Parallel Computing: All-Pairs Shortest Path Problem

Hugo Valdrez
Computer Science Master's Degree
Universidade do Porto
up201704962@up.pt

Luís Pinto
Computer Science Master's Degree
Universidade do Porto
up201704025@up.pt

***Index Terms*—C Language, MPI**

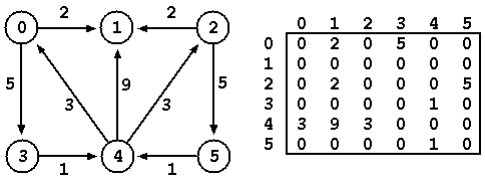
I. Introduction

THIS project offers a parallel computing solution to tackle the All-Pairs Shortest Path Problem within distributed memory environments utilizing the MPI framework. In our endeavor to solve this problem we have addressed the associated challenges, providing insights into our choices and the results we were able to achieve.

II. Understanding the problem

A. All pairs shortest path

The **All-Pairs Shortest Path Problem** is a classic problem in graph theory and computer science. It involves finding the shortest path between all pairs of nodes (or vertices) in a weighted graph. In other words, for a given graph with weighted edges, the goal is to determine the shortest distance between every pair of nodes in the graph.



	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

Fig. 1. Graph and link sizes between each node represented as a matrix (Adjacency Matrix)

B. Fox algorithm

The **Fox algorithm** is a parallel matrix multiplication algorithm often used in high-performance computing and distributed memory environments. The Fox algorithm is designed to efficiently multiply two large matrices by distributing the work across multiple processors or nodes in a parallel computing system.

Our interpretation and visualization of the process can be understood by thinking of the processors and their communication pattern as forming a torus [1].

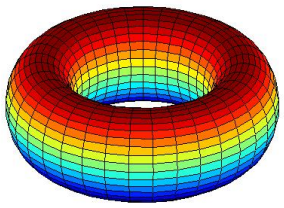


Fig. 2. Torus

The intuition behind it goes as it follows:

1. In the Fox algorithm, processors are arranged in a 2D grid. The grid is typically square, meaning it has the same number of rows and columns. Each processor corresponds to a cell in this grid.

2. The input matrices to be multiplied are divided into smaller blocks, and these blocks are distributed across the processors in the grid. Each processor is responsible for computing a portion of the resulting product matrix.

3. The key idea behind the torus topology is how processors communicate with their neighbors. Each processor communicates with its adjacent neighbors (above, below, left, and right) in a cyclic manner. This cyclic communication is akin to how a torus surface wraps around, creating a loop.

C. MPI

MPI: MPI is a widely-adopted, standardized, and portable message-passing system meticulously designed for the realm of parallel and distributed computing. It serves as the backbone for seamless communication between processes—individual programs or components—within a parallel or distributed computing environment, facilitating the exchange of messages to collaborate effectively.

MPI and MPI_Cart_create with the Torus Interpretation: A good synergy emerges when we harness the torus interpretation, which structures the parallel environment akin to a grid, in tandem with MPI's functionality, particularly exemplified by the MPI_Cart_create function. This coupling offers a good solution for orchestrating parallel computations that entail grid-like arrangements of processors. Notably, it accomplishes two pivotal objectives: simplifying the intricate organization of processes and streamlining the intricate web of communication and coordination within the grid.

D. Min-plus

Min-plus matrix multiplication, also known as Distance Product, is an adapted form of matrix multiplication in which the multiplication and sum operations are replaced by operations of sum and minimum, respectively. Starting with A, B as two matrices of size $n \times n$, in order to obtain $C = A \star B$ we proceed as it follows:

1. Initialize C with $C_{i,j} = \infty$.

2. Repeatedly update a matrix C using min-plus multiplication:

$$C_{i,j} = \min(C_{i,j}, A_{i,k} + A_{k,j}) \text{ for all } k \in \{1, ..., n\}$$

Applying *Min-plus* to the **All-Pairs Shortest Path Problem**. Assuming A is our adjacency matrix of size n:

1. Initialize C with

$$C_{i,j} = \begin{cases} A_{i,j}, \text{ if } A_{j,j} \neq 0 \\ \infty, \text{ if } A_{j,j} = 0 \end{cases}$$

2. Repeat for $\lfloor \log n \rfloor$ times $C = C \star C$

Resulting in the shortest paths between all nodes of A

III. Implementation

In this section, we delve into the practical implementation of the parallel computing solution for the All-Pairs Shortest Path Problem. The code's functionality encompasses matrix initialization, MPI initialization, reading the input matrix, validation of configurations, matrix block setup, and the application of the Fox algorithm for distributed matrix multiplication.

1. The code initializes the Matrix for all processes. It allocates memory for the Matrix in non-master processes and subsequently broadcasts the initial matrix size and the entire Matrix to all processes.

```
    if (rank == 0) {
        scanf("%d", &matrixDim);
        alloc_contiguous(&Matrix, matrixDim);
        read_matrix(matrixDim, Matrix);
    }
    MPI_Bcast(&matrixDim, 1, MPI_INT, 0,
    MPI_COMM_WORLD);
    ...
    if (rank != 0) {alloc_contiguous(&Matrix,
    matrixDim);}
    MPI_Bcast(Matrix, matrixDim*matrixDim, MPI_INT,
    0, MPI_COMM_WORLD);
```

2. It calculates the number of blocks (numblocks) and the size of each block (blocksize) based on the matrix size and the number of processes.

```
    numblocks = sqrt(numprocs);
    blocksize = matrixDim / numblocks;
```

Exploring the key aspects of our code, it proceeds as follows:

3. The code initializes the matrices, including Matrix_A, Matrix_B, Matrix_C, and Matrix_aux, which will be used for matrix multiplication and storage of intermediate results.

```
    Matrix_A = (int *)malloc(sizeof(int) * blocksize
     * blocksize);
    Matrix_B = (int *)malloc(sizeof(int) * blocksize
     * blocksize);
    Matrix_C = (int *)malloc(sizeof(int) * blocksize
     * blocksize);
    Matrix_aux = (int *)malloc(sizeof(int) *
    blocksize * blocksize);
```

4. The code creates a Cartesian grid communicator (gridComm) based on the 2D topology with dimensions specified by sizes. This grid is used for organizing processors into a grid structure.

```
    int  sizes[dim_cart_comm] = {numblocks,
    numblocks};
    int   wrap[dim_cart_comm] = {1, 1};
    int colDir[dim_cart_comm] = {1, 0};
    int rowDir[dim_cart_comm] = {0, 1};
    int optm = 1, gridRank, gridCoords[dim_cart_comm
    ], row, col;
    ...
    MPI_Cart_create(MPI_COMM_WORLD, dim_cart_comm,
    sizes, wrap, optm, &gridComm);
    MPI_Comm_rank(gridComm, &gridRank);
    MPI_Cart_coords(gridComm, gridRank,
    dim_cart_comm, gridCoords);
    MPI_Cart_sub(gridComm, colDir, &colComm);
    MPI_Cart_sub(gridComm, rowDir, &rowComm);
```

5. The code iteratively calculates the shortest paths between all pairs of nodes by applying the Fox algorithm. At each step, it multiplies matrices Matrix_A and Matrix_B to update Matrix_C, gradually refining the shortest path distances.

```
    for (int step=1; step<matrixDim-1; step*=2) {
        fox_alg(numblocks, blocksize, Matrix_A,
    Matrix_B, Matrix_C, Matrix_aux);
        copy_matrix(Matrix_C, Matrix_A, blocksize);
        copy_matrix(Matrix_C, Matrix_B, blocksize);
    }
    ...
void fox_alg(int numblocks, int blocksize, int*
    Matrix_A, int* Matrix_B, int* Matrix_C, int*
    Matrix_aux) {
    int i, j, root;
    for (i = 0; i < numblocks; i++) {
        root = ( myRow  + i ) % numblocks;
        if(root == myCol){
            // snd my matrix A to neighbour
            MPI_Bcast(Matrix_A, blocksquare, MPI_INT
    , root, rowComm);
            min_plus_matrix(Matrix_A, Matrix_B,
    Matrix_C, blocksize);
        } else {
            // rcv an A from neighbour
            MPI_Bcast(Matrix_aux, blocksquare,
    MPI_INT, root, rowComm);
            min_plus_matrix(Matrix_aux, Matrix_B,
    Matrix_C, blocksize);
        }
        // rotate B's
        MPI_Sendrecv_replace(Matrix_B, blocksquare,
    MPI_INT, dst, tag, src, tag, colComm, &status);
    }
}
```

6. The code gathers the local Matrix_C results from all processes into the MatrixResult. Afterward, it reorders the elements in MatrixResult to match the desired matrix order, since it was written in a contiguous way. The final result represents the shortest path distances between all pairs

of nodes and is made consistent for further analysis and presentation.

```
MPI_Gather(Matrix_C, blocksquare, MPI_INT,
MatrixResult, blocksquare, MPI_INT, 0,
MPI_COMM_WORLD);
if (rank==0) {
    int *tmp = (int *)malloc(matrixDim*matrixDim
 * sizeof(int));
    int next_elem = 0;
    for (int n=0; n<numblocks*numblocks; n++) {
        for (int i=0; i<blocksize; i++) {
            for (int j=0; j<blocksize; j++) {
                row = (int) n / numblocks;
                col = n % numblocks;
                tmp[i*matrixDim + j + col*
blocksize + row*blocksize*matrixDim] =
MatrixResult[next_elem]<INF ? MatrixResult[
next_elem] : 0;
                next_elem++;
            }
        }
    }
    for (int i=0; i<matrixDim*matrixDim; i++) {
        MatrixResult[i] = tmp[i];
    }
}
```

IV. Performance

To assess the efficacy of our implementation of the **Fox Algorithm** applied to the **All-Pairs Shortest Path Problem**, we conducted a series of tests on matrices of varying dimensions. The execution times (in seconds) were measured using a different numbers of processes. The results are presented in the table below:

Dim \ Processes	1	4	9
Input6	0.00026	0.00037	0.00068
Input300	1.05224	0.27398	0.22733
Input600	9.48172	2.48093	2.00836
Input900	31.73073	8.38730	6.84998
Input1200	82.78222	21.96372	17.61832
	16	**25**	**36**
Input6	NA	NA	1.01104
Input300	1.02740	1.88055	3.00533
Input600	2.90841	3.55514	4.28495
Input900	7.93949	8.14382	7.80679
Input1200	18.04232	16.76800	16.30964

As anticipated, there exists an inverse relationship between matrix dimensions and execution time. Notably, smaller matrices show limited improvement with an increased number of processes, suggesting an optimal number of processes for a given matrix dimension.

We hypothesize that the potential gains from distributing the workload among multiple processes are counteracted by the increased communication overhead required to synchronize the higher number of processes.

This observation underscores the importance of considering both matrix size and parallelization strategies for optimal performance in solving the **All-Pairs Shortest Path Problem** using the **Fox Algorithm**. Further investigation into the intricate balance between computation and communication is warranted to refine our understanding of these observed patterns.

V. Final Comments

During the implementation of our algorithm, we encountered several challenges that required careful consideration and analysis. Our initial approach involved building upon an existing implementation of the **Fox Algorithm**, but we faced obstacles along the way. Notable issues included:

- **Communicator Initialization:** We grappled with determining the optimal location for initializing communicators. Originally, we defined communicators within our "fox_algorithm" function, resulting in sub-optimal performance. Through thorough analysis, we realized that initializing communicators once before the execution of the `fox_algorithm` function significantly improved efficiency.
- **Data Sharing Strategy:** Initially, we implemented data sharing in each iteration of the Fox Algorithm, distributing results across all processes. However, we discovered that gathering data needs to occur only once at the conclusion of all Fox Algorithm executions, leading to a more streamlined and efficient implementation.

Additional challenges arose when dealing with broadcasting and sending/receiving data. To address these challenges, we opted for matrices with contiguous memory allocation instead of using a pointer array. This decision, while not extensively exploring the creation of MPI data types, facilitated rapid execution and mitigated data transmission issues.

A noteworthy problem emerged post-gathering of results. As each process was enumerated from 1 to n, the gathering process wrote content to incorrect positions. Overcoming this challenge required matrix rearrangement post-gathering, a solution that proved effective in ensuring accurate data placement.

In summary, our iterative problem-solving approach allowed us to address various issues, leading to an optimized MPI implementation of the Fox Algorithm for the All-Pairs Shortest Path Problem. These experiences underscore the importance of thoughtful design decisions and iterative refinement in achieving efficient parallel algorithms.

References

[1] Lecture from UNE - http://turing.une.edu.au/ cosc330/lectures/lecture_17/