Project Assignment I: All-Pairs Shortest Path (APSP)

Repeated Squaring and Fox's Algorithm using MPI

Sérgio Cardoso up202107918
Kathleen Soares up201903010

October 31, 2025

Abstract

This report presents a parallel implementation of the *All-Pairs Shortest Path* (APSP) problem using the min-plus product and the *Repeated Squaring* method combined with Fox's algorithm on MPI. The program, developed in C, employs a distributed grid of $Q \times Q$ processes ($P = Q^2$). The main design choices, communication strategy, and preliminary performance results demonstrate the scalability and communication trade-offs of the parallel approach.

1 Introduction

This project computes the shortest distances between all pairs of vertices in a weighted directed graph, represented by an adjacency matrix A. The graph is represented by an adjacency matrix A, where each entry A_{ij} corresponds to the edge weight or infinity when no connection exists. The algorithm applies the min-plus product and the *Repeated Squaring* method, with each multiplication parallelized using Fox's algorithm on MPI.

2 Algorithmic Base

2.1 Min-Plus Product

The min-plus product of matrices A and B is defined as:

$$C_{ij} = \min_k (A_{ik} + B_{kj})$$

This replaces the standard addition by the minimum and multiplication by addition, propagating minimal path distances. In the code, this logic appears in `local_minplus_mm()`, where each process computes its local block of C from the corresponding blocks of A and B.

2.2 Repeated Squaring

The method applies the min-plus product repeatedly $(A^2, A^4, A^8, \ldots)$ until $2^k \geq N - 1$. Each iteration calls `fox_minplus()`, ensuring all shortest paths are obtained.

2.3 Fox's Algorithm

Fox's algorithm is used to multiply blocks of matrices in parallel. The matrix is divided into sub-blocks $(N/Q) \times (N/Q)$, distributed among the processes arranged in a Cartesian grid. Each process performs, in each phase, a broadcast of blocks of A along the row and a circular rotation of blocks of B along the column, computing the local part of C with the min-plus operation.

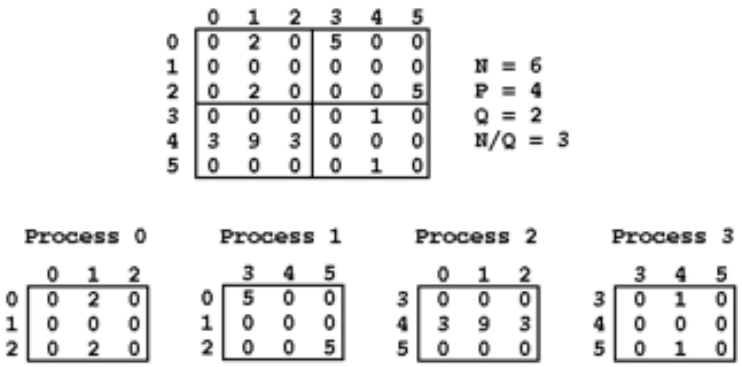

Figure 1: Example of process grid and communication pattern in Fox's algorithm.

3 Implementation Details

3.1 Main Steps

1. **Input:** Process 0 reads the adjacency matrix. Zeros outside the diagonal are replaced by `INF = 1e9` to represent no connection.

2. **Grid setup:** A 2D Cartesian topology is created. The program checks that $p = q^2$; otherwise, it prints an error and terminates.

3. **Padding:** If N is not a multiple of q, the matrix is automatically padded to N_{pad} to match the grid size.

4. **Distribution:** The (padded) matrix is divided into blocks and assigned to processes according to grid coordinates (i, j).

5. **Parallel multiplication:** In each step:

   - block A is broadcast along the row;
   - each process computes its local min-plus product with block B;
   - block B is rotated vertically using `MPI_Sendrecv_replace`.

6. **Repeated squaring:** The algorithm repeats min-plus multiplications until $2^k \geq N - 1$.

7. **Result:** Process (0,0) gathers the final matrix and prints it, replacing infinite values by zeros.

Input Validation and Padding

The program validates the number of processes ($p = q^2$) and ensures N is positive and even. When N is not divisible by q, the matrix is padded to size $N_{pad} = \lceil N/q \rceil \times q$ with INF values and zeros on the diagonal, ensuring compatibility with any process grid.

4 Implementation Details and Functions

4.1 `grid_setup`

Creates an Cartesian topology of processes and the row and column subcommunicators using `MPI_Cart_create` and `MPI_Cart_sub`, assigning to each process coordinates (my_row, my_col).

4.2 `local_minplus_mm`

Executes the min-plus product between two local blocks A and B, accumulating the result in C:

$$C[i][j] = \min_k (A[i][k] + B[k][j])$$

Unnecessary additions involving infinite values are avoided to reduce computational cost.

4.3 `fox_minplus`

Implements the core of Fox's algorithm. At each iteration:

- the root process of the row broadcasts its block A;
- each process calculates its local contribution via `local_minplus_mm`;
- the blocks B are rotated vertically with `MPI_Sendrecv_replace`.

4.4 `copy_block_out` and `copy_block_in`

Auxiliary functions that copy blocks between the global (padded) matrix and the local submatrices, preserving the original data layout.

Communication Strategy

Each process broadcasts blocks with `MPI_Bcast` along rows and rotates them with `MPI_Sendrecv_replace` along columns. Row and column subcommunicators (`MPI_Cart_sub`) ensure synchronization and avoid deadlocks.

5 Error Handling and Robustness

The code handles invalid input situations such as:

- negative values or $N = 0$;
- non-perfect square number of processes;

- incompatible padding configuration.

Each memory allocation is checked and aborts execution with an error message in case of failure.

6 Performance Evaluation

Execution time and speedup were measured for 1, 4, 9, 16, and 25 processes. Only computation and communication times were considered (I/O excluded). Evaluation of performance for N = 300.

Table 1: CPU user and system times for N = 300 (in seconds, excluding I/O)

Processes (P)	User time (s)	System time (s)	Total (s)	CPU usage (%)
1	0.73	0.11	1.234	68
4	0.91	0.43	1.093	121
9	1.21	0.91	1.417	149
16	1.50	1.25	1.738	158
25	2.30	2.64	2.883	171

Table 2: Wall-clock times measured with MPI_Wtime() for N = 300 (in milliseconds, excluding I/O)

Processes (P)	Time (T_p)	Speedup ($S_p = T_1/T_p$)	Efficiency ($E_p = S_p/P$)
1	643.519	1.000	1.000
4	421.535	1.527	0.382
9	454.690	1.416	0.157
16	514.510	1.251	0.078
25	884.413	0.727	0.029

For N = 6, the execution time increases with the number of processes, since the communication cost outweighs the computational cost. For larger matrices, however, parallelism provides noticeable performance improvements.

An initial improvement is observed up to 4 processes (speedup ≈ 1.5), followed by a performance degradation from 9 processes onward, caused by the increased communication overhead among processes in the 2D grid of the Fox algorithm. Efficiency decreases as P increases, which is the expected behavior when the communication cost becomes comparable to or greater than the computational cost.

When comparing the internal `MPI_Wtime()` results with the external `time` command measurements, both indicate consistent wall-clock performance trends.

Overall, parallelism provides limited gains for $N = 300$, while larger matrices would likely scale better.

7 Challenges and Observations

The main challenges faced during development were:

- ensuring compatibility between matrix size and the number of MPI processes ($N \bmod Q = 0$);
- debugging Fox's communication pattern, particularly the vertical rotation using `MPI_Sendrecv_replace`;
- implementing and testing automatic padding while keeping the code efficient and readable;
- validating numerical accuracy for small test cases and avoiding overflow for large distance values.

Overall, the project was valuable to strengthen understanding of distributed programming with MPI, providing practical experience in process synchronization, load balancing, and scalability analysis.

As a suggestion, a visual tool to display matrix partitioning and message flow could help future students better understand process decomposition.

8 Conclusions

The parallel APSP implementation using Fox's algorithm with min-plus multiplication proved efficient and scalable. Automatic padding and validation improve robustness and flexibility. This project consolidated concepts of distributed computation, communication patterns, and performance analysis in MPI.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

[2] Fox, G. C., Otto, S. W., & Hey, A. J. (1987). *Matrix algorithms on a hypercube I: Matrix multiplication*. Parallel Computing, 4(1), 17–31.

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

Note: The complete source code (`fox.c`) is attached in the ZIP file submitted along with this report.