discreta:

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

Nesta equação, $C_{i,j}^{t}$ representa a concentração do contaminante na célula (i, j) no instante t, D é o coeficiente de difusão, Δt o intervalo de tempo discreto e Δx o espaçamento espacial. O objetivo principal é desenvolver uma simulação que modele a difusão de contaminantes aplicando programação paralela para acelerar os cálculos e analisar o comportamento dos poluentes ao longo do tempo. Serão comparadas as versões sequencial e paralela do algoritmo, utilizando OpenMP, CUDA e MPI para explorar o processamento simultâneo em múltiplos núcleos e dispositivos. Os resultados serão validados por meio de mapas de calor, gráficos de speedup e eficiência, além da comparação das matrizes geradas. Este estudo demonstra como técnicas de programação concorrente e distribuída podem otimizar simulações numéricas complexas, reforçando os conceitos aprendidos na disciplina e demonstrando sua aplicação prática no desenvolvimento de soluções eficientes.

2. Implementação do Algorítimo

2.1. Código Sequencial

O código sequencial implementa a solução numérica da equação de difusão usando uma abordagem serial. Utilizando-se do método de diferenças finitas, é simulado a dispersão de uma substância em uma matriz bidimensional. Cada célula da matriz representa a concentração de uma substância em um ponto do espaço.

O cálculo é realizado em um laço de repetição que itera sobre todas as células da matriz. A atualização de cada célula depende da média das concentrações dos seus vizinhos imediatos e de parâmetros físicos como coeficiente de difusão, o intervalo de tempo Δt e o espaçamento espacial Δx.

```c
double sequential_diff_eq(double **C, double **C_new, DiffEqArgs *args)
    {
    int N = args->N;
    double D = args->D;
    double DELTA_T = args->DELTA_T;
    double DELTA_X = args->DELTA_X;
    double difmedio = 0.;

    for (int i = 1; i < N - 1; i++) {
        for (int j = 1; j < N - 1; j++) {
            C_new[i][j] = C[i][j] + D * DELTA_T * ((C[i + 1][j] + C[i -
                1][j] + C[i][j + 1] + C[i][j - 1] - 4 * C[i][j]) / (
               DELTA_X * DELTA_X));
            difmedio += fabs(C_new[i][j] - C[i][j]);
        }
    }

    return difmedio / ((N - 2) * (N - 2));
}
```

Código 1. Código sequencial da para cálculo da difusão, que será utilizado como base para as demais implementações.

2.2. Código Paralelo em OpenMP

A implementação paralela utiliza a biblioteca Open Multi-Processing (OpenMP) para distribuir a carga de trabalho entre múltiplos núcleos, mantendo a lógica do algoritmo sequencial. Essa distribuição é realizada através de diretivas específicas inseridas na estrutura do código original.

```c
double omp_diff_eq(double **C, double **C_new, DiffEqArgs *args) {
    int N = args->N;
    double D = args->D;
    double DELTA_T = args->DELTA_T;
    double DELTA_X = args->DELTA_X;
    double difmedio = 0.;

#pragma omp parallel for collapse(2) reduction(+ : difmedio)
    for (int i = 1; i < N - 1; i++) {
        for (int j = 1; j < N - 1; j++) {
            C_new[i][j] = C[i][j] + D * DELTA_T * ((C[i + 1][j] + C[i -
                1][j] + C[i][j + 1] + C[i][j - 1] - 4 * C[i][j]) / (
               DELTA_X * DELTA_X));
            difmedio += fabs(C_new[i][j] - C[i][j]);
        }
    }

    return difmedio / ((N - 2) * (N - 2));
}
```

Código 2. Implementação paralelizada utilizando a biblioteca OpenMP.

`#pragma omp parallel for collapse(2):` Esta diretiva divide automaticamente os laços de cálculo entre diversos *threads*. Cada *thread* é responsável por

atualizar uma parte distinta da matriz, permitindo que várias seções do cálculo sejam processadas simultaneamente. O parâmetro `collapse(2)` indica que os dois laços aninhados serão combinados para otimizar o paralelismo.

`#pragma omp parallel for reduction(+:difmedio) collapse(2):` Assim como a diretiva anterior, esta instrução paraleliza as iterações do laço, porém com o acréscimo de uma operação de redução. Essa redução garante que o somatório, utilizado para monitorar a convergência do algoritmo, seja realizado de forma segura entre os diferentes *threads*.

`omp_set_num_threads(int num_threads):` Esta função permite configurar dinamicamente o número de *threads* a serem utilizados, sendo essencial para testes e análises de desempenho.

2.3. Código Paralelo em CUDA

Nesta implementação, o algoritmo de difusão foi otimizado para execução em GPUs, aproveitando a arquitetura CUDA (Compute Unified Device Architecture). A abordagem distribui o cálculo da atualização das células da matriz de concentração entre milhares de `threads`, onde cada uma processa individualmente uma célula da matriz utilizando o método de diferenças finitas. Essa estratégia maximiza a paralelização, resultando em desempenho superior quando comparada às implementações sequencial e mesmo às paralelas em CPU.

```
__global__ void diffusion_kernel(double *C, double *C_new, double *
    block_sums, int N, double D, double DELTA_T, double DELTA_X) {
    extern __shared__ double sdata[];

    int i = blockIdx.y * blockDim.y + threadIdx.y + 1;
    int j = blockIdx.x * blockDim.x + threadIdx.x + 1;

    double diff_val = 0.0f;

    if (i < N - 1 && j < N - 1) {
        int idx = i * N + j;
        double up = C[(i - 1) * N + j];
        double down = C[(i + 1) * N + j];
        double left = C[i * N + (j - 1)];
        double right = C[i * N + (j + 1)];
        double center = C[idx];

        C_new[idx] = center + D * DELTA_T * ((up + down + left + right
            - 4 * center) / (DELTA_X * DELTA_X));

        diff_val = fabs(C_new[idx] - center);
    }

    int tid = threadIdx.y * blockDim.x + threadIdx.x;
    sdata[tid] = diff_val;
    __syncthreads();

    for (unsigned int s = (blockDim.x * blockDim.y) / 2; s > 32; s >>=
        1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
```

```
        }
        __syncthreads();
    }

    if (tid < 32) {
        volatile double *vsmem = sdata;
        vsmem[tid] += vsmem[tid + 32];
        vsmem[tid] += vsmem[tid + 16];
        vsmem[tid] += vsmem[tid + 8];
        vsmem[tid] += vsmem[tid + 4];
        vsmem[tid] += vsmem[tid + 2];
        vsmem[tid] += vsmem[tid + 1];
    }

    if (tid == 0) {
        block_sums[blockIdx.y * gridDim.x + blockIdx.x] = sdata[0];
    }
}
```

Código 3. Implementação paralelizada utilizando CUDA.

O kernel `diffusion_kernel` é responsável por calcular a nova concentração de cada célula com base nos valores dos seus vizinhos imediatos, aplicando a fórmula de difusão que incorpora os parâmetros de difusão, tempo e espaço. Este kernel é lançado em uma grade (*grid*) composta por blocos de `threads`, cujas dimensões podem ser configuradas dinamicamente através da função `set_block_dimensions`. A utilização de memória compartilhada (`__shared__`) acelera a acumulação dos valores diferenciais, facilitando a verificação da convergência do algoritmo.

Etapas do Fluxo de Execução:

1. **Inicialização (`cuda_init`):** Aloca a memória necessária na GPU e transfere os dados iniciais da CPU para a memória do dispositivo.
2. **Execução do Kernel (`cuda_diff_eq`):** Lança o kernel que atualiza os valores da matriz e calcula a diferença média entre iterações.
3. **Recuperação dos Dados (`cuda_get_result`):** Transfere os resultados computados na GPU de volta para a CPU.
4. **Finalização (`cuda_finalize`):** Libera a memória previamente alocada na GPU.

Para facilitar a implementação do algoritmo *stencil* na GPU, a matriz bidimensional foi convertida em um vetor unidimensional, onde as linhas são concatenadas. Essa abordagem simplifica o acesso à memória, exigindo apenas um cálculo cuidadoso dos índices para que cada *thread* acesse o elemento correto.

Para a operação de redução, cada *thread* inicialmente armazena sua contribuição em memória compartilhada. Em seguida, o kernel realiza uma soma hierárquica, agregando os valores de forma progressiva até que reste um único valor por bloco. Esse resultado final é copiado para a memória global, possibilitando o cálculo da diferença média total já na CPU e, consequentemente, a verificação da convergência do algoritmo.

Adicionalmente, a função `set_block_dimensions` possibilita o ajuste das dimensões dos blocos de *threads*, permitindo a experimentação com diferentes granularida-

des de paralelismo para um balanceamento eficiente da carga entre os multiprocessadores da GPU.

2.4. Código Paralelo em MPI

O último código paralelo foi desenvolvido utilizando MPI (Message Passing Interface) e OpenMP para combinar processamento distribuído e paralelismo em nível de threads. O MPI é uma biblioteca de comunicação que permite a troca de dados entre processos independentes em um ambiente de computação distribuída. Ele é perfeito para aplicações que exigem aplicações que exigem alto desempenho e escalabilidade, possibilitando a comunicação eficiente entre processos distribuídos em diferentes nós de um cluster. A combinação de MPI e OpenMP permite a execução paralela em múltiplos núcleos de processamento em cada nó.

Etapas do Fluxo de Execução:

1. **Inicialização:** Inicia com MPI_Init_thread(), definindo o número de processos e threads OpenMP. Cada processo MPI recebe sua parte da matriz global.
2. **Troca de dados entre processo:** Os processos MPI trocam as fronteiras das suas submatrizes com processos vizinhos usando comunicação assíncrona (MPI_Isend() e MPI_Irecv()).
3. **Calculo Paralelo:** Cada processo MPI aplica a equação diferencial na sua submatriz, paralelizando o cálculo com OpenMP (#pragma omp parallel for).
4. **Sincronização global:** A diferença entre os estados da matriz é calculada localmente e reduzida globalmente com MPI_Allreduce(), garantindo consistência entre os processos.
5. **Finalização:** Após completar todas as iterações, a memória é liberada e MPI_Finalize() é chamado. O tempo total de execução é registrado e exibido pelo processo mestre.

Esse fluxo de execução garante que a carga de trabalho seja distribuída de maneira eficiente entre os processos MPI, enquanto OpenMP melhora o desempenho dentro de cada nó de processamento.

```
double mpi_omp_diff_eq(double **C, double **C_new, DiffEqArgs *args,
                       int localN, int N, int rank, int size) {
    double D = args->D;
    double DELTA_T = args->DELTA_T;
    double DELTA_X = args->DELTA_X;
    double difmedio_local = 0.0;

    MPI_Request reqs[4];
    int req_count = 0;

    if (rank > 0) {
        MPI_Irecv(C[0], N, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &
            reqs[req_count++]);
        MPI_Isend(C[1], N, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD, &
            reqs[req_count++]);
    }

    if (rank < size - 1) {
        MPI_Irecv(C[localN + 1], N, MPI_DOUBLE, rank + 1, 1,
            MPI_COMM_WORLD, &reqs[req_count++]);
```

```
        MPI_Isend(C[localN], N, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD
            , &reqs[req_count++]);
    }

    MPI_Waitall(req_count, reqs, MPI_STATUSES_IGNORE);

    double local_sum = 0.0;
#pragma omp parallel for collapse(2) reduction(+ : local_sum)
    for (int i = 1; i <= localN; i++) {
        for (int j = 1; j < N - 1; j++) {
            C_new[i][j] =
                C[i][j] +
                D * DELTA_T *
                    ((C[i + 1][j] + C[i - 1][j] + C[i][j + 1] + C[i][j
                        - 1] - 4.0 * C[i][j]) / (DELTA_X * DELTA_X));

            local_sum += fabs(C_new[i][j] - C[i][j]);
        }
    }

    double global_sum = 0.0;
    MPI_Allreduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM,
        MPI_COMM_WORLD);

    double difmedio_global = global_sum / ((N - 2) * (N - 2));

    return difmedio_global;
}
```

Código 4. Implementação paralelizada utilizando MPI.

2.5. Interface Python e ferramenta CMake

O projeto utiliza o CMake como sistema de compilação para gerenciar processos e dependências tanto da implementação em OpenMP quanto da versão CUDA, definindo tudo por meio de arquivos de configuração. Para destacar as diferenças de desempenho entre as abordagens sequencial, OpenMP e CUDA, as otimizações do compilador — como a vetorização automática que inicialmente minimizava essas disparidades — são desabilitadas por meio de flags específicas, permitindo uma execução mais direta e comparável.

Paralelamente, foi implementada uma interface Python usando o módulo *ctypes* para carregar dinamicamente as bibliotecas compiladas e mapear suas funções, facilitando a integração com métodos de análise e visualização desenvolvidos em notebooks Jupyter. Inspirada em bibliotecas como o NumPy, essa abordagem combina a eficiência das implementações em C com a flexibilidade e a facilidade de uso do Python, possibilitando a configuração de parâmetros como tamanho da matriz, coeficiente de difusão e dimensões dos blocos de *threads*.

```
from diffusion import (
    SequentialDiffusionEquation,
    OMPdiffusionEquation,
    CUDADiffusionEquation,
)

```

```
lib_path = "./build/libDiffusionEquation.so"

with SequentialDiffusionEquation(
    library_path=lib_path, N=200, D=0.05, DELTA_T=0.02, DELTA_X=1.0,
    initial_concentration_points={(100, 100): 1.0},
) as seq_solver:
    for _ in range(1000): # Perform 1000 simulation steps
        diff_seq = seq_solver.step()  # Execute seq C code step

    value_at_center = seq_solver.concentration_matrix[100][100]
    print(f"Sequential diffusion value at center: {value_at_center}")

with OMPdiffusionEquation(
    library_path=lib_path, N=200, D=0.05, DELTA_T=0.02, DELTA_X=1.0,
    initial_concentration_points={(100, 100): 1.0},
) as omp_solver:
    for _ in range(1000):
        diff_omp = omp_solver.step()  # Execute OpenMP step

    value_at_center = omp_solver.concentration_matrix[100][100]
    print(f"OMP diffusion value at center: {value_at_center}")

with CUDADiffusionEquation(
    library_path=lib_path, N=200, D=0.05, DELTA_T=0.02, DELTA_X=1.0,
    initial_concentration_points={(100, 100): 1.0},
) as cuda_solver:
    for _ in range(1000):
        diff_cuda = cuda_solver.step()  # Execute cuda step

    cuda_solver.get_result() # Get the result from device to host
    value_at_center = cuda_solver.concentration_matrix[100][100]
    print(f"CUDA diffusion value at center: {value_at_center}")
```

Código 5. Implementação paralelizada utilizando CUDA.

3. Resultados

Nesta seção, apresentamos os resultados obtidos de nossa implementação. Inicialmente, analisamos a equivalência lógica entre os códigos sequencial e paralelo, considerando possíveis erros que podem surgir na paralelização, como condições de corrida ou inconsistências de sincronização. Em seguida, ilustramos, por meio de mapas de calor, a atualização dos valores da matriz ao longo do tempo. Por fim, realizamos uma análise comparativa dos tempos médios de execução e *speedup* entre as duas versões.

3.1. Validação da Implementação - Numérico

Para assegurar a correção das duas implementações, verificamos em cada iteração se os valores presentes em cada célula da matriz são idênticos. Dessa forma, o resultado na última iteração deve ser o mesmo em ambas as versões.

Por meio desse procedimento, utilizando a interface Python em conjunto com um Jupyter Notebook, comprovamos que as duas soluções produzem resultados idênticos. Isso era esperado, pois no código paralelo não ocorrem condições de corrida, uma vez que a escrita não é realizada na mesma região de memória das leituras, tornando o processamento de cada célula pelas *threads* independente.

3.2. Validação da Implementação - Ilustrativo

Para ilustrar o funcionamento da implementação, foram gerados mapas de calor, representado pela Figura 1, nos quais cada ponto de uma matriz 50×50 é representado por uma cor distinta. Cores escuras correspondem a valores próximos de um, indicando alta concentração do contaminante, enquanto cores claras representam valores próximos de zero, indicando baixa presença de contaminação.

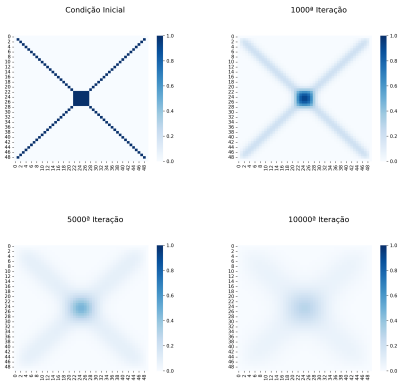

Figura 1. Mapa de calor em quatro instantes distintos da simulação.

Analisando a progressão dos mapas de calor, observamos que o comportamento faz sentido no contexto da solução proposta. Inicialmente, o contaminante é adicionado com alta concentração nas diagonais e no centro da matriz, evidenciado pelas regiões azuis-escuros. Com o avanço das iterações, o contaminante começa a se difundir para as regiões adjacentes, aumentando gradativamente a luminosidade nessas áreas e diminuindo nos pontos de concentração inicial. Na última iteração, a concentração se distribui uniformemente pela matriz, com valores próximos entre si.

3.3. Análise de Desempenho

A análise de desempenho foi realizada em um computador *desktop* com as especificações apresentadas na Tabela 1. Ademais, as especificações dos parâmetros do problema foram incluídas na Tabela 2. Note que a simulação do código MPI foi executada com variações no número de threads, porém mantendo em apenas uma única thread OMP.

Tabela 1. Tabela de especificação de Hardware

Especificações	Detalhes
Processador	Intel i7–7500U @ 2.7GHz–3.5GHz
Núcleos / Lógicos	2 / 4
Memória RAM	10 GB
Sistema Operacional	Ubuntu 22.04.05 (via WSL)
GPU	NVIDIA GeForce 940MX - 2GB VRAM

Para obter valores mais consistentes e minimizar influências externas, como outros programas em execução, cada teste foi executado quinze vezes e, assim, calculamos o

Tabela 2. Tabela de especificação da Simulação

Especificações	Detalhes
Dimensão da Matriz (N $\times$ N)	3000 $\times$ 3000
Número de Iterações	1000
Distribuição Inicial	Alta concentração no centro
Coeficiente de Difusão	0.1
Δt	0.01
Δx	1.0

tempo médio gasto e seu desvio padrão. O *speedup* é calculado dividindo-se o tempo de execução sequencial pelo tempo de execução CUDA correspondente.

Tabela 3. Tabela de comparação de desempenho entre o código sequencial e o utilizando MPI.

Experimento	Tempo	SpeedUp
Sequencial	27.23 $\pm$ 2.36	1.0
OMP (1T)	26.33 $\pm$ 1.09	1.03
OMP (2T)	16.58 $\pm$ 1.72	1.64
OMP (4T)	13.34 $\pm$ 1.52	2.04
OMP (8T)	13.38 $\pm$ 1.54	2.35
OMP (16T)	13.75 $\pm$ 1.42	1.98
OMP (32T)	13.96 $\pm$ 1.46	1.95
MPI (1P)	25.47 $\pm$ 1.64	1.07
MPI (2P)	16.51 $\pm$ 1.61	1.65
MPI (4P)	13.53 $\pm$ 1.52	2.01
MPI (8P)	15.88 $\pm$ 1.71	1.71
MPI (16P)	18.03 $\pm$ 1.40	1.51
MPI (32P)	19.60 $\pm$ 1.51	1.39
CUDA (16, 16)	5.23 $\pm$ 1.04	5.21

Os resultados apresentados na Tabela 3 estão em conformidade com as expectativas, considerando as especificações de hardware descritas na Tabela 1. Dentre as abordagens testadas, a implementação em CUDA destacou-se por alcançar o melhor desempenho. Esse resultado se deve, principalmente, à capacidade das GPUs de realizar um paralelismo massivo – permitindo a execução simultânea de centenas de *threads* –, bem como à gestão eficiente da memória. Em contrapartida, as implementações com OpenMP e MPI esbarram nas limitações impostas pelos 4 núcleos lógicos da CPU. Enquanto o OpenMP consegue mitigar essa restrição com a criação de *threads* adicionais sem perdas expressivas de desempenho, o MPI sofre com um *overhead* considerável decorrente da criação de novos processos e da comunicação via troca de mensagens.

A eficiência do processamento paralelo evidenciada pela abordagem CUDA ressalta o potencial das GPUs para tarefas que exigem cálculos intensivos, como os algoritmos de aprendizado de máquina. Contudo, em simulações de menor escala os benefícios

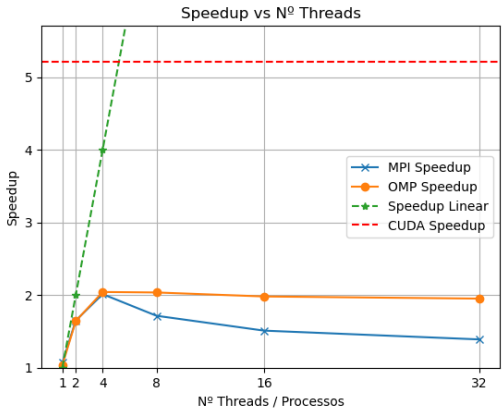

Figura 2. Gráfico de Speedup para experimentos da Tabela 3.

podem ser menos significativos, pois os custos de alocação, transferência e gerenciamento dos dados na GPU podem superar as vantagens do paralelismo. Além disso, embora o MPI apresente limitações em ambientes com poucos núcleos lógicos, sua utilização em *clusters* distribuídos pode explorar um paralelismo massivo, resultando em desempenho potencialmente superior.

4. Conclusão

Com as implementações paralelas utilizando CUDA, MPI e OpenMP, além da versão sequencial, validou-se a exatidão dos resultados e analisou-se o impacto do paralelismo no tempo de execução. A estratégia com CUDA explorou o potencial massivo das GPUs para reduzir significativamente os tempos de processamento, apesar dos desafios de comunicação entre CPU e GPU e das limitações arquiteturais que podem afetar sua escalabilidade. Em contrapartida, a abordagem com MPI mostrou-se eficaz na distribuição da carga entre nós em ambientes distribuídos, mesmo considerando o custo adicional de comunicação, enquanto o OpenMP proporcionou ganhos expressivos em sistemas multicore, apesar da limitação imposta pelo número de núcleos lógicos.

Assim, o estudo ressalta a importância de compreender as peculiaridades de cada técnica paralela. As GPUs destacam-se em simulações intensivas, MPI possibilita a expansão dos recursos computacionais em clusters e o OpenMP se mostra eficiente em ambientes multicore. Essa combinação de abordagens oferece uma visão abrangente dos desafios e benefícios da aceleração por hardware paralelo, contribuindo para a otimização de algoritmos numéricos em contextos computacionais exigentes.

Referências

Crank, J. (1979). *The Mathematics of Diffusion*. Oxford science publications. Clarendon Press.

Rauber, T. and Runger, G. (2013). *Parallel Programming*. Springer.