# Project Assignment I: All-Pairs Shortest Path Problem

Emanuel Rodrigues de Sousa Tomé

(200702634)

Parallel Computing 2020/2021

MSc Data Science

Department of Computer Science

Faculty of Sciences of the University of Porto

## 1 Introduction

The problem to be solved consists in determining all shortest paths between pairs of nodes in a given graph, known as all-pairs shortest path problem. Given a graph G = (V,E) in which V is the set of nodes and E is the set of links between nodes $(v_i, v_j)$, the goal is to determine for each pair of nodes the size of the shortest path that begins in $v_i$ and ends in $v_j$.

The application to be developed must accept as input a file with the description of the graph to be considered and as output the print of the solution of the problem represented by the matrix $D_f$.

The main goal of the present work is to solve the all-pairs shortest path problem using parallel programming for distributed memory environment using the MPI framework. For that propose, the Fox algorithm used for parallel matrix multiplication was implemented. However, note that the operations multiplication and sum in a traditional matrix multiplication are here replaced by operations of sum and minimum. Hereafter, when the terminology "matrix multiplication" is used, it is referring to these operations.

To compile the developed program, one should do in the terminal the following command:

```
mpicc fox.c -o fox.out -lm
```

To run in, the following command should be executed:

```
mpirun -np 16 --hostfile hosts --map-by node fox.out input300
```

where hosts are the file with the node names of the used cluster and input300 is the input matrix.

# 2 Implemented algorithm

## 2.1 General description

The developed algorithm to solve the all-pairs shortest path problem was based in what is present in reference [1], namely the structure GRID_INFO_TYPE and the functions Setup_grid and Fox. All the remaining code was developed by the author, namely:

- The macros MIN (to determine the minimum value between two values) and matrix (to easily handle matrixes).
- The structure LOCAL_MATRIX_TYPE
- The functions read_files (to read the input files), write_files (to write the output files), Local_matrix_multiply (to locally multiply the matrixes), Set_to_zero (to restart the resulting matrix C), copy_matrix (to copy matrixes) and main (which includes the initialization of the program, allocation of memory, reading of the input matrixes, output of the resulting matrixes, messaging between processors, output and written to a text file of the execution time.

## 2.2 Main data structures

Two data structures have been used in the algorithm: GRID_INFO_TYPE and LOCAL_MATRIX TYPUE. The former was used to group the necessary data for the grid used to communicate between rows and columns of the grid communicator. It is composed of five integer variables (number of processors, p, the order of the grid, q, the row number and column number of the process on the grid and the rank number of the process on the grid) and three MPI_comm variables (communicator for the entire grid, communicator of the row and communicator of the column). The structure LOCAL_MATRIX_TYPE was used to store the matrixes, namely the integer pointer of the matrix and the dimension size of the matrix. This structure was used to handle all the matrixes in the program, both global and local matrixes.

## 2.3 Relevant auxiliary functions and type of communicators

The most relevant auxiliary functions of the program are the functions Setup_grid, Fox and read_files. The first is where the grid communicator is established using the MPI functions MPI_Cart_create, MPI_Cart_coords MPI_cart_sub. The first is used to create a new communicator identical to MPI_COMM_WORLD but with the processes organized in a cartesian grid. The second is used to obtain the coordinates of the current process in the grid. The MPI_cart_sub is used to obtain the row and column communicators of the grid.

In the function Fox is where the local matrixes are multiplied and changed between the processes according to the Fox algorithm. In order to communicate the local matrixes between processes the MPI functions MPI_Bcast and MPI_Send/MPI_Recv are used. The MPI_Bcast broadcast the matrix among all the processes in the shared communicator (for sending the local matrix A to all the processes of the same row of the grid) and the MPI_Send to send the local matrix B to the upper row in the same column of the grid.

The function read_files is also considered relevant essentially because of the code present in Source code 1. When the input matrix is read, the matrix is stored in memory in a reordered in order to ease the process of sending the local matrixes from the root to the remaining processors. Note that the input matrix is stored in order to have the local matrixes sequentially stored. At the same time, zeros outside the main diagonal are substituted by 999999, which works as infinite.

```
(…)
for (i=0; i<q; i++){
   for (j=0; j<cols; j++){
      for (k=0; k< n_bar; k++){
         pos = cols * cols/p*(j%q) + j/q*n_bar + k + i*n_bar*n_bar*q;
         fscanf(fp, "%d", (MM->Mat + pos));
         if ((cont%(cols+1)!=0) && *(MM->Mat + pos)==0) {
            *(MM->Mat + pos) = 999999;
         }
         cont++;
}}}
(…)
```

Source code 1 – Source code of function read_files (excerpt)

## 2.4 Performance evaluation

With regards to performance evaluation, 170 runs of the program were performed with a different number of processes for the available input matrixes of sizes 6, 300, 600, 900 and

1200. In Figure 1 and Table 1 the median execution time of each configuration and input matrix are presented. The first conclusion that can be drawn from these results is that paralleling speed up the computation time. With the exception of the smallest matrix of size 6, with more processes, lower computation time was obtained. This demonstrates that execution time is conditioned by the matrixes multiplications and not by the communication between the processes.

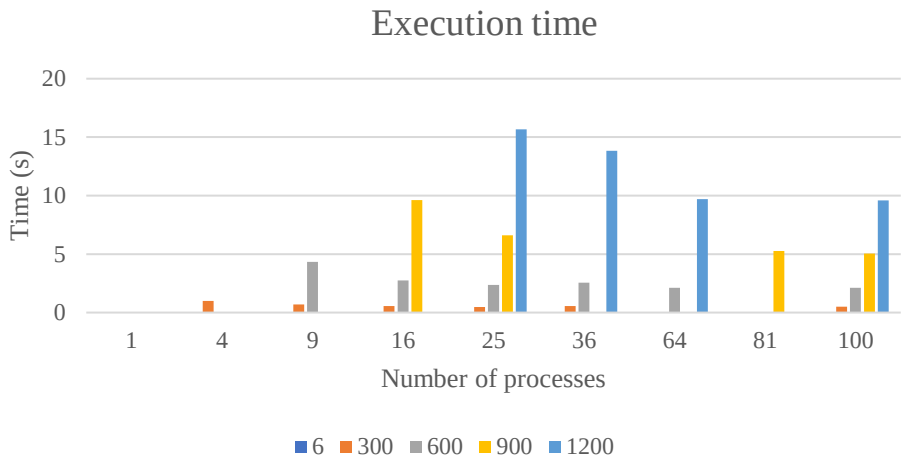


Figure 1 – Execution time (median, in seconds).

Table 1 – Execution time (median, in seconds).

| No. Processes | Matrix dimension | | | | |
|---|---|---|---|---|---|
| | 6 | 300 | 600 | 900 | 1200 |
| 1 | 0.0226 | | | | |
| 4 | 0.0281 | 1.0059 | | | |
| 9 | 0.0266 | 0.6883 | 4.3332 | | |
| 16 | | 0.5471 | 2.7593 | 9.6177 | |
| 25 | | 0.4792 | 2.3689 | 6.5937 | 15.6578 |
| 36 | | 0.5631 | 2.5625 | | 13.8467 |
| 64 | | | 2.1132 | | 9.7051 |
| 81 | | | | 5.2697 | |
| 100 | | 0.4971 | 2.1046 | 5.0592 | 9.5840 |

## 2.5 Main difficulties and challenges

Since the author does not have previous background in the programming language C, the greatest difficulty in an initial phase was to absorb all the particularities of the programming language C. Namely, the way how to handle and store matrixes was considerable difficult to understand in an initial phase and led to the author to spend an amount of time much larger

than would be expected. However, the author believes that those difficulties have been successfully overcome at the end of the work. Therefore, the author believes that this work was important not only to consolidate the knowledge about MPI but also about the programming language C.

In the second stage, the major difficulty was the communication of the different local matrixes from the initial input matrix. More than one solution was possible, but at the end, a reordering of the input matrix along with its reading and storage in memory was adopted.

# 3 Conclusions

The all-pairs shortest path problem proposed was, as intended, solved using parallel programming for distributed memory environments using the MPI framework. In the present report the main points of the developed algorithm were described. A performance evaluation was also presented for different sizes of the input matrix and number of processes. It was demonstrated that (with exception of the matrix with a very small size of 6), the execution time was improved when a bigger number of processes are used. Finally, the main difficulties of the author were described, namely the no previous knowledge about the C programming language. Indeed, these difficulties may have compromised the obtained results, since the obtained execution times are bigger than what the author was expecting and when the local matrix is equal or bigger than 300 the developed program stays computing without ending. Although all the efforts placed by the author, he was not able to detect the source of that problem in his code.

# 4 References

[1] Pacheco, P.S. (1998). *A User's Guide to MPI*, San Francisco, CA: University of San Francisco.