All-Pairs Shortest Path using Fox's Algorithm

Project Overview

This project implements a parallel solution to the **All-Pairs Shortest Path Problem** using **Fox's Algorithm** for distributed matrix multiplication with **MPI (Message Passing Interface)**. The implementation uses the **Repeated Squaring Algorithm** with **min-plus matrix multiplication** to efficiently compute shortest paths between all pairs of nodes in a directed graph.

Problem Description

What the Project Does

The program solves the all-pairs shortest path problem for directed graphs by:

1. **Input**: Reading an adjacency matrix representing a directed graph where each element (i,j) represents the weight of the edge from node i to node j
2. **Processing**: Using Fox's algorithm to perform min-plus matrix multiplications in parallel across multiple MPI processes
3. **Output**: Producing a distance matrix where each element (i,j) represents the shortest path distance from node i to node j

The Problem It Solves

Given a directed graph G = (V,E) with: - **V**: Set of vertices (nodes) - **E**: Set of edges (links) with weights

The goal is to find the shortest path distance between every pair of vertices. This is fundamental in: **- Network routing protocols - Transportation systems - Social network analysis - Game AI pathfinding - Supply chain optimization**

Parallel Computing Implementation

Fox's Algorithm

Fox's Algorithm is a parallel matrix multiplication algorithm designed for distributed memory systems. Key characteristics:

- **Process Grid**: Arranges P processes in a $\sqrt{P} \times \sqrt{P}$ grid
- **Matrix Partitioning**: Divides $N \times N$ matrices into $(N/\sqrt{P}) \times (N/\sqrt{P})$ blocks
- **Communication Pattern**: Each process communicates only with processes in the same row and column
- **Scalability**: Reduces communication overhead compared to naive parallel approaches

Min-Plus Matrix Multiplication

Instead of traditional matrix multiplication ($\times$, $+$), we use **min-plus algebra**: - **Multiplication operation** $\rightarrow$ **Addition** - **Addition operation** $\rightarrow$ **Minimum**

For matrices A and B, element (i,j) of result C:

```
C[i][j] = min{A[i][k] + B[k][j]} for all k
```

Repeated Squaring Algorithm

To find all shortest paths, we compute: - D_1 = initial adjacency matrix - $D_2 = D_1 \otimes D_1$ (shortest paths with $\leq$ 2 edges) - $D_4 = D_2 \otimes D_2$ (shortest paths with $\leq$ 4 edges) - $\ldots$ - D_n = final result (shortest paths with $\leq$ N edges)

Where $\otimes$ represents min-plus matrix multiplication.

Code Structure and Organization

Main Components

1. **Main Function (`main`)**
   - **MPI Initialization**: Sets up MPI environment
   - **Input Validation**: Checks Fox's algorithm constraints
   - **Process Grid Setup**: Creates 2D Cartesian topology
   - **Orchestration**: Coordinates the overall algorithm execution

2. **Matrix Operations**
   - `allocate_matrix`: Dynamic memory allocation for 2D matrices
   - `free_matrix`: Memory cleanup
   - `read_input_matrix`: Input parsing with zero-to-infinity conversion
   - `print_matrix`: Output formatting

3. **Core Algorithm Functions**
   - `min_plus_multiply`: Performs min-plus multiplication on matrix blocks
   - `min_plus_square`: Implements Fox's algorithm for min-plus matrix squaring
   - `fox_algorithm`: Main Fox's algorithm implementation

4. **Communication Setup**
   - **Grid Topology**: 2D Cartesian process grid with periodic boundaries
   - **Row Communicators**: For broadcasting blocks within rows
   - **Column Communicators**: For shifting blocks within columns

Data Structures

```c
// Process grid coordinates
int grid_coords[2];        // [row, column] in process grid

// Communication contexts
MPI_Comm grid_comm;        // 2D Cartesian grid
MPI_Comm row_comm;         // Row-wise communication
MPI_Comm col_comm;         // Column-wise communication

// Matrix blocks (local to each process)
int **local_A;             // Input matrix block
int **local_result;        // Result matrix block
```

Algorithm Flow

1. **Initialization**
   - Read matrix dimension and validate constraints
   - Create 2D process grid topology
   - Distribute matrix blocks to processes
2. **Repeated Squaring Loop**

   ```c
   for (int iter = 0; iter < ceil(log2(n)); iter++) {
       min_plus_square(local_result, local_result, ...);
   }
   ```
3. **Fox's Algorithm Steps** (within each squaring operation)
   - **Stage Loop**: For each stage (0 to $\sqrt{P}$ - 1)
   - **Broadcast**: Send appropriate A-block within row
   - **Multiply**: Perform local min-plus multiplication
   - **Shift**: Circularly shift B-blocks within column
4. **Result Collection**
   - Gather all matrix blocks to process 0
   - Reconstruct complete result matrix

Compilation and Execution Instructions

Prerequisites

- **MPI Implementation**: OpenMPI, MPICH, or Intel MPI
- **C Compiler**: GCC or Intel C Compiler
- **System**: Linux/Unix environment

Installing MPI (if not available) On Ubuntu/Debian systems:

```
sudo apt update
sudo apt install openmpi-bin openmpi-common openmpi-doc libopenmpi-dev
```

On CentOS/RHEL systems:

```
sudo yum install openmpi openmpi-devel
# or
sudo dnf install openmpi openmpi-devel
```

Note: If MPI is not available, the Makefile will automatically build a sequential version (`fox_sequential`) that validates the algorithm correctness using the Floyd-Warshall algorithm.

Compilation

```
# Using the provided Makefile
make

# Or manually
mpicc -Wall -Wextra -O3 -std=c99 fox.c -o fox -lm
```

Execution

Basic Usage

```
# Run with 4 processes
mpirun -np 4 ./fox < input.txt

# Run with 9 processes
mpirun -np 9 ./fox < input.txt
```

Input Format

```
6
0 2 0 5 0 0
0 0 0 0 0 0
0 2 0 0 0 5
0 0 0 0 1 0
3 9 3 0 0 0
0 0 0 0 1 0
```

Using Makefile Targets

```
# Create test input file
make test_input

# Run with different process counts
make run1    # 1 process
make run4    # 4 processes
```

```bash
make run9    # 9 processes

# Performance benchmark
make benchmark
```

Expected Outputs and Performance Analysis

Example Output

For the sample 6×6 graph input:

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

Performance Characteristics

Time Complexity

- **Sequential**: $O(N^3)$ for Floyd-Warshall
- **Parallel**: $O(N^3/P + \log N \times \text{communication\_cost})$
- **Communication**: $O(\sqrt{P} \times N^2/P)$ per iteration

Space Complexity

- **Per Process**: $O(N^2/P)$ for local matrix blocks
- **Total**: $O(N^2)$ distributed across all processes

Expected Speedup For ideal conditions with P processes: - **Theoretical Maximum**: P (perfect scaling) - **Realistic**: 0.7P to 0.9P (considering communication overhead)

Performance Factors

1. **Matrix Size**: Larger matrices show better parallel efficiency
2. **Process Count**: Must be perfect squares (1, 4, 9, 16, 25, …)
3. **Network Latency**: Affects communication-intensive phases
4. **Load Balance**: Fox's algorithm provides excellent load distribution

Benchmark Results Format

```
Execution time: X.XX ms
Process Count: P
Speedup: S.SS
Efficiency: E.EE%
```

Algorithms and Techniques Used

1. Fox's Algorithm

- **Purpose**: Parallel matrix multiplication for distributed memory
- **Key Idea**: 2D block decomposition with systematic communication
- **Advantage**: O($\sqrt{P}$) communication complexity vs O(P) for naive approaches

2. Min-Plus Algebra

- **Operations**: (min, +) semiring instead of (+, ×) ring
- **Properties**: Associative, commutative (min), distributive
- **Application**: Shortest path problems map naturally to min-plus algebra

3. Repeated Squaring

- **Concept**: Compute A, A^2, A^4, A^8, $\ldots$ until $A^{2^k} \geq A^N$
- **Efficiency**: O(log N) matrix multiplications vs O(N) for naive approach
- **Convergence**: Result stabilizes when all shortest paths are found

4. MPI Cartesian Topology

- **Grid Creation**: `MPI_Cart_create` for 2D process arrangement
- **Subcommunicators**: Row and column communicators for efficient broadcasting
- **Coordinates**: Logical process positioning for algorithm coordination

5. Communication Patterns

- **Broadcast**: Row-wise distribution of A-blocks
- **Circular Shift**: Column-wise rotation of B-blocks
- **Point-to-Point**: Final result gathering

Error Handling and Constraints

Validation Checks

1. **Perfect Square Processes**: P = Q^2 where Q is integer
2. **Matrix Divisibility**: N mod Q = 0
3. **Input Format**: Proper matrix dimension and values
4. **Memory Allocation**: Robust error checking for malloc failures

Error Messages

```
Error: Number of processes (X) must be a perfect square
Error: Matrix dimension (N) must be divisible by sqrt(processes) (Q)
Error reading matrix element [i][j]
Memory allocation failed
```

Implementation Notes

Optimizations Applied

1. **Memory Layout**: Contiguous allocation for better cache performance
2. **Communication Overlap**: Asynchronous operations where possible
3. **Compiler Optimizations**: -O3 flag for aggressive optimization
4. **Infinity Handling**: Efficient representation using INT_MAX

Debugging Features

- **Timing Information**: Execution time measurement (excluding I/O)
- **Error Reporting**: Detailed error messages with context
- **Rank-based Output**: Process 0 handles all I/O operations

Scalability Considerations

- **Memory Per Process**: $O(N^2/P)$ scaling
- **Communication Volume**: $O(N^2/\sqrt{P})$ per process
- **Synchronization Points**: Minimized barrier operations

This implementation provides a robust, scalable solution to the all-pairs shortest path problem, demonstrating key concepts in parallel computing, distributed algorithms, and high-performance computing with MPI.