Project done by: **Diogo Alves**, 202006033 and **Mário Minhava**, 202206190

Project Overview

This project implements a parallel solution for **All-Pairs Shortest Path computation** using the **Fox Algorithm for matrix multiplication**. The implementation leverages the **MPI (Message Passing Interface)** library to distribute computation across multiple processes in a 2D process grid topology.

Algorithm Implementation

All-Pairs Shortest Path Problem

The algorithm implemented solves the All-Pairs Shortest Path problem, which consists of finding the shortest path from each vertex to every other vertex in a directed graph. The solution approach involves:

1. **Graph Representation**: The graph is represented as an N×N matrix (where N is the number of vertices)
   - Each position (row, column) represents the cost of traveling directly from vertex `row` to vertex `column`
   - If no direct path exists between two nodes, the position is set to ∞
   - The solution is an N×N matrix where each position (row, column) corresponds to the cost of the shortest path between vertices `row` and `column`

Distance Product Matrix Multiplication

This algorithm adapts normal matrix multiplication for shortest path computation by: - **Replacing multiplication with addition**: Instead of `a[i][k] * b[k][j]`, we use `a[i][k] + b[k][j]` - **Replacing addition with minimum**: Instead of summing products, we take the minimum of all sums

Given matrices `D^f` and `D^k` (where f and k represent the maximum path depth), the algorithm produces `D^(f+k)`, combining paths from both input matrices. Path depth refers to the number of intermediate nodes traversed from origin to destination.

To solve the All-Pairs Shortest Path problem, we need matrix `D^N` that considers all possible paths. Since paths longer than N cannot be shorter than existing paths, `D^N` contains the optimal solution.

Naive Approach: Multiply `D^1` by itself N times - **Complexity**: O(N^4) - N matrix multiplications, each O(N^3)

Repeated Squaring Optimization

We exploit the mathematical property that `D^f * D^k = D^(f+k)` to reduce complexity:

1. Start with weight matrix `D^1`
2. Repeatedly square: `D^k * D^k = D^(2k)`
3. Continue until `2^k >= N`

Optimized Complexity: O($\sqrt{N}$) * O(N^3) = O(N^3.5), significantly better than O(N^4)

Fox's Algorithm Parallelization

Fox's algorithm parallelizes matrix multiplication across multiple processors:

Requirements: - Number of processes `p = q^2` (perfect square) - `q` must divide `N` evenly - Each process handles an `(N/q) × (N/q)` submatrix

Process Organization: - Processes arranged in a `q × q` 2D grid - Matrix divided into blocks distributed in checkerboard fashion - Process assignment: `f(p) = (p / q, p mod q)`

Algorithm Steps (for q iterations): 1. **Matrix Selection**: In each row, select submatrix `A[r,u]` where `u = (r + step) mod q` 2. **Row Broadcast**: Broadcast selected A submatrix to all processes in the same row 3. **Local Multiplication**: Each process multiplies received A submatrix by its local B submatrix 4. **Column Rotation**: Circularly shift B submatrices up within each column (top row sends to bottom row)

Project Structure

The project is modularized into separate components for clarity and reusability:

File	Purpose
`main.c`	Main program, MPI setup, and algorithm orchestration.
`fox.h` / `fox.c`	Structures and functions for the Fox Algorithm.
`matrix.h` / `matrix.c`	Matrix allocation, I/O, and manipulation.
`io.h`	Input/output handling
`Makefile`	Build system with compilation and automated tests

Architecture and Data Structures

Core Structures

`struct FoxDetails` The central configuration structure that contains all grid and matrix parameters:

- `Q`: Square root of total processes (grid dimension) - creates a Q×Q process grid
- `N`: Global matrix size (N×N input matrix)

- `per_process_n`: Local submatrix size per process (N/Q)×(N/Q)
- `myRow`, `myColumn`: Process coordinates in the 2D grid (0 to Q-1)
- `myRank`: Process rank within the Cartesian communicator
- `envData`: Contains MPI environment information (total processor count)

Validation Requirements: - Total processes must be a perfect square ($p = Q^2$) - Matrix size N must be divisible by Q - Each process handles exactly $(N/Q)^2$ matrix elements

`struct FoxMPI` Complete MPI communication infrastructure for the Fox algorithm:

- `fox_details`: Embedded FoxDetails structure with grid configuration
- `cart`: 2D Cartesian communicator with wraparound topology for the entire Q×Q grid
- `row`: Row-specific subcommunicator enabling broadcasts within process rows
- `col`: Column-specific subcommunicator for vertical data circulation
- `datatype`: Custom MPI datatype representing a contiguous (N/Q)×(N/Q) matrix block

Communication Topology: - **Cartesian Grid**: Processes arranged in a Q×Q torus with periodic boundaries - **Row Communicators**: Enable simultaneous broadcasting of A-blocks across each row - **Column Communicators**: Support circular shifting of B-blocks vertically

---

Process Grid Architecture

2D Cartesian Topology Setup The implementation creates a **Q×Q torus topology** with the following characteristics:

Grid Organization: - Process assignment: `f(p) = (p / Q, p mod Q)` maps linear rank to (row, col) coordinates - **Periodic boundaries**: Processes at edges wrap around (top connects to bottom, left to right) - **Coordinate system**: (0,0) at top-left, (Q-1,Q-1) at bottom-right

Local Data Management: Each process (i,j) manages: - **Local A-block**: Submatrix A[i][j] of size (N/Q)×(N/Q) - **Local B-block**: Submatrix B[i][j] of size (N/Q)×(N/Q)
- **Local C-block**: Result submatrix C[i][j] for accumulating partial products - **Communication buffers**: Temporary storage for received A and B blocks

Memory Layout: - Matrices stored as **contiguous 1D arrays** for efficient MPI communication - Custom MPI datatype represents entire (N/Q)×(N/Q) submatrix block - Matrix element indexing: `element[row][col] = array[row * (N/Q) + col]`

Communication Patterns **Row-wise Broadcasting** (A-block distribution): - In step k, each row r broadcasts A-block from column `(r + k) mod Q` - Uses row subcommunicators for simultaneous broadcasts across all rows - Ensures all processes in a row receive the same A-block for multiplication

Column-wise Circulation (B-block rotation): - B-blocks shift upward within each column after each multiplication - Circular topology: top process sends to bottom process - Maintains synchronized access to different B-blocks across iterations

---

Core Functions

1. Main MPI Setup (`main.c`)

- **MPI Initialization**: Sets up MPI environment and determines process grid dimensions
- **Data Distribution**: Handles matrix scattering from root to all processes
- **Result Collection**: Gathers computed results back to root process
- **Process Coordination**: Manages synchronization between computation phases

2. Fox Algorithm Implementation (`fox.c`)

- `performFoxAlgorithm()`: Orchestrates the Fox algorithm steps
- `performAllPairsShortestPath()`: Implements repeated squaring using Fox multiplication
- `divideMatrix()`: Partitions matrices into submatrices for distribution
- **Matrix Operations**: Local matrix multiplication and minimum path updates

3. Matrix Operations (`matrix.c`)

- **Matrix I/O**: Reading adjacency matrices from input files
- **Memory Management**: Allocation and deallocation of matrix structures
- **Utility Functions**: Matrix copying, initialization, and validation

4. MPI Communication Framework

To enable parallel computation across multiple processes, the implementation leverages **MPI (Message Passing Interface)**, a standardized framework for process communication in distributed computing environments.

MPI provides the foundation for scaling the Fox algorithm from single-machine execution to large-scale supercomputer deployments without requiring code modifications. The same program can seamlessly operate across multiple cores, nodes, or entire computing clusters.

4.1 Communicator Architecture MPI organizes processes into **communicators** - groups that define communication scope and topology. The implementation uses a hierarchical communicator structure:

Global Communicator (`MPI_COMM_WORLD`): - Contains all processes participating in the computation - Used for initial setup and final result gathering - Provides the foundation for creating specialized subcommunicators

Custom Communicators: - **Row Communicators**: Enable efficient broadcasting within each process row - **Column Communicators**: Support vertical circulation of matrix blocks - **Cartesian Communicator**: 2D grid topology with wraparound boundaries

This design optimizes communication efficiency by limiting message scope to relevant process subsets, rather than broadcasting to all processes globally.

4.2 Custom MPI Datatype The implementation defines a **custom MPI datatype** to represent matrix subblocks:

```
MPI_Type_contiguous(perProcessMatrixSize * perProcessMatrixSize,
                    MATRIX_ELEMENT_MPI, &datatype);
```

Benefits: - **Atomic Operations**: Entire (N/Q)×(N/Q) submatrices transmitted as single units - **Type Safety**: Ensures consistent data interpretation across processes - **Performance**: Reduces communication overhead compared to element-by-element transfers

4.3 Communication Patterns Matrix Distribution (`MPI_Scatter`): - Root process divides input matrix into subblocks - Each process receives its assigned (N/Q)×(N/Q) submatrix - Ensures load balancing across the process grid

Row-wise Broadcasting (`MPI_Bcast`): - A-blocks broadcast horizontally within each row - Uses row subcommunicators for simultaneous broadcasts - Critical for Fox algorithm's matrix multiplication phase

Column-wise Circulation (`MPI_Sendrecv_replace`): - B-blocks circulate vertically within each column - Simultaneous send to process below and receive from process above - Implements circular topology with wraparound (top to and from bottom)

Result Collection (`MPI_Gather`): - Each process contributes its computed C-submatrix - Root process assembles final result matrix - Preserves original matrix structure and ordering

4.4 MPI Functions Utilized The implementation employs the following MPI operations:

- `MPI_Cart_create`: Establishes 2D Cartesian topology with periodic boundaries

- `MPI_Cart_sub`: Creates row and column subcommunicators from Cartesian grid
- `MPI_Bcast`: Broadcasts A-blocks within process rows
- `MPI_Sendrecv_replace`: Implements circular B-block rotation in columns
- `MPI_Scatter`: Distributes input matrix subblocks to processes
- `MPI_Gather`: Collects result submatrices from all processes
- `MPI_Type_contiguous`: Defines custom datatype for matrix subblocks
- `MPI_Comm_rank` / `MPI_Comm_size`: Process identification and grid setup

Supported Configurations

Process Counts and Matrix Compatibility

The implementation supports various process counts with corresponding matrix size requirements:

Process Count (P)	Grid Layout	Compatible Matrix Sizes
1	1 x 1	All sizes
4	2 x 2	Multiples of 2
9	3 x 3	Multiples of 3
16	4 x 4	Multiples of 4
25	5 x 5	Multiples of 5

Test Matrix Configurations

- **input5**: 5x5 matrix (compatible with P=1 only)
- **input6**: 6x6 matrix (compatible with P=1,4,9)
- **input16**: 16x16 matrix (compatible with P=1,4,16)
- **input25**: 25x25 matrix (compatible with P=1,25)
- **input300**: 300x300 matrix (compatible with P=1,4,9,25)
- **input600**: 600x600 matrix (compatible with P=1,4,9,16,25)
- **input1200**: 1200x1200 matrix (compatible with P=1,4,9,16,25)

Build System

Makefile Targets

```
# Compilation
make                    # Build main executable
make clean              # Clean build artifacts

# Testing
make test-multi         # Run all 22 comprehensive tests
make test-p1            # Test single process execution
make test-p4            # Test 2x2 process grid
make test-p9            # Test 3x3 process grid
```

```
make test-p16          # Test 4x4 process grid
make test-p25          # Test 5x5 process grid
```

Compilation Flags

```
CFLAGS = -std=c11 -O2 -Wall -Wextra
MPICC = mpicc
```

Architecture Decisions

Error Handling

- **Input Validation**: Ensures matrix dimensions are compatible with process counts
- **MPI Error Checking**: Validates MPI operation success
- **Resource Cleanup**: Proper deallocation of matrices and MPI communicators

Scalability Considerations

- **Submatrix Size**: Each process handles (n/q)x(n/q) elements
- **Communication Overhead**: Minimized through efficient broadcasting and shifting patterns
- **Load Balance**: Equal distribution of work across all processes

Testing Infrastructure

Comprehensive Test Suite

The project includes 22 different test cases covering all valid matrix-process combinations:

```
# Example test execution
mpirun -np 9 ./main matrix_examples/input300  # 3x3 grid, 100x100 per process
mpirun -np 16 ./main matrix_examples/input1200 # 4x4 grid, 300x300 per process
```

Validation Process

1. **Correctness Verification**: Compare parallel results with sequential computation
2. **Process Count Testing**: Validate all supported process configurations
3. **Matrix Size Testing**: Test various matrix dimensions for compatibility
4. **Performance Monitoring**: Track execution times across different configurations

Performance Benchmarks

Expected Results:

- Performance analysis will be conducted on cluster infrastructure
- Comparison of sequential vs. parallel execution times
- Identification of optimal process counts for different matrix sizes
- Analysis of communication vs. computation trade-offs

Compilation Requirements

```
# Required system components
mpicc (GCC-based MPI compiler)
OpenMPI or MPICH implementation
```

Input Format

```
n
a11 a12 ... a1n
a21 a22 ... a2n
...
an1 an2 ... ann
```

Performance Analysis

Overview

The experimental results show the execution times for different matrix sizes (`N`) and process counts (`P`) when running the parallel All-Pairs Shortest Path (APSP) implementation based on the Fox Algorithm.

The goal of these measurements is to evaluate how well the algorithm scales with an increasing number of processes.

Processes	Matrix Size	Time (s)
1	6	0.341
	300	0.573
	600	2.129
	900	6.342
	1200	15.391
4	6	0.327
	300	0.430
	600	0.879
	900	2.148
	1200	4.922
9	6	0.384
	300	0.457
	600	0.818
	900	1.881
	1200	4.020
16	300	0.531

Processes	Matrix Size	Time (s)
	600	0.924
	900	2.037
	1200	4.138
25	300	0.591
	600	0.962
	900	1.948
	1200	4.260

Observed Trends

1. **Clear parallel speedup for large matrices**
   - As the matrix size increases, parallelization provides a **significant reduction in execution time**.
   - For example, with `N=1200`, runtime drops from **15.39s (P=1)** to ~**4.02s (P=9)** – roughly a **3.8× speedup**.
2. **Diminishing returns for small matrices**
   - For small sizes (`N=6`, `N=300`), the execution time does not improve significantly with more processes, and in some cases even increases.
   - This happens because **communication costs outweigh computation time**.
     For example:
     - `N=300`: 0.430s (P=4) → 0.531s (P=16) → 0.591s (P=25)
     - Here, more processes lead to **higher MPI overhead** with little computational gain.
3. **Non-linear scalability**
   - Between `P=9` and `P=16`, we see **non-monotonic behavior**:
     - `N=600`: 0.818s (P=9) vs. 0.924s (P=16)
     - `N=900`: 1.881s (P=9) vs. 2.037s (P=16)
   - This is a typical sign of **communication bottlenecks** caused by:
     - Increased number of broadcasts and shifts in Fox Algorithm
     - Smaller local matrices (less work per process)
     - Higher synchronization frequency
4. **Optimal process count depends on matrix size**
   - For smaller matrices (≤ 600), optimal performance occurs around **P=4–9**.
   - For larger matrices (≥ 900), using up to **P=16 or P=25** remains beneficial, though with reduced efficiency.

Speedup and Efficiency Discussion

Let (T_1) be the time with one process and (T_P) the time with P processes. Then:

Speedup = T1 / TP
Efficiency = Speedup / P

Approximate values for (N=1200):

P	Time (s)	Speedup	Efficiency
1	15.391	1.00x	100%
4	4.922	3.13x	78%
9	4.020	3.83x	43%
16	4.138	3.72x	23%
25	4.260	3.61x	14%

We can see that:

- **Speedup** increases initially but **saturates** beyond 9 processes.
- **Efficiency** drops as more processes are added, due to the increasing proportion of communication.

Communication Overhead Analysis

The Fox Algorithm requires:

1. **Row-wise broadcasts** of A-blocks
2. **Column-wise rotations** of B-blocks
3. **Synchronization** at each iteration

Each step introduces latency that grows with (Q) (the grid dimension).
For large (Q) (e.g., 16 or 25 processes), these costs dominate the runtime, especially for smaller matrices.

This explains why:

- `P=16` and `P=25` sometimes perform **worse than P=9**, particularly for `N=300--600`.
- The computation per process becomes too small to **hide the communication cost**.

Summary of Findings

- **Parallel efficiency** is high for large matrix sizes and moderate process counts.

- **Scalability** degrades when too many processes are used for small problems.
- The algorithm achieves near-optimal performance for:
  - `N $\geq$ 900`
  - `P $\approx$ 9--16`
- Further improvements could be achieved with:
  - **Non-blocking MPI communications**
  - **Computation-communication overlap**
  - **Hybrid parallelism (MPI + OpenMP)**

Difficulties

Along the making of this project we had a few complications: "Random" Segmentation Faults, after some debug we conclude the only place that could be causing this was the MPI_Finalize step, once we tried to understant why on StackOverflow we concluded that this is a recurring problem in the library, and the fixes suggested there didn't help. Oversubscribe flag was not supported through different machines. This problem was utterly ignored after running on the cluster and nothing happened. Finally to replicate the way we treat our matrices we tried using MPI_Type_vector, for some reason we couldn't understand the program just wouldn't work, so we ended up replicating the same effect without using the provided method

Conclusion

This implementation successfully demonstrates parallel computation of all-pairs shortest paths using the Fox algorithm with MPI. The modular design allows for easy testing across different process configurations while maintaining correctness and efficiency. The comprehensive test suite ensures reliability across various matrix sizes and process counts, making it suitable for deployment on distributed computing clusters.