CP_T1 - Parallel Computing Assignment Report

Project Overview

This project implements a parallel solution for **All-Pairs Shortest Path computation** using the **Floyd-Warshall algorithm** parallelized through 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 Description

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm computes the shortest paths between all pairs of vertices in a weighted graph. It uses dynamic programming with the recurrence relation:

```
dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
```

Fox Algorithm Integration

Instead of implementing Floyd-Warshall directly, we use the **repeated squaring approach** where:

- The adjacency matrix is repeatedly multiplied by itself
- Each multiplication represents one more step in the shortest path computation
- The Fox algorithm parallelizes these matrix multiplications efficiently

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` / `io.c`	Input/output handling and matrix reading/writing.

Architecture and Data Structures

Core Structures

`struct FoxDetails` Contains the configuration and metadata for the parallel grid:

- `Q`: number of processes per dimension (grid size $Q \times Q$)
- `N`: global matrix size
- `per_process_n`: local matrix block size
- `myRow`, `myColumn`, `myRank`: process position in the grid
- `envData`: environment information (MPI context)

`struct FoxMPI` Defines the MPI communication context:

- `cart`: global 2D Cartesian communicator
- `row`, `col`: subcommunicators for row and column communication
- `datatype`: custom MPI datatype representing a matrix block

4.2 Process Grid

Processes are arranged in a **2D Cartesian topology**

Each process manages:

- A **local submatrix** of A, B, and C of size $(N/Q) \times (N/Q)$
- MPI communication with neighboring processes in its row and column.

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

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

Cluster Testing Requirements

[This section will be populated after cluster execution]

Testing Environment:

- Cluster specifications: TBD
- Node configuration: TBD
- Network topology: TBD

Benchmark Matrices:

- Small scale: 300x300, 600x600
- Medium scale: 1200x1200
- Large scale: TBD based on cluster capacity

Metrics to Collect:

- Execution time vs. process count
- Speedup calculations
- Efficiency measurements
- Communication overhead analysis
- Scalability curves

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

Processes	Matrix Size	Time (s)
	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
	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)**

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.