U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Resolvendo o problema do menor caminho

Paralelização através MPI com o algoritmo de *Fox*

Computação Paralela
CC4014

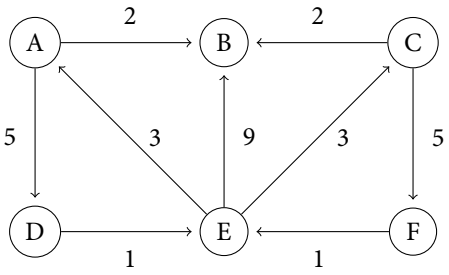
Tarso Boudet Caldas
up202204297

Porto — 2023

Sumário

1 Introdução

O problema de encontrar o menor caminho (*All pairs shortest path*) é aquele em que buscamos o menor caminho entro quaisquer dois pares de nós em grafo direcionado. A Figura 1 exemplifica o tipo de grafo que tratamos e a matriz que usamos para representar este mesmo grafo, onde cada entrada M_{ij} é a distância da aresta que vai do nó i ao nó j.



$$M = \begin{bmatrix} 0 & 2 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 9 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Figura 1: Grafo direcionado com valores atribuídos para cada aresta e a matriz quadrada correspondente.

A solução deste problema é também uma matriz quadrada, onde cada entrada N_{ij} é o disância do menor caminho possível a ser percorrida para ir do nó i ao j.

$$N = \begin{bmatrix} 0 & 2 & 9 & 5 & 6 & 14 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 2 & 0 & 14 & 6 & 5 \\ 4 & 6 & 4 & 0 & 1 & 9 \\ 3 & 5 & 3 & 8 & 0 & 8 \\ 4 & 6 & 4 & 9 & 1 & 0 \end{bmatrix}$$

Neste trabalho, iremos descrever uma solução para este problema adotando paralelismo através de *Message Passing Interface* (MPI) usando o *algoritmo de Fox*, distribuindo processos entre diversas configurações de máquinas e os organizando em um comunicador de *grid*.

2 Solucionando o problema

Para encontrar a matriz N dos menores caminhos a partir de M, elevar M ao quadrado sucessivamente até que $N = M^{2^{\log(n-1)}}$, onde n é o número de nós do grafo. Para isso, implementamos o seguinte algoritmo de multiplicação de matrizes apresentado no Código 1.

```
54 void matrixMultiply(int** matrixA, int** matrixB, int** matrixC, int size) {
55   int i, j, k, min;
56   for (i = 0; i < size; i++) {
57     for (j = 0; j < size; j++) {
58       min = NULLPATH;
59       for (k = 0; k < size; k++) {
60         if (matrixA[i][k] > 0 && matrixB[k][j] > 0) {
61           if ((min < 0) || (matrixA[i][k] + matrixB[k][j]) < min)
62             min = (matrixA[i][k] + matrixB[k][j]);
63         }
64         if ((min ≥ 0) && (matrixC[i][j] < 0 || matrixC[i][j] > min))
65           matrixC[i][j] = min;
66       }
67     }
68   }
69 }
```

Código 1: Algoritmo de multiplicação de matrizes no arquivo `mtrxop.c`.

O que difere este algoritmo de uma multiplicação é que em vez de multiplicar as entradas $A_{i,k}$ e $B_{k,j}$, com $k = 1, \ldots, n$, e somar os respectivos k resultados, somamos $A_{i,k}$ com $B_{i,k}$ e escolhemos os menor resultado (ignorando as entradas com valor 0). Com isso encontramos o *produto da distância* entre duas matrizes, e repetindo este processo sucessivamente, até M^{n-1}, encontramos a matriz de distância, e portanto, resolvemos o problema.

3 Paralelizando a multiplicação através do algoritmo de Fox

O *algoritmo de Fox* nos permite dividir a tarefa de multiplicação da matriz em diversos processos fazendo multiplicações de matrizes menores em cada um. Para isso, dividimos a matriz em um *grid*, isto é, uma malha com divisões iguais ao número de processos. Por exemplo, na matriz M, poderiamos ter

$$M_0 = \begin{bmatrix} 0 & 2 & 9 \\ 0 & 0 & 0 \\ 9 & 2 & 0 \end{bmatrix}, \quad M_1 = \begin{bmatrix} 5 & 6 & 14 \\ 0 & 0 & 0 \\ 14 & 6 & 5 \end{bmatrix}, \quad M_2 = \begin{bmatrix} 4 & 6 & 4 \\ 3 & 5 & 3 \\ 4 & 6 & 4 \end{bmatrix}, M_3 = \begin{bmatrix} 0 & 1 & 9 \\ 8 & 0 & 8 \\ 9 & 1 & 0 \end{bmatrix}.$$

Com MPI, podemos definir o grid como um *struct* `GRID_TYPE`, como mostra o Código 2.

```
6  typedef struct {
7    int nprocs;
8    MPI_Comm comm;
9    MPI_Comm row_comm;
10   MPI_Comm col_comm;
11   int gridOrder;
12   int row;
13   int col;
14   int rank;
15 } GRID_TYPE;
```

Código 2: Definição do tipo `GRID_TYPE` no arquivo `mtrxop.h`

Aqui definimos três comunicadores, um para o grid, um para linhas e outro para colunas, além de definirmos o número de processos, a ordem (lado) do grid, além de ter a informação da linha e coluna no grid. Usamos então para inicar o *grid* a função `setupGrid(GRID_TYPE)`, definida no Código 3.

```
85  void setupGrid(GRID_TYPE* grid) {
86    int rank, dimensions[2], periods[2], coordinates[2], varying_coords[2];
87
88    MPI_Comm_size(MPI_COMM_WORLD, &(grid->nprocs));
89    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
90    grid->gridOrder = (int)sqrt(grid->nprocs);
91    dimensions[0] = dimensions[1] = grid->gridOrder;
92    periods[0] = periods[1] = 1;
93    MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, periods, 1, &(grid->comm));
94    MPI_Comm_rank(grid->comm, &(grid->rank));
95    MPI_Cart_coords(grid->comm, grid->rank, 2, coordinates);
96    grid->row = coordinates[0];
97    grid->col = coordinates[1];
98
99    varying_coords[0] = 0;
100   varying_coords[1] = 1;
101   MPI_Cart_sub(grid->comm, varying_coords, &(grid->row_comm));
102   varying_coords[0] = 1;
103   varying_coords[1] = 0;
104   MPI_Cart_sub(grid->comm, varying_coords, &(grid->col_comm));
105 }
```

Código 3: Inicialização do *grid* definida em `mtrxop.c`.

Por fim, definimos no Código 4 função que aplica o algoritmo de Fox para efetuar a multiplicação de matrizes. Este algoritmo escolhe uma submatriz A de cada linha e a envia para os outros processos daquela linha, e então cada processo multiplica a matriz recebida com uma matriz B presente previamente (inicialmente igual a A). Cada processo então envia a matriz B para o processo acima no *grid*.

```c
void Fox(GRID_TYPE* grid, int** matrixA, int** matrixB, int** matrixC,
         int** tempMatrix, int size) {
  int bcast_root, source, dest;
  int tag = 0;

  source = (grid->row + 1) % grid->gridOrder;
  dest = (grid->row + grid->gridOrder - 1) %
         grid->gridOrder;

  int step;
  for (step = 0; step < grid->gridOrder; step++) {
    bcast_root = (grid->row + step) % grid->gridOrder;
    if (bcast_root == grid->col) {
      MPI_Bcast(matrixA[0], size * size, MPI_INT, bcast_root, grid->row_comm);
      matrixMultiply(matrixA, matrixB, matrixC, size);
    } else {
      MPI_Bcast(tempMatrix[0], size * size, MPI_INT, bcast_root,
                grid->row_comm); // &tempA[0][0] redundante
      matrixMultiply(tempMatrix, matrixB, matrixC, size);
    }
```

Código 4: Algoritmo de Fox.

4 Execução do programa

Para executar o programa, basta compilar o projeto usando `make`, e então executar como seguem as instruções no Código 5

```
Usage: mpirun -np <processes> shortest-path [options]
    -i <file> --input <file>    Set matrix input file
    -o <file> --output <file>   Specify custom output name
    -h --help                   Display this help message
    -t --timing-only            Don't print the solution, only the timing
    -r --random-matrix <size>   Use random matrix as input
                                (only if no input file provided)
```

Código 5: Funcionalidade do programa `shortest-path`.

É necessário que o número de processos seja um quadrado perfeito, a fim de que estes possam ser organizados em um *grid*.

Podemos rodar o programa tanto utilizando um arquivo com uma matriz em que a primeira linha contém o número de nós do grafo, ou então gerar uma matriz aleatória de tamanho arbitrário. Em ambos os casos, as matrizes planificadas em um *array* `totalMatrix[]` de tamanho $n \times n$, que é então populado nas matrizes A de cada processo através de `MPI_Scatter()`. Compiamos então a matriz A para b e C, e podemos aplicar o algoritmo de Fox até que tenhamos a matriz de distância, como ilustra o Código 6.

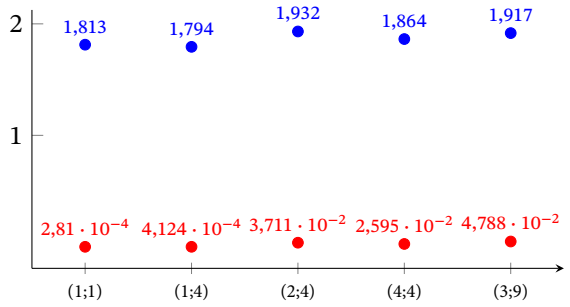
```c
  int g = 1;
  int** matrixB = initMatrix(submtrxSize);
  int** matrixC = initMatrix(submtrxSize);
  int** tempMatrix = initMatrix(submtrxSize);
  copyMatrix(matrixA, matrixB, submtrxSize);
  copyMatrix(matrixA, matrixC, submtrxSize);
  while (g < mtrxSize) {
    Fox(&grid, &matrixA[0], &matrixB[0], &matrixC[0], tempMatrix, submtrxSize);
    copyMatrix(&matrixC[0], &matrixA[0], submtrxSize);
    copyMatrix(&matrixC[0], &matrixB[0], submtrxSize);
    g *= 2;
```

Código 6: Loop do algoritmo Fox sobre as matrizes A e B de cada processo.

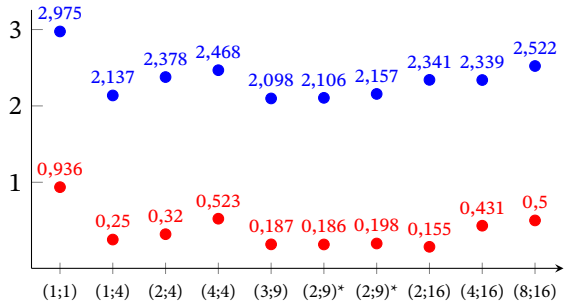
Terminamos usamos `MPI_Gather()` para copiar as matrizes de volta para o array `totalMatrix[]` e temos o nosso resultado.

5 Análise de performance

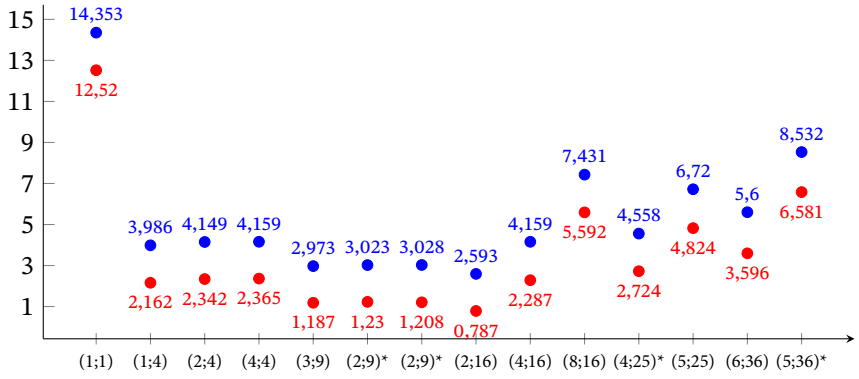
Para analisar a o desempenho deste programa, o rodamos com diferentes configurações de matrizes, de processos e número de máquinas em comunicação. Cada configuração foi corrida 50 vezes através de um script, e foram registrados o tempo de cada corrida através de `MPI_Wait()` e também registrado o tempo total de rodar as 50 vezes.No primeiro caso, não é incluída o tempo de comunicação entre as máquinas, portanto é possível registrar o tempo de execução do processo em si, como também a diferença de tempo de comunicação. Nos gráficos da Figura 2 temos o resultado de tempo.



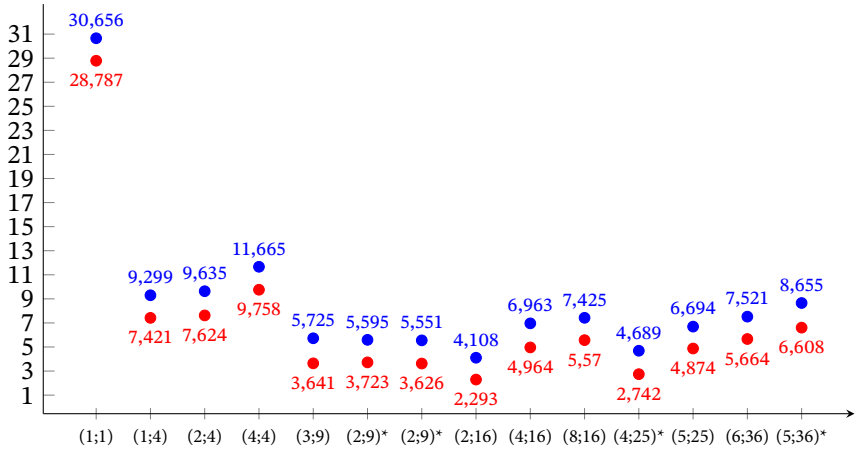
(a) Tempos de matrizes de ordem 6.



(b) Tempos de matrizes de ordem 300.



(c) Tempos de matrizes de ordem 600.



(d) Tempos de matrizes de ordem 900.

Figura 2: Gráficos com os tempos das matrizes. Os pontos vermelhos representam as médias tempo de resolução do problema já com a comunicação iniciada, e os pontos azuis as médias de tempo execução da chamada de `mpirun` 50 vezes. No eixo x temos as configurações das máquinas no formado $c_i = (m; p)$, onde m representa o número de máquinas usadas e p o de processos. As configurações com "$*$" indicam que a distribuição dos processos entre as máquinas não é balanceada.

A partir dos gráficos, é possível analisar que para matrizes pequenas como na Figura 2a, a paralelização não faz tanto sentido, já que a diferença de tempo é negativa, quanto mais processos e máquinas utilizamos para resolver o problema, visto que o tempo de comunicação do MPI é o gargalo da aplicação.

A diferença passa a ser perceptível (mas ainda tímida) quando olhamos os casos das matrizes de 300 nós, onde o uso de

4 processos em uma máquina é quase $\frac{1}{4}$ do de um processo, mas é possível observar que distribuir estes processos entre um número maior de máquinas cria um *overhead* significativo, o que nos leva a crer que é preferível rodar os processos em um número menor de máquinas possível. No gráfico da Figura 2b, o melhor resultado é quando usamos 16 processos, mas apenas quando são executados em duas máquinas, e ainda assim, o tempo total de execução foi maior neste caso do que quando se usam 9 processos. Os casos de 9 processos são notáveis, pois contradiz a observação feita anteriormente, pois o melhor resultado é dividindo em 3 máquinas em vez de duas, onde ambas configurações $c_i = [p_1, \dots, p_m]$, do número de processos da máquina i foram $c_7 = [4, 5]$ (uma máquina com 4 processos e outra com 5) e $c_7 = [8, 1]$, respectivamente, e a diferença é pouco notável entre os três casos, apesar de c_5 possuir uma vantagem sobre $c_5 = (3; 9)$ no tempo de execução do processo, o tempo total de execução de c_5 foi ligeiramente melhor.

É muito mais considerável a diferença entre um e mais processos chegamos a matrizes de ordem 600, onde 1 processo demora seis vezes mais do que 4 em uma máquina. Também vemos na Figura 2c que $c_8 = (2; 16)$ continua a ser dar o melhor resultado na resolução, e isto também se repente nas matrizes de ordem 900. É mais considerável no caso de 600 nós do que no de 300 a discrepância na duração do programa quando introduzimos mais máquinas e mantemos o número de processos, com uma importante exceção que é quando temos $c_{12} = (6; 36)$, que consegue ter vantagem sobre a configuração com estes processos distribuídos em apenas 5, que é o caso de $c_{13} = [8, 8, 8, 8, 4]$. Isto talvez ocorra pois c_{12} se enquadra melhor na definição do comunicador em *grid*.

O gráfico da Figura 2d possui a mesma tendência do anterior, sua principal variação é que em matrizes maiores, aumentar o número de processos após 16 não cria um *overhead* tão considerável, possivelmente pois o tempo de processamento em si começa a alcançar o *bottleneck* dos comunicadores. É possível que se passarmos a lidar com matrizes ainda maiores, configuração com um número de processadores maiores consigam term um melhor desempenho do que a configuração c_8.

6 Conclusão

Com este trabalho foi possível chegar à conclusão que o tamanho do problema é fundamental quando queremos encontrar o ponto ótimo de eficiência da distribuição de processos usando MPI. Quando o problema é pequeno demais, o custo da comunicação não compensa as possíveis vantagens do paralelismo, e mesmo quando consideramos problemas maiores, nem sempre mais máquinas ou mais processos significa uma melhoria no tempo de execução. Por este motivo, antes de se decidir uma configuração para executar um programa em paralelo, é sempre importante ter os *benchmarks* adequados para o tipo e tamanho de problema que estamos lidando.