Simulação e Análise de Modelos de Difusão de Contaminantes em Água utilizando Mpi

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 trabalho visa modelar a difusão de concentração em uma grade bidimensional. Foi utilizado o mpi com o intuito de melhorar o desempenho de um codigo sequencial. Os resultados mostram um desempenho bom sob certas circunstâncias em relação ao sequencial e ao openMp.*

1. Introdução

Para a modelagem de uma difusão de contaminantes em água, temos um código sequencial e com o intuito de melhorar seu desempenho, utilizamos o mpi(*Message Passing Interface*), uma api que permite a divisão de trabalho entre processos, e desta forma, dividimos trechos da grade para os N processos e avaliamos o seu desempenho.

1.1. Metodologia

Na mesma máquina que foi implementada o codigo sequencial e com openMp, utilizamos para testar a versão com mpi. Para facilitar a visualização e reduzir a redundância, vamos mostrar trechos de código que diferenciam a versão mpi da sequencial.

No codigo abaixo temos a função diffeq(), ela possui mais parametros que a versão sequencial, o *rank* que contem o id do processo, a quantidade de processos e o localN que contém a quantidade de linhas que cada processo opera. O funcionamento é bem simples, vamos pegar a grade NxN e dividir pelo número de processos desejados, sendo assim, cada processo vai ser responsável por uma fração da matriz. Na função, vamos verificar qual o processo atual, para garantir uma comunicação correta, se for 0 temos que enviar a ultima linha da sua matriz, se for o ultimo temos que mandar a sua primeira linha para o seu antecessor e se for intermediario temos que distribuir a primeira e a ultima linha, isso é necessário pois o calculo depende dos vizinhos da célula da grade(esquerda, baixo, cima e direita). A função MPISendrecv entra em ação para efetuar essa troca de linhas, e seu diferencial é que ela é livre de *deadlocks*. Depois de comunicar entre processos efetuamos os cálculos(usando o MPIReduce), atualizamos a grade e a cada 100 interações imprimimos a difusão média na tela.

```
1 void diff_eq(double **C, double **C_new, int local_n, int rank,
     int numProc) {
2     for (int t = 0; t < T; t++) {
3         int previous = rank - 1;
4         int next     = rank + 1;
5
6         if (rank == 0) {
7         // Processo 0: apenas troca com o proximo.
```

```
8           MPI_Sendrecv(&C[local_n][0],N, MPI_DOUBLE,next, 0,
9                        &C[local_n+1][0],N, MPI_DOUBLE,next, 1,
10                       MPI_COMM_WORLD,MPI_STATUS_IGNORE);
11      } else if (rank == numProc - 1) {
12          // ultimo processo: apenas troca com o anterior.
13          MPI_Sendrecv(&C[1][0],N, MPI_DOUBLE,previous, 1,
14                       &C[0][0],N, MPI_DOUBLE,previous, 0,
15                       MPI_COMM_WORLD,MPI_STATUS_IGNORE);
16      } else {
17          // Processos intermedi rios: troca com o anterior e com
                o proximo
18          // Troca com o anterior:
19          MPI_Sendrecv(&C[1][0],N, MPI_DOUBLE,previous, 1,
20                       &C[0][0],N, MPI_DOUBLE,previous, 0,
21                       MPI_COMM_WORLD,MPI_STATUS_IGNORE);
22          // Troca com o proximo:
23          MPI_Sendrecv(&C[local_n][0],N, MPI_DOUBLE,next, 0,
24                       &C[local_n+1][0],N, MPI_DOUBLE,next, 1,
25                       MPI_COMM_WORLD,MPI_STATUS_IGNORE);
26      }
27
28          for (int i = 1; i <= local_n; i++) {
29              for (int j = 1; j < N - 1; j++) {
30                  C_new[i][j] = C[i][j] + D * DELTA_T *
31                      ((C[i+1][j] + C[i-1][j] + C[i][j+1] + C[i][j
                            -1] - 4 * C[i][j])
32                       / (DELTA_X * DELTA_X));
33              }
34          }
35
36          double difmedio_local = 0.0;
37          for (int i = 1; i <= local_n; i++) {
38              for (int j = 1; j < N - 1; j++) {
39                  difmedio_local += fabs(C_new[i][j] - C[i][j]);
40                  C[i][j] = C_new[i][j];
41              }
42          }
43
44          double difmedio_total = 0.0;
45          MPI_Reduce(&difmedio_local, &difmedio_total, 1,
                MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
46          if (rank == 0 && t % 100 == 0) {
47              printf("Iteracao %d - diferenca=%g\n", t,
                    difmedio_total / ((N-2)*(N-2)));
48          }
49      }
50  }
```

Listing 1. Função diffEq

Na função principal, temos o trabalho de balancear a carga entre os processos alguns calculos simples e ajustes do resto são necessários. E para garantir que a comunicação funcione, temos que incluir duas linhas a mais, para receber as linhas dos processos vizinhos. Logo após executar a diffeq() vamos chamar a função Gather da api, para o calculo final do centro.

```
int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    int rank, numProc;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numProc);

    int base = N / numProc;
    int resto = N % numProc;
    int local_n = (rank < resto) ? base + 1 : base;
    int inicio = 1 + ((rank < resto) ? rank*(base+1) : resto*(
       base+1) + (rank-resto)*base);
    int final = inicio + local_n - 1;

    /*Criando matrizes com N/numProc linhas, sendo
    adicionado linhas nos extremos para a comunicacao*/

    double **C = (double **)malloc((local_n + 2) * sizeof(double
        *));
    double **C_new = (double **)malloc((local_n + 2) * sizeof(
       double *));

    for (int i = 0; i < local_n + 2; i++) {
        C[i] = (double *)malloc(N * sizeof(double));
        C_new[i] = (double *)malloc(N * sizeof(double));
        for (int j = 0; j < N; j++) {
            C[i][j] = 0.0;
            C_new[i][j] = 0.0;
        }
    }

    int center_y = (N / 2)-1;
    int center_x = center_y - inicio+1;

    if (center_y >= inicio && center_y <= final) {
        C[center_x][center_y] = 1.0;

    }
    diff_eq(C, C_new, local_n, rank, numProc);
    double conc_center = 0.0;
    if (center_y >= inicio && center_y <= final) {
        conc_center = C[center_x][center_y];
    }

    double *final_conc = NULL;
```

```
42      if (rank == 0) final_conc = (double *)malloc(numProc *
            sizeof(double));
43      MPI_Gather(&conc_center, 1, MPI_DOUBLE, final_conc, 1,
            MPI_DOUBLE, 0, MPI_COMM_WORLD);
44
45      if (rank == 0) {
46          for (int i = 0; i < numProc; i++) {
47              if (final_conc[i] > 0) {
48                  printf("Concentra o final no centro: %f\n",
                        final_conc[i]);
49                  break;
50              }
51          }
52          free(final_conc);
53      }
54
55      // Liberar memoria
56      for (int i = 0; i < local_n + 2; i++) {
57          free(C[i]);
58          free(C_new[i]);
59      }
60      free(C);
61      free(C_new);
62
63      MPI_Finalize();
64      return 0;
65  }
```

Listing 2. função principal usando mpi

2. Resultados e Discussões

Utilizando o time, podemos obter informações para comparar com outras versões. No terminal abaixo estamos exibindo para 8 processos, este mesmo codigo foi compilado com 2, 4 e 8 processos.

```
1  mpicc -o mpi_exe mpi_diff.c -lm
2  time mpirun -np 8 ./mpi_exec
3  Iteracao 0 - Diferen a media=2.00401e-09
4  Iteracao 100 - Diferen a media=1.23248e-09
5  Iteracao 200 - Diferen a media=7.81794e-10
6  Iteracao 300 - Diferen a media=5.11528e-10
7  Iteracao 400 - Diferen a media=4.21632e-10
8  Iteracao 500 - Diferen a media=3.62223e-10
9  Iteracao 600 - Diferen a media=3.05976e-10
10 Iteracao 700 - Diferen a media=2.57135e-10
11 Iteracao 800 - Diferen a media=2.21174e-10
12 Iteracao 900 - Diferen a media=2.00244e-10
13 Concentracao final no centro: 0.095045
14
```

```
real    0m11,493s
user    1m28,387s
sys 0m0,779s
```

Listing 3. Comandos e resultados da execução do código

Na tabela abaixo podemos comparar o desempenho do codigo sequencial, usando mpi(em azul) e com openMP. Usamos o mesmo numero de threads e processos e testamos na mesma maquina, para que a comparação seja justa.

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

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

Em relação ao código sequencial a vantagem é visível, agora comparando com o openMp, percebe-se uma leve vantagem quando se utiliza poucos processos(2 e 4), mas observando o openMp com 8 *threads* e o mpi com 8 processos, a eficiencia do mpi cai significativamente. Isso faz sentido pois a medida em que acrescentamos processos estamos dividindo a grade principal em maior número de matrizes, oque implica em um número muito maior de troca de mensagens(chamadas de MPISendRecv) entre os processos, reduzindo drasticamente a vantagem que teriamos ao dividir a carga de trabalho. Vale observar que utilizar um número muito grande de processos não garante uma vantagem na execução das operações, pois é necessário levar em consideração a granularidade adequada para que a concorrência melhore o desempenho.