U. PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Parallel Computing

Project Assignment I

All-Pairs Shortest Path Problem

Paulo Araújo - 201805060
Rodrigo Salles - 201908577

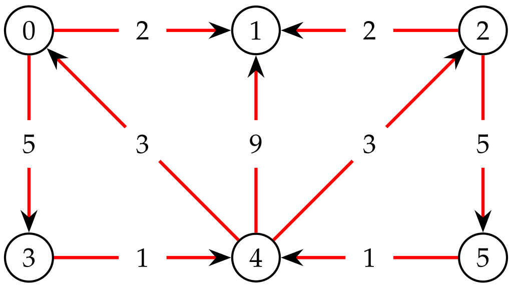
Porto, 10th December 2021

1 Introduction

Advances in science present us with increasingly complex problems. We must work with computationally demanding methods, and huge amounts of data. In an attempt to solve this dilemma, we can opt for a dedicated machine, with special characteristics, or conventional machines, which, in a distributed manner, can deal with the problem. For economic and practical reasons, the second option has become very popular. Thus, Parallel Computing techniques are needed. In this context, we have the first practical project of the discipline of parallel computing. The problem to be solved, known as **All-pairs shortest Path Problem**, will allow greater sensitivity in solving real problems, as well as greater experience in using tools dedicated to parallel computing, in this case, parallel programming for distributed memory environments using the MPI framework .

2 Problem description

The problem to be solved, better known as all-pairs shortest path problem, consists in determining all shortest paths between pairs of nodes in a given graph.



Given a directed graph $G = (V, E)$ in which V is the set of nodes, or vertices, and E is the set of links between nodes. The goal is to determine for each pair of nodes (v_i, v_j) the size of the shortest path that begins in v_i and ends in v_j. A path is simply a sequence of links in which the end node and start node of two consecutive links are the same.

A directed graph with V nodes can be represented as a matrix D_1 with dimension NxN (N = 6 in the example shown above) where each element (i, j) of the matrix represents the size of the link that connects node vi with node v_j.

The shortest path between the links, that is, the objective of the work, can be seen in table 2.

	0	**1**	**2**	**3**	**4**	**5**
0	0	2	0	5	0	0
1	0	0	0	0	0	0
2	0	2	0	0	0	5
3	0	0	0	0	1	0
4	3	9	3	0	0	0
5	0	0	0	0	1	0

Table 1: Matrix D1 that represents the size of the link that connects nodes

	0	**1**	**2**	**3**	**4**	**5**
0	0	2	9	5	6	14
1	0	0	0	0	0	0
2	9	2	0	14	6	5
3	4	6	4	0	1	9
4	3	5	3	8	0	8
5	4	6	4	9	1	0

Table 2: Matrix that corresponds to the size of the shortest path

There are several techniques for calculating the shortest path between pairs of links. For this work, the Fox method was adopted, which allows the operation with matrices in parallel, allowing the calculation of minimum distances to be parallelized.

3 Fox's Algorithm

Fox's algorithm is a parallel matrix multiplication function, which distributes the matrix using a checkerboard scheme. This means that the processes are viewed as a grid, and, rather than assigning entire rows or entire columns to each process, we assign small sub-matrices.

Given a NxN matrix and P processes, and assuming P is a perfect square, the algorithm starts by assigning each process a position in a matrix of order $\sqrt(P)$. Then each process is assigned a sub-matrix of order $N = \sqrt(P)$ to start the computation with. Given $N = 4$ and $P = 4$, then $Q = 2$. The matrix distribution would look like:

$$\left[\begin{array}{cc|cc} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ \hline A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{array}\right]$$

Where each quadrant represents the sub-matrix for one of the four processes. Three properties must be understood in order to comprehend Fox's Algorithm:

- The computation takes Q steps, as it iterates rows;
- Each process has three matrices in memory.
- Each process needs only to communicate with the processes in the same row or column in the process grid.

As for the three matrices:

- Matrix A is the starting matrix and the matrix which is broadcast to other processes in the same row;
- Matrix B starts as a copy of A and is updated at each stage, receiving B from the process directly below in the grid, wrapping around the matrix if needed;
- Matrix C holds the successive results from multiplication.

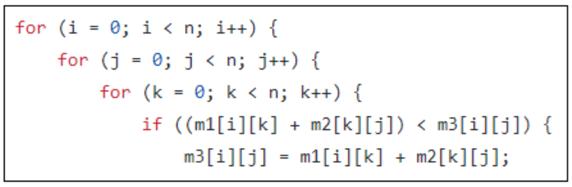
Each stage in the computation iterates on which process in each row will be the broad- caster. Then each process multiplies A of the broadcaster process with its local B, saving the result in C. At the end of the stage each process sends B to the process directly above it in the grid. Analogously each process also receives B from the process directly below.

This computation has to be repeated $\sqrt(N)$ times, as that's enough steps to reach the shortest distances for all pairs. At the start of each computation B is initialized with the result from the last computation, saved in C.

4 Fox Algorithm Implementation

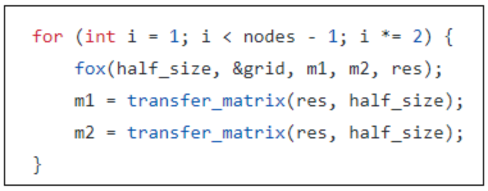
The algorithm was implemented according to the model presented in the Peter Pacheco's book *A User's Guide to MPI*[2][1]. The grid's order is calculated from the square root of the number of processes being used.

```
typedef struct {
    int p;             /* Total number of processes */
    MPI_Comm comm;     /* Communicator for entire grid */
    MPI_Comm row_comm; /* Communicator for my row */
    MPI_Comm col_comm; /* Communicator for my col */
    int q;             /* Order of grid */
    int my_row;        /* My row number */
    int my_col;        /* My column number */
    int my_rank;       /* My rank in the grid communicator */
} GRID_INFO_TYPE;
```



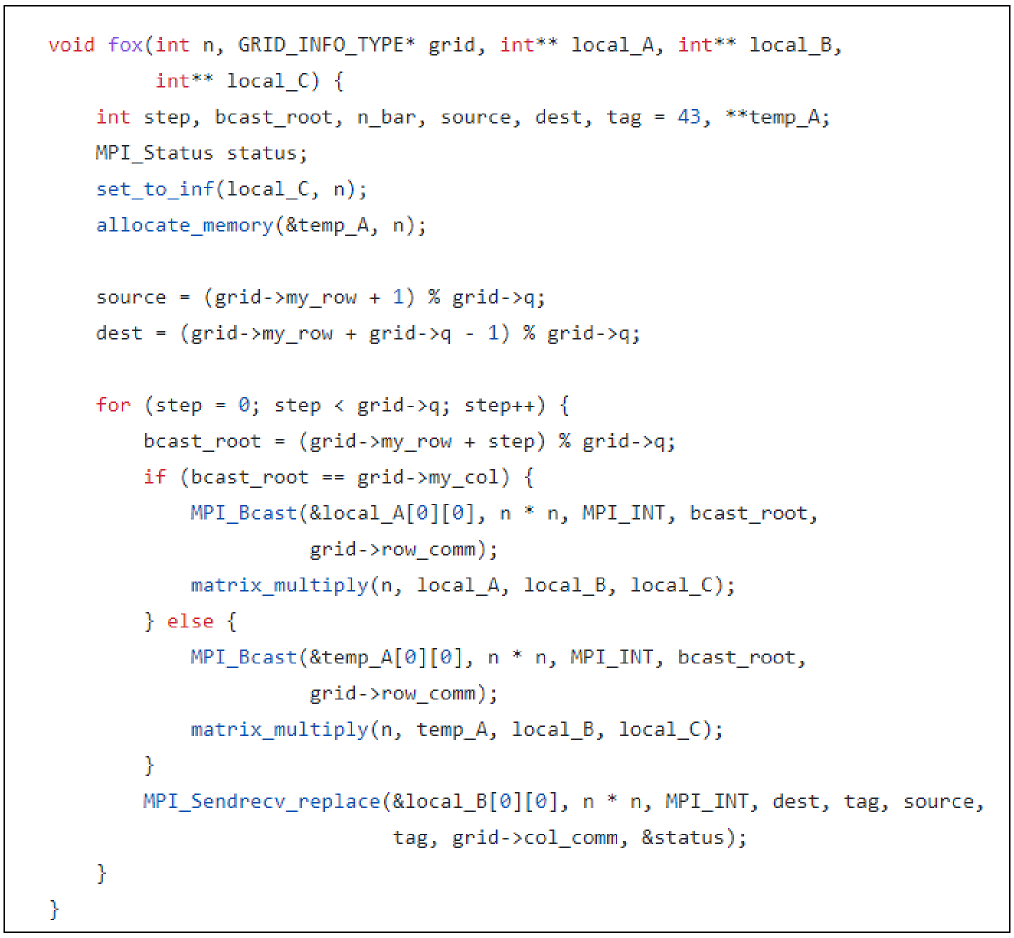
```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            if ((m1[i][k] + m2[k][j]) < m3[i][j]) {
                m3[i][j] = m1[i][k] + m2[k][j];
```

Figure 1: For every pair of nodes $(i; j)$, test if there is a shorter path.



```
for (int i = 1; i < nodes - 1; i *= 2) {
    fox(half_size, &grid, m1, m2, res);
    m1 = transfer_matrix(res, half_size);
    m2 = transfer_matrix(res, half_size);
}
```

Figure 2: The developed algorithm with Repeated Squaring algorithm calculates the shortest distance between nodes through successive iterations.

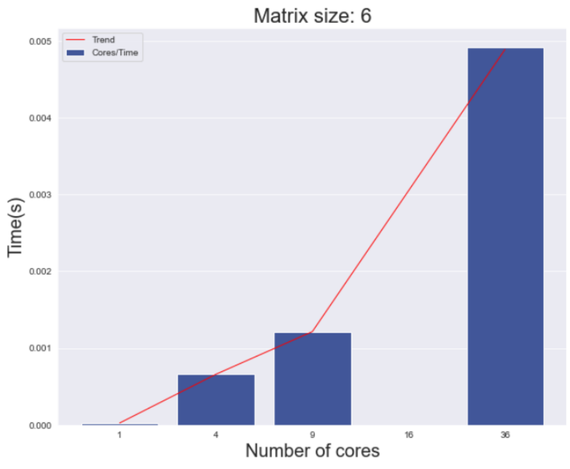


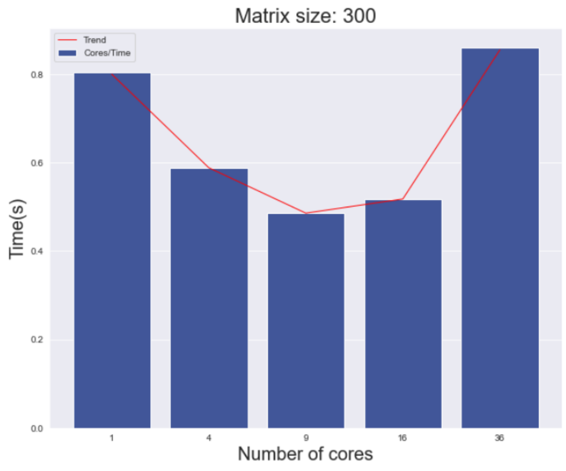
```
void fox(int n, GRID_INFO_TYPE* grid, int** local_A, int** local_B,
         int** local_C) {
    int step, bcast_root, n_bar, source, dest, tag = 43, **temp_A;
    MPI_Status status;
    set_to_inf(local_C, n);
    allocate_memory(&temp_A, n);

    source = (grid->my_row + 1) % grid->q;
    dest = (grid->my_row + grid->q - 1) % grid->q;

    for (step = 0; step < grid->q; step++) {
        bcast_root = (grid->my_row + step) % grid->q;
        if (bcast_root == grid->my_col) {
            MPI_Bcast(&local_A[0][0], n * n, MPI_INT, bcast_root,
                      grid->row_comm);
            matrix_multiply(n, local_A, local_B, local_C);
        } else {
            MPI_Bcast(&temp_A[0][0], n * n, MPI_INT, bcast_root,
                      grid->row_comm);
            matrix_multiply(n, temp_A, local_B, local_C);
        }
        MPI_Sendrecv_replace(&local_B[0][0], n * n, MPI_INT, dest, tag, source,
                             tag, grid->col_comm, &status);
    }
}
```

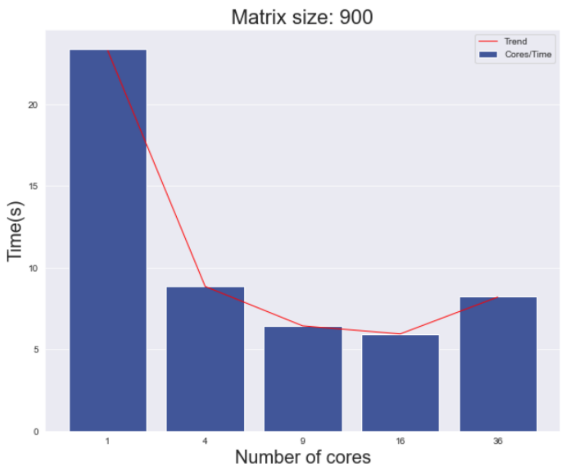
Figure 3: Implementation of the fox algorithm

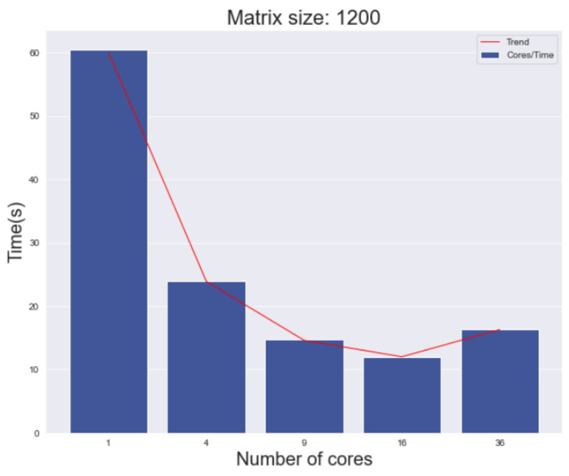
5 Performance evaluation

To evaluate the performance gain in the execution of parallel tasks, the execution of the shortest path calculation, described above, was analyzed in several situations, with variations in the size of the input matrices, and in the number of cores used. The results can be seen below:









For smaller arrays the process is fast, and as more processors are added the performance gets worse, with increasing time. This is explained by the need for communication between processes. For larger arrays the performance improves, to some extent, with the addition of processors. After a certain value, the performance worsens, which can be explained, again, by communication problems between the processes.

6 Difficulties encountered in the Project Assignment I

Some difficulties were encountered in the execution of the project. The first one was related to how we were managing the input. We assumed the distance between a node and itself was 0, but we also assumed that when there wasn't any path between two nodes the distance was also 0, leading to a wrong result. By defining the distance to "infinity" when there was no path, the problem was solved.

Another problem we encountered was with sending the right matrix to each process. The first approach was to use MPI Scatter. The problem with MPI Scatter is that it would send the data to every process in a wrong way, so we created a new MPI Datatype to define how a block was. Now, the MPI Scatter would send the data right but only to the first process. Latter, we found out that the solution was to use MPI Send instead of MPI Scatter. By defining an offset, we could send the right block to the right process and proceed with the computation. This method would, latter on, be useful to receive the results from each process.

With the use of MPI Send, we faced another problem. With small inputs (ie. Matrixes 6x6) the program worked just fine but, due to the fact that MPI Send blocks the execution of the program while the message is not all sent, with larger inputs (ie. Matrixes 300x300 or more), the program blocked. To resolve this problem, we needed an asynchronous function. We changed the MPI Send to MPI Isend and the problem was now solved.

7 Conclusions

The project allowed a better understanding of the relationship between the number of processors and performance. Adding processors does not always improve performance when performing tasks. The size of the task, characteristics of the machines and forms of communication must be considered. In many situations, a task executed sequentially is faster because it is not necessary to exchange messages between processors. For larger problems, where parallelization represents an advantage, the number of processors used must be considered, and the number from which communication starts to be a problem should be considered, and what is the ideal value.

References

[1] Peter Pacheco and Matthew Malensek. *An introduction to parallel programming*. Morgan Kaufmann, 2021.

[2] Peter S Pacheco et al. A user's guide to mpi. *University of San Francisco*, 56, 1998.