Simulação e Análise de Modelos de Difusão de Contaminantes em Água utilizando técnicas concorrentes

Gabriel Amaral[1], **Lucas Kuroki**[1]

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

kenzo.kuroki@unifesp.br

Resumo. *Este artigo descreve uma implementação de codigo de uma grade NxN, onde o objetivo é torna-la concorrente, utilizando recursos como colab e gcc time, para diferentes implementações, openMp, mpi, sequencial e CUDA, testou-se diversas quantidades de recursos, variando o numero de processos e threads, foi possível verificar qual o ganho de cada uma.*

1. Introdução

A difusão de partículas em um meio, tem um papel significativo em áreas como física, química, engenharia e ciência de dados. A equação de difusão bidimensional, que descreve a evolução temporal da concentração de uma certa substância, é um modelo clássico para estudar processos de transporte. No entanto, trabalhar com cálculos de matrizes demanda recursos computacionais consideráveis, especialmente quando se deseja computar equações aninhadas e substituições de linhas de matrizes.

Neste contexto, técnicas de computação concorrente surgem como ferramentas essenciais para acelerar simulações e viabilizar estudos que seriam muito demorados com abordagens sequenciais. Este projeto tem como objetivo explorar a aplicação de três ferramentas de paralelismo, OpenMP (para paralelismo em memória compartilhada), CUDA (para aceleração utilizando GPUs) e MPI (para computação distribuída em processos), ao código sequencial de difusão fornecido. O trabalho não apenas implementa cada técnica, mas também analisa seu desempenho, escalabilidade, precisão e complexidade de implementação, destacando as diferenças entre elas.

A motivação central reside na necessidade de otimizar simulações científicas para problemas de larga escala, onde a redução do tempo de execução é crítica. O código original, que utiliza uma abordagem explícita de diferenças finitas em uma grade 2D, serve como um estudo de caso ideal para explorar como diferentes modelos de concorrência podem explorar arquiteturas modernas (multicore, GPUs, clusters). O relatório está organizado da seguinte forma: primeiro, será revisado o modelo matemático da difusão e a discretização utilizada no código. Em seguida, cada técnica de paralelismo será detalhada, com ênfase nas modificações importantes no código sequencial, estratégias de divisão de trabalho e sincronização. Por fim, os resultados serão comparados por meio de métricas como speedup, eficiência e escalabilidade, culminando em uma discussão sobre os cenários em que cada abordagem é mais vantajosa. Este estudo não apenas mostra o potencial da computação de alto desempenho (HPC) para problemas científicos, mas também serve como um guia prático para a transição de códigos sequenciais para arquiteturas paralelas.

2. Descrição do Problema

A simulação de difusão em uma grade bidimensional de tamanho 2000×2000 implementada no código sequencial, envolve um alto custo computacional devido à natureza iterativa do método de diferenças finitas. O alto custo reside na função diffEq, que atualiza a concentração C em cada ponto da grade ao longo de 1000 iterações temporais. A complexidade algorítmica é O(T*N**2). Desta forma o desafio para implementar as técnicas de concorrência se resume a distribuir laços da função de difusão para threads e GPUs, ou dividir o trabalho desta função entre os processos. Abaixo temos o trecho com os laços que precisam de paralelismo.

```
#define N 2000  // Tamanho da grade
#define T 1000 // Numero de iteracoes no tempo
void diff_eq(double **C, double **C_new) {
    for (int t = 0; t < T; t++) {
        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)
                );
            }
        }
        double difmedio = 0.;
        for (int i = 1; i < N - 1; i++) {
            for (int j = 1; j < N - 1; j++) {
                difmedio += fabs(C_new[i][j] - C[i][j]);
                C[i][j] = C_new[i][j];
            }
        }
        if ((t%100) == 0)
          printf("interacao %d - diferenca=%g\n", t, difmedio/((
             N-2)*(N-2)));
    }
}
```

Um código implementado de forma sequencial demanda de varias iterações uma atrás da outra, demandando tempo de execução maior e também desperdiça recursos modernos, ou seja, tem escalabilidade baixa. Consdiderando estes pontos, observa-se que cada iteração espacial (i,j) é independente dentro de um mesmo passo temporal(t), permitindo paralelização por decomposição de domínio (distribuir linhas/colunas entre threads ou processos), e para a atualização de Cnew para C é necessário sincronização, pois ocorre depois dos calculos da linha de C.

3. Implementação e Testes

As implementações paralelas descritas neste projeto foram desenvolvidas com o objetivo de explorar as diferenças entre os paradigmas de concorrência(OpenMP, CUDA e MPI) em um problema de difusão 2D. Para maior equidade, testamos os códigos na mesma máquina.

Abaixo temos a função principal do código sequencial, ele inicia a grade com 1 no centro e envia para a função, que por sua vez realiza estes cálculos e retorna a concentração no centro e imprime na tela, essa logica se mantém próxima para as demais implementações.

```c
int main() {
    double **C = (double **)malloc(N * sizeof(double *));
    if (C == NULL) {
      fprintf(stderr, "Memory allocation failed\n");
      return 1;
    }
    for (int i = 0; i < N; i++) {
      C[i] = (double *)malloc(N * sizeof(double));
      if (C[i] == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
      }
    }
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
        C[i][j] = 0.;
      }
    }
    // Concentracao para a proxima iteracao
    double **C_new = (double **)malloc(N * sizeof(double *));
    if (C_new == NULL) {
      fprintf(stderr, "Memory allocation failed\n");
      return 1;
    }
    for (int i = 0; i < N; i++) {
      C_new[i] = (double *)malloc(N * sizeof(double));
      if (C_new[i] == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
      }
    }
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
        C_new[i][j] = 0.;
      }
    }
    C[N/2][N/2] = 1.0;
    diff_eq(C, C_new);
    printf("Concentracao final no centro: %f\n", C[N/2][N/2]);
    return 0;
}
```

Para cada uma, vamos executar com time, dessa forma, ele retorna o tempo para executar o código:

```
gcc diff.c -o diff
time ./diff
Iteracao 0 - Diferenca media=2.00401e-09
Iteracao 100 - Diferenca media=1.23248e-09
Iteracao 200 - Diferenca media=7.81794e-10
Iteracao 300 - Diferenca media=5.11528e-10
Iteracao 400 - Diferenca media=4.21632e-10
Iteracao 500 - Diferenca media=3.62223e-10
Iteracao 600 - Diferenca media=3.05976e-10
Iteracao 700 - Diferenca media=2.57135e-10
Iteracao 800 - Diferenca media=2.21174e-10
Iteracao 900 - Diferenca media=2.00244e-10
Concentracao final no centro: 0.095045

real    1m7,130s
user    1m7,087s
sys 0m0,037s
```

Listing 1. Comandos e resultados da execução do código Sequencial

3.1. Implementação com OpenMp

O Open Mp trabalha com o paradigma de programação de memória compartilhada, onde *threads* executam o programa em paralelo e compartilham a mesma memória([SILVA 2022]). Para decompor a tarefa em openMp é relativamente simples, busca-se paralelizar partes que demandam computação intensa, neste caso os laços aninhados. Com openMp utiliza-se diretiva pragma omp(linhas em C com significado diferente em compiladores), junto coloca-se a diretiva *parallel* que distribui um trecho de código com um time de *threads* especificado na linha de comando. Caso necessário, é possível especificar se certas variáveis podem ou não ser acessadas por todas as *threads* através de cláusulas como private([OpenMp]).

Foi implementado linhas com diretivas openMp para tratar dos laços aninhados da função diffEq, os trechos mencionados estão logo abaixo. Para além das diretivas, foi utilizado também a cláusula reduction e for, na primeira ele avisa que uma variável específica receberá a operação de redução, e na segunda é indicado que o laço for deve ser separado entre as *threads*.

```
    void diff_eq(double **C, double **C_new) {
    for (int t = 0; t < T; t++) {
        #pragma omp parallel for
        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)
                );
            }
        }
        // Atualizar matriz para a proxima iteracao
        double difmedio = 0.0;
```

```
        #pragma omp parallel for reduction(+:difmedio)
        for (int i = 1; i < N - 1; i++) {
            for (int j = 1; j < N - 1; j++) {
                difmedio += fabs(C_new[i][j] - C[i][j]);
                C[i][j] = C_new[i][j];
            }
        }
        if ((t % 100) == 0) {
            printf("Iteracao %d - Diferenca media=%g\n", t,
               difmedio / ((N-2) * (N-2)));
        }
    }
}
//funcao principal semelhante
```

Com código que implementa openMp, é necessário avisar o compilador para executar as diretivas, isso é feito nas linhas de comando, junto com a especificação da quantidade de *threads* que executam o código. Dessa forma o trecho de código dentro da diretiva é dividido entre as *threads*, como foi dito, o openMp faz esse balanceamento de forma automática, portanto basta especificar os trechos em que se deseja executar em concorrência([SILVA 2022]).

```
gcc -o diff_omp -o diff_omp.c -fopenmp -lm
time ./diff_omp 2

real    0m35,456s
user    1m10,761s
sys     0m0,062s
```

O recorte acima mostra o tempo de execução para quantidades de *threads* diferentes, está informação é útil para gerar medidas de desempenho como *speedup* e possibilitar futuras comparações com outras implementações. E logo após a execução, é colocado o número de *threads* para trabalhar. Vale notar que os prints das interações foram omitidas pois são redundantes.

3.2. Implementação com CUDA

O CUDA nos fornece recursos para utilizar a GPU em paralelo para realizar a computação do programa, permitindo a criação de blocos que interagem entre si e implementa recursos para realizar calculos vetoriais e lidar com memória compartilhada([Nvidia]). No código abixo, temos na diffEq uma grade de threads, com cada thread atualizando uma célula da matriz. Uma função auxiliar atomicAddDouble, que realiza somas atômicas, e um kernel computeDifference, que realiza os cálculos, nele, cada *thread* é responsável por calcular a diferença absoluta entre as matrizes e o vetor armazena elas localmente dentro de cada bloco, realizando um reduction atraves da thread 0, que acumula os valores da global diff.

Na main, aloca-se a memória na GPU, por meio do cudaMalloc e reserva espaço, realizando uma cópia do host para a GPU, em seguida foi configurado a quantidade de grids e blocos, no caso, 16*16= 256 threads e o grid se ajusta com o N. No bloco de

iterações no tempo, atualizamos a nova matriz de concentração, e a cada 100 iterações zeramos o ddiff. Um novo cálculo por meio da função computeDifference, pega a diferença acumulada para o host e imprime esta diferença a cada 100 iterações. E no final, colocamos a matriz atualizada no host e imprimimos no terminal a concentração do centro.

```
#define N 2000
#define T 1000

__global__ void diff_eq(double *C, double *C_new) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i * N + j;
    if (i > 0 && i < N-1 && j > 0 && j < N-1) {
        C_new[index] = C[index] + D * DELTA_T * (
            (C[index + N] + C[index - N] + C[index + 1] + C[
               index - 1] - 4 * C[index]) / (DELTA_X * DELTA_X)
        );
    }
}

static __device__ double atomicAddDouble(double* address, double
    val) {
    unsigned long long int* address_as_ull = (unsigned long long
        int*)address;
    unsigned long long int old = *address_as_ull, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
           __double_as_longlong(val + __longlong_as_double(
           assumed)));
    } while (assumed != old);

    return __longlong_as_double(old);
}

__global__ void compute_difference(double *C, double *C_new,
   double *dif) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i * N + j;
    __shared__ double dif_per_block[256];
    int t_idx = threadIdx.x + threadIdx.y * blockDim.x;

    if (i < N && j < N) {
        dif_per_block[t_idx] = fabs(C_new[index] - C[index]);
    } else {
        dif_per_block[t_idx] = 0.0;
    }
    __syncthreads();

```

```
40      for (int s = blockDim.x * blockDim.y / 2; s > 0; s >>= 1) {
41          if (t_idx < s) {
42              dif_per_block[t_idx] += dif_per_block[t_idx + s];
43          }
44          __syncthreads();
45      }
46
47      if (t_idx == 0) {
48          atomicAddDouble(dif, dif_per_block[0]);
49      }
50  }
51
52  int main() {
53      // Alocacao de memoria no host
54      // Inicializacao da memoria C, C new e inicia o centro
55
56      // Alocacao de memoria na GPU
57      double *d_C, *d_C_new, *d_dif;
58      cudaMalloc((void **)&d_C, N * N * sizeof(double));
59      cudaMalloc((void **)&d_C_new, N * N * sizeof(double));
60      cudaMalloc((void **)&d_dif, sizeof(double));
61
62      // Copiar dados do host para a GPU
63      cudaMemcpy(d_C, C, N * N * sizeof(double),
            cudaMemcpyHostToDevice);
64      cudaMemcpy(d_C_new, C_new, N * N * sizeof(double),
            cudaMemcpyHostToDevice);
65
66      // Dimensoes do bloco e da grade
67      dim3 threadsPerBlock(16, 16);
68      dim3 blocksPerGrid((N + threadsPerBlock.x - 1) /
            threadsPerBlock.x, (N + threadsPerBlock.y - 1) /
            threadsPerBlock.y);
69
70      // Iteracoes no tempo
71      for (int t = 0; t < T; t++) {
72          diff_eq<<<>>>(d_C, d_C_new);
73
74          if ((t % 100) == 0) {
75              cudaMemset(d_dif, 0, sizeof(double));
76              compute_difference<<<>>>(d_C, d_C_new, d_dif);
77
78              double difmedio;
79              cudaMemcpy(&difmedio, d_dif, sizeof(double),
                    cudaMemcpyDeviceToHost);
80              difmedio /= ((N-2)*(N-2));
81              printf("Iteracao %d - Diferenca media=%g\n", t,
                    difmedio);
82          }
```

```
83
84          // Trocar os ponteiros
85          double *temp = d_C;
86          d_C = d_C_new;
87          d_C_new = temp;
88      }
89
90      // Copiar resultado final da GPU para o host
91      cudaMemcpy(C, d_C, N * N * sizeof(double),
            cudaMemcpyDeviceToHost);
92
93      // Exibir resultado final no centro da grade
94      printf("Concentracao final no centro: %f\n", C[(N/2) * N + (
            N/2)]);
95
96      // Liberar memoria e return
97  }
```

Executando o código no colab, tem-se o seguinte resultado no terminal:

```
1  nvcc -o diffusion diffusion.cu -lm
2  time ./diffusion
3  !nvcc -o diffusion diffusion.cu -lm
4  !time ./diffusion
5
6  real 0m1.536s
7  user 0m1.073s
8  sys 0m0.297s
```

3.3. Implementação com MPI

Agora, utilizando uma abordagem de sistemas distribuídos, foi utilizado o MPI, uma api que subdivide códigos entre processos e fornece meios para a comunicação entre eles. Diferente das outras abordagens, para este caso o processo de distribuição de tarefas ocorre por parte do programador, ou seja, em caso de falhas de comunicação é possível travar o sistema com deadlocks([Mpi-forum]).

A abordagem desta api é por meio de funções, são diversas, cada uma delas nos auxilia tanto na comunicação como na execução de trechos de código, são amplamente utilizadas em sistemas distribuídos, visando alto desempenho e simplicidade([COULOURIS 2013]). A estrutura básica contém o MPIInit e o MPIFinilize, ambos sincronizam os processos no início e no fim, respectivamente. Durante a execução do programa, dependendo do tipo da tarefa, podemos gerar subdivisões com granularidade grossa ou fina([SILVA 2022]), para esta implementação, faremos testes para 2, 4 e 8 processos, gerando grãos distintos, que podem ser analisados posteriormente. Para melhorar o desempenho de cálculos sobre uma grade, utilizou-se a divisão de tarefas por linhas, nesta abordagem é como se cada processo fosse responsável por uma matriz menor(levando a operações menores, pois a matriz é menor), e foi preciso que linhas auxiliares fossem inclusas para que após as operações nas linhas, fosse

possível a comunicação para os vizinhos, afinal, a operação para cada linha(i) depende da linha anterior(i-1) e da linha posterior(i+1). Visando um balanceamento justo, separamos a carga de tarefa através de ifs que manipulam a quantidade total de processos, desta forma foi possivel gerar cargas de trabalho bem semenlhante entre os m processos.

```
void diff_eq(double **C, double **C_new, int local_n, int rank,
   int numProc) {
    for (int t = 0; t < T; t++) {
        int previous = rank - 1;
        int next     = rank + 1;

        if (rank == 0) {
        MPI_Sendrecv(&C[local_n][0],N, MPI_DOUBLE,next, 0,
                     &C[local_n+1][0],N, MPI_DOUBLE,next, 1,
                     MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        } else if (rank == numProc - 1) {
            MPI_Sendrecv(&C[1][0],N, MPI_DOUBLE,previous, 1,
                         &C[0][0],N, MPI_DOUBLE,previous, 0,
                         MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        } else {
            MPI_Sendrecv(&C[1][0],N, MPI_DOUBLE,previous, 1,
                         &C[0][0],N, MPI_DOUBLE,previous, 0,
                         MPI_COMM_WORLD,MPI_STATUS_IGNORE);
            MPI_Sendrecv(&C[local_n][0],N, MPI_DOUBLE,next, 0,
                         &C[local_n+1][0],N, MPI_DOUBLE,next, 1,
                         MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
        for (int i = 1; i <= local_n; 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));
            }
        }
        double difmedio_local = 0.0;
        for (int i = 1; i <= local_n; i++) {
            for (int j = 1; j < N - 1; j++) {
                difmedio_local += fabs(C_new[i][j] - C[i][j]);
                C[i][j] = C_new[i][j];
            }
        }
        double difmedio_total = 0.0;
        MPI_Reduce(&difmedio_local, &difmedio_total, 1,
           MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        if (rank == 0 && t % 100 == 0) {
            printf("Iteracao %d - diferenca=%g\n", t,
               difmedio_total / ((N-2)*(N-2)));
        }
    }
}
```

Listing 2. Função diffEq

No código, é possível perceber a manipulação para gerar os grão(matrizes de tamanho localN +2), e na função diffEq, é onde ocorre os calculos das matrizes menores, que agora percorren¥m matrizes menores, porém com a adição de uma sequência de comunicações através da MPISendRecv, que trata de receber a ultima linha da matriz anterior para sua primeira linha e enviar a sua ultima para a primeira linha da próxima matriz, com excessão da primeira e ultima matriz, onde essa lógica é facilmente observada no código. Logo após a comunicaçãos entre as linhas, executa-se as operações necessárias que são reduzidas por meio da função MPIReduce(função que aplica um tipo de operação em celulas especificadas). Ao terminar a execução de diffEq, realizamos um *Gather* que pega o valor da difusão média e junta em uma única variável para exibir na tela. É importante observar que o MPISendRecv é implementado livre de deadlocks, se fosse utilizado de maneira separada, seria necessário tratar através de um ordenamento lógico, ou algo semelhante.

```
1 mpicc -o mpi_exe mpi_diff.c -lm
2 time mpirun -np 8 ./mpi_exec
3
4 real    0m11,493s
5 user    1m28,387s
6 sys 0m0,779s
```

Listing 3. Comandos e resultados da execução do código

Sua execução segue os moldes do openMp, neste caso utiliza-se 8 processos na execução, e os testes variam de 2 a 8 processos.

4. Resultados e Discussão

Agora comparando as implementações temos a tabela abaixo, nela temos uma divisão das linhas por cores. Uma medida fundamental é o speedup, representa o ganho em relação ao sequencial, e a eficiência mostra a relação entre o ganho e o número de threads/processos.

A eficiência auxilia na visualização da escalabilidade, afinal, o programa é o mesmo e a medida em que se aumenta a quantidade de recursos(processos/threads), não necessariamente o desempenho aumenta, pois pela lei de amdhal sempre existe uma parcela do codigo que é sequencial, dessa forma quanto maior a quantidade de recursos não necessariamente o speedup sera maior, isto reflete na eficiência do código, porém em termos de tempo, a evolução é bem evidende então mesmo que exista um ”desperdício” de recursos, para projetos de engenharia e científicos o tempo gasto com operações é bem significativo.

Olhando para a execução do Cuda percebe-se um speedup otimo em relação as outras implementações, agora olhando para o openMp e mpi, as medidas são próximas, mas aparentemente a medida em que os recursos crescem o openMp tende a ser uma opção melhor.

Numero de threads/processos	Tempo (s)	Speedup	Eficiencia
Serial	67,13	1	100%
2(OpenMP)	35,456	1,893	94,67
4(OpenMP)	18,022	3,725	93,12
8(OpenMP)	10,611	6,326	79,08
2(MPI)	34,824	1,928	96,38
4(MPI)	17,955	3,739	93,47
8(MPI)	11,496	5,839	72,99
CUDA	1,536	43,704	X

Figure 1. Medidas de desempenho usando implementação sequencial, openMp, CUDA e mpi

5. Conclusão

É importante considarar o tipo de problema, neste projeto, como estamos calculando entre células vizinhas e atualizando matrizes, dessa forma, para as implementações haverá um limite em relação a granularidade, como neste programa precisamos de 3 linhas para atulaizar para quantidades de recursos muito grandes haverá uma dependência considerável entre as tarefas, alem da necessidade de comunicar e sincronização entre processos e threads.

6. References

References

COULOURIS, George; DOLLIMORE, J. K. T. e. a. (2013). *Sistemas distribuídos. 5. ed.* Bookman.

Mpi-forum. *https://www.mpi-forum.org/*.

Nvidia. *https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html*.

OpenMp. *https://www.openmp.org/*.

SILVA, G. P.; BIANCHINI, C. P. C. E. B. (2022). *MPI, OpenMP e OpenACC para computação de alto desempenho.* Casa do Código.