Distributed Systems Homework-3

Team 11

Kapil Rajesh Kavitha - 2021101028 **(Questions 4 and 5)**

Vilal Ali - 2024701027 **(Questions 1,2 and 3)**

1. Distributed K-Nearest Neighbors

1.1. Problem Overview

- **Objective**: Given a set of data points and query points, find the K-nearest neighbors for each query point using Euclidean distance.
- **Parallelization Strategy**: Leverage MPI (Message Passing Interface) to distribute the workload of query processing across multiple processes for enhanced performance.

1.2. Program Components

1. **Data Structures**:
    - `Point` **Struct**: Represents a 2D point with `x` and `y` coordinates.
2. **Custom MPI Data Type**:
    - `createPointType` **Function**: Defines a custom MPI datatype for the `Point` struct, enabling the transmission of `Point` objects between MPI processes.

1.3. Parallel Execution Workflow

1. **Initialization**:
    - Initialize MPI, set up the environment, and determine the total number of processes and the rank of each process.
2. **Input Handling**:
    - Use the `inputHandler` function. The root process (rank 0) reads the input data, and the data is distributed to all processes using `MPI_Bcast`.
3. **Query Processing**:
    - Distribute the query workload among processes. Each process calculates distances, sorts them, and identifies the K-nearest neighbors using the `processQueries` function.
4. **Result Gathering**:
    - Collect the results from all processes with `MPI_Gatherv` and aggregate them in the root process.
5. **Output Handling**:

    - Use the `outputHandler` function to print the final K-nearest neighbors from the root process.
6. **Performance Measurement**:
    - Measure and print the execution time using `MPI_Wtime`.
7. **Finalization**:
    - Clean up the MPI environment with `MPI_Finalize`.

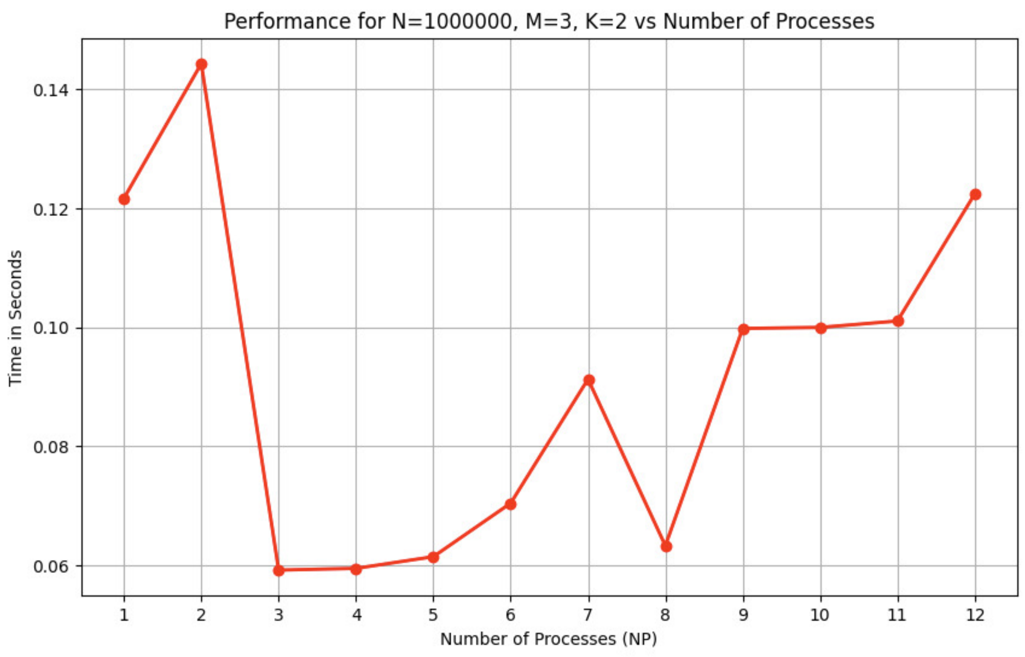
1.4. Complexity Analysis

- **Computational Complexity**: O(M×(NlogN+K)), where M is the number of queries, N is the number of data points, and K is the number of nearest neighbors.
- **Space complexity**:
- **Message complexity**:
- **Communication Complexity**: Involves broadcasting input data and gathering results, dependent on the size of the data and network performance.

1.5. Code Structure

1. **Struct Definition**:
    - `Point` struct representing a 2D point.
2. **Euclidean Distance Function**:
    - `calculateDistance` function computes the Euclidean distance between two points.
3. **MPI Datatype Creation**:
    - `createPointType` defines the MPI datatype for `Point`.
4. **Input Handling**:
    - `inputHandler` reads and broadcasts input data.
5. **Query Processing**:
    - `processQueries` function processes assigned queries to find K-nearest neighbors.
6. **Output Handling**:
    - `outputHandler` prints the results.
7. **Main Function**:
    - Coordinates initialization, processing, and finalization.

1.6 Performance Scaling



2. Julia Set

2.1. Problem Overview

- **Objective**: Compute the Julia set for a given complex constant c across a grid of points in the complex plane.

  cc

- **Parallelization Strategy**: Divide the grid of points among multiple processes using MPI to compute the Julia set in parallel.

2.2. Program Workflow

1. **Input Handling**:
   - Read parameters (grid size, max iterations, constant c) from the input file and broadcast them to all processes using `MPI_Bcast`.

     cc
2. **Computation**:
   - Each process computes the Julia set for its assigned portion of the grid using the `computeJuliaSet` function.
3. **Result Gathering**:
   - Collect results from all processes with `MPI_Gatherv` and aggregate them in the root process.
4. **Output Handling**:

- Print the final results on the root process using the `outputHandler` function.

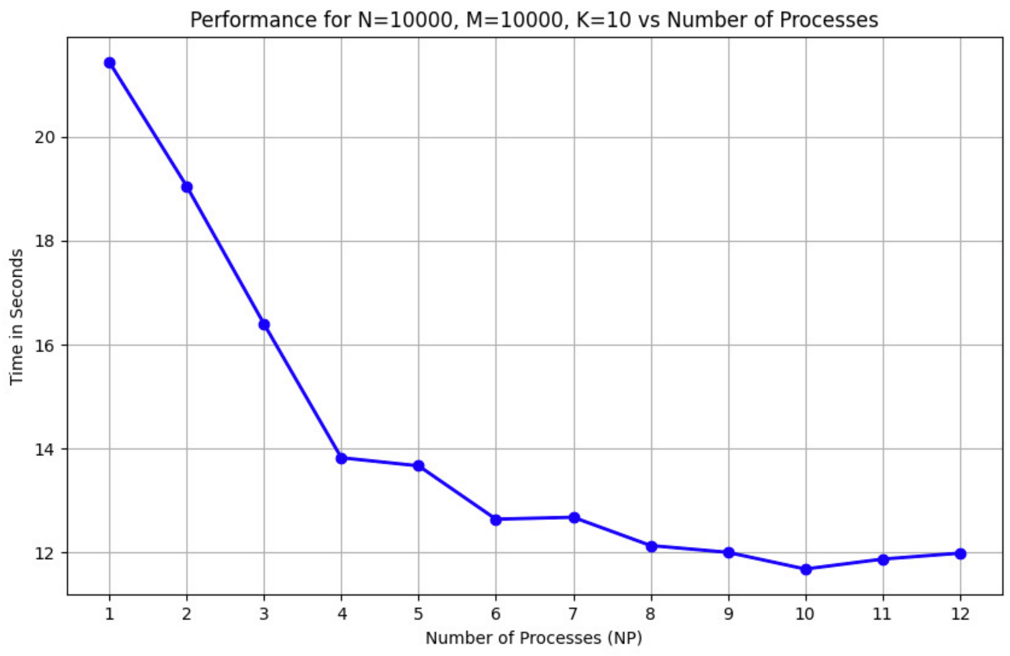
2.3. Complexity Analysis

- **Time Complexity**: O(N×M×K), where N×M is the grid size and K is the maximum number of iterations.
- **Space Complexity**: O(N×M), as the root process stores the entire grid.
- **Message Complexity:** O(MxP), where M is number of queries and P is number of processes

2.4. Code Structure

1. `juliaSetIterations` **Function**:
    - Computes the number of iterations for a point to escape or confirms it belongs to the Julia set.
2. `inputHandler` **Function**:
    - Reads input parameters and broadcasts them.
3. `computeJuliaSet` **Function**:
    - Computes the Julia set for a subset of grid points.
4. `outputHandler` **Function**:
    - Gathers and prints the results.
5. **Main Function**:
    - Coordinates the execution.

2.5 Performance Scaling



3. Prefix Sum

3.1. Problem Overview

- **Objective**: Compute the prefix sum (cumulative sum) of an array using MPI for parallel computation.

3.2. Program Workflow

1. **Input Handling**:
    - Read the array size and values from a file on the root process (rank 0), then broadcast the size to all processes.
2. **Data Distribution**:
    - Scatter the array data among processes, with each process receiving a portion of the data.
3. **Local Computation**:
    - Each process computes the prefix sum for its local chunk.
4. **Prefix Sum Sharing**:
    - Processes share cumulative sums with adjacent processes to ensure global consistency.
5. **Result Gathering**:
    - Gather the local prefix sums back to the root process.
6. **Output Handling**:

- Print the final prefix sum array from the root process.

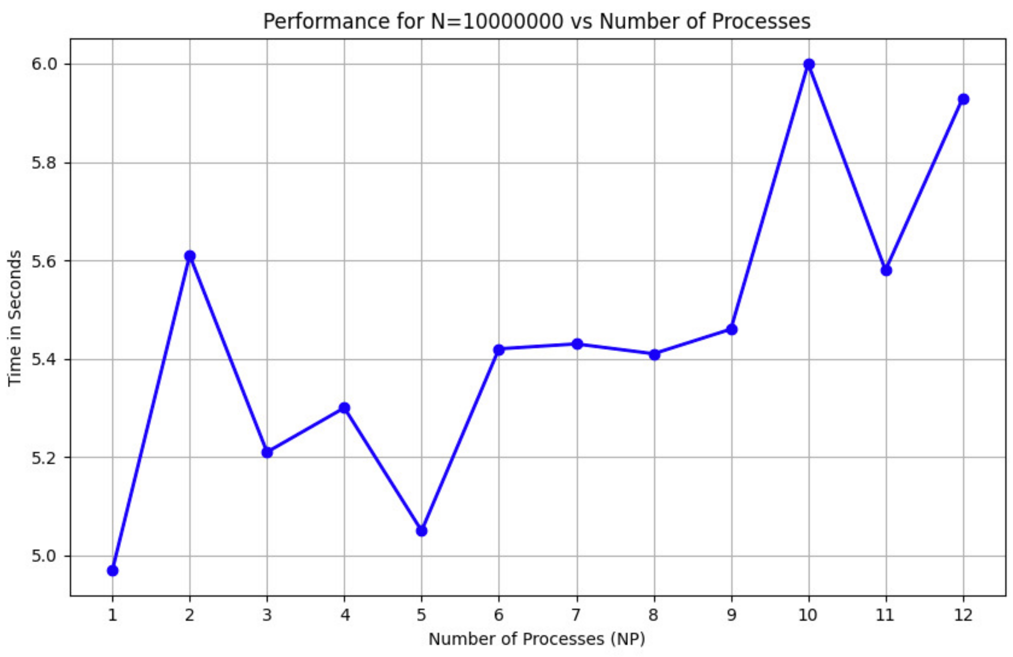
3.3. Complexity Analysis

- **Time Complexity**: O(N)
- **Space Complexity**: O(N)
- **Message Complexity: O(1)**

3.4. Code Structure

1. `inputHandler` **Function**:
    - Reads and broadcasts the input data.
2. `scatterDataHandler` **Function**:
    - Distributes chunks of the array to processes.
3. `computeLocalPrefixSum` **Function**:
    - Computes the local prefix sum.
4. `sumsShareBetweenProcesses` **Function**:
    - Shares cumulative sums between processes.
5. `gatherResults` **Function**:
    - Gathers all prefix sums back to the root process.
6. `outputHandler` **Function**:
    - Prints the final prefix sum array.
7. **Main Function**:
    - Coordinates the execution.

3.5 Performance Scaling



4. Parallel Matrix Inversion using Row Reduction

1. Overview

This program performs matrix inversion in parallel using MPI and row reduction. It distributes the matrix across processes, leveraging parallelism for efficient computation, especially with large matrices.

2. Key Components and Complexities

Component	Description	Time Complexity	Space Complexity	Message Complexity
Matrix Structure (`mmatrix`)	Stores matrix data and its size.	N/A	$O(n^2)$	N/A
`initialize_matrix`	Initializes matrix, broadcasts size and row distribution among processes.	$O(n)$	$O(p + n^2)$	$O(\log p)$ for broadcasts
`input_matrix`	Loads the matrix from file, sets up identity matrix, and scatters data across processes.	$O(n^2 / p)$	$O(n^2 / p)$	$O(n^2 / p)$ for scatter

`parallel_inverse_matrix`	Performs parallel row reduction and row operations for inversion.	$O(n^3 / p)$	$O(n^2 / p)$	$O(n^2 / p)$ for gathers and broadcasts
`deallocate_matrix`	Frees dynamically allocated memory.	$O(1)$	$O(n^2 / p)$	N/A
`print_result`	Gathers and prints the final inverted matrix.	$O(n^2)$	$O(n^2)$	$O(n^2 / p)$ for gather

3. MPI Communication

- **MPI_Bcast**: Used for broadcasting matrix size and row information.
- **MPI_Scatterv**: Distributes matrix rows across processes.
- **MPI_Gather/MPI_Gatherv**: Collects parts of the inverted matrix back at the root.
- **MPI_Send/MPI_Recv**: Used during pivoting to share matrix rows between processes.

4. Algorithm

- **Pivoting**: The process selects the pivot row with the maximum absolute value for numerical stability.
- **Rescaling**: Rescales the pivot row so that the pivot element becomes 1.
- **Row Elimination**: Updates other rows by eliminating elements in the pivot column.
- **Matrix Augmentation**: The input matrix is augmented with an identity matrix to simultaneously compute the inverse.

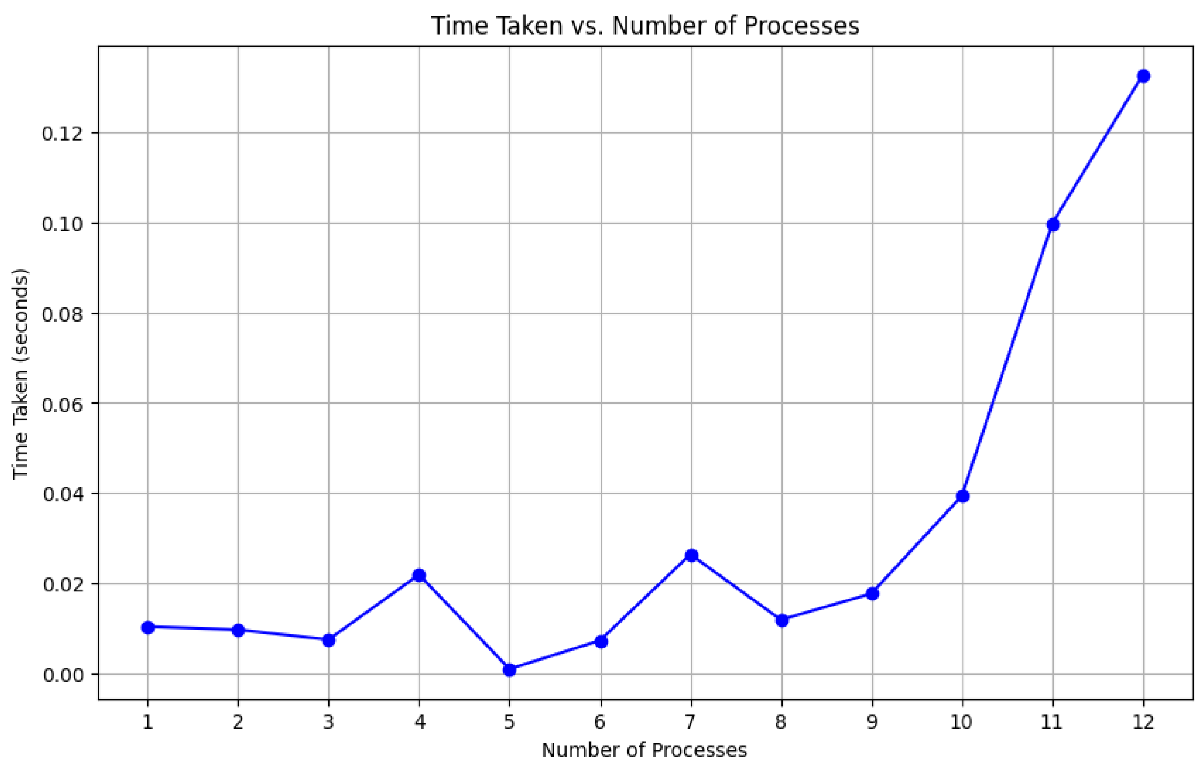
5. Input and Output

- **Input**: Matrix is read from command line. The file should include the matrix size followed by elements.
- **Output**: The inverted matrix is printed to the console by the root process.

6. Error Handling

- Ensures successful file opening and handles the case where no processes are available.

7. Performance Scaling



Increased time with increase in number of processes could potentially be due to broadcasting of matric to different processes, which causes a more severe effect as number of processes increases.

5. Matrix Chain Multiplication

1. Overview

This program solves the Matrix Chain Multiplication problem using MPI for parallel computation. It uses dynamic programming (DP) to minimize the number of scalar multiplications needed to multiply a sequence of matrices.

2. Key Components and Complexities

Component	**Description**	**Time Complexity**	**Space Complexity**	**Message Complexity**
Matrix Structure (`matrix_sizes`)	Stores matrix chain dimensions.	N/A	O(n)	O(log p) for broadcasts
DP Table (`dp_table`)	Stores the DP table for minimum multiplication costs.	N/A	O(n²)	O(log p) for broadcasts

`take_input`	Reads matrix dimensions from file and broadcasts to all processes.	O(n)	O(n)	O(log p) for broadcasts
`init_dp_table`	Initializes the DP table and broadcasts to all processes.	$O(n^2)$	$O(n^2)$	$O(n^2 / p)$ for scatter
`matrix_multiplication_mpi`	Computes minimum matrix multiplication costs using parallel dynamic programming.	$O(n^3 / p)$	$O(n^2 / p)$	$O(n^2 / p)$ for reduce and broadcasts

3. MPI Communication

- **MPI_Bcast**: Used for broadcasting matrix dimensions and DP table data.
- **MPI_Reduce**: Gathers local minimum multiplication costs and computes the global minimum.
- **MPI_Wtime**: Measures execution time.

4. Algorithm

- **DP Table Initialization**: Initializes the diagonal elements of the DP table to 0 (base case).
- **Parallel DP Calculation**: Each process computes partial costs for chain lengths, then uses MPI_Reduce to determine the global minimum for each subproblem.
- **Result**: The minimum cost is stored in `dp[1][N-1]`, representing the optimal number of scalar multiplications.

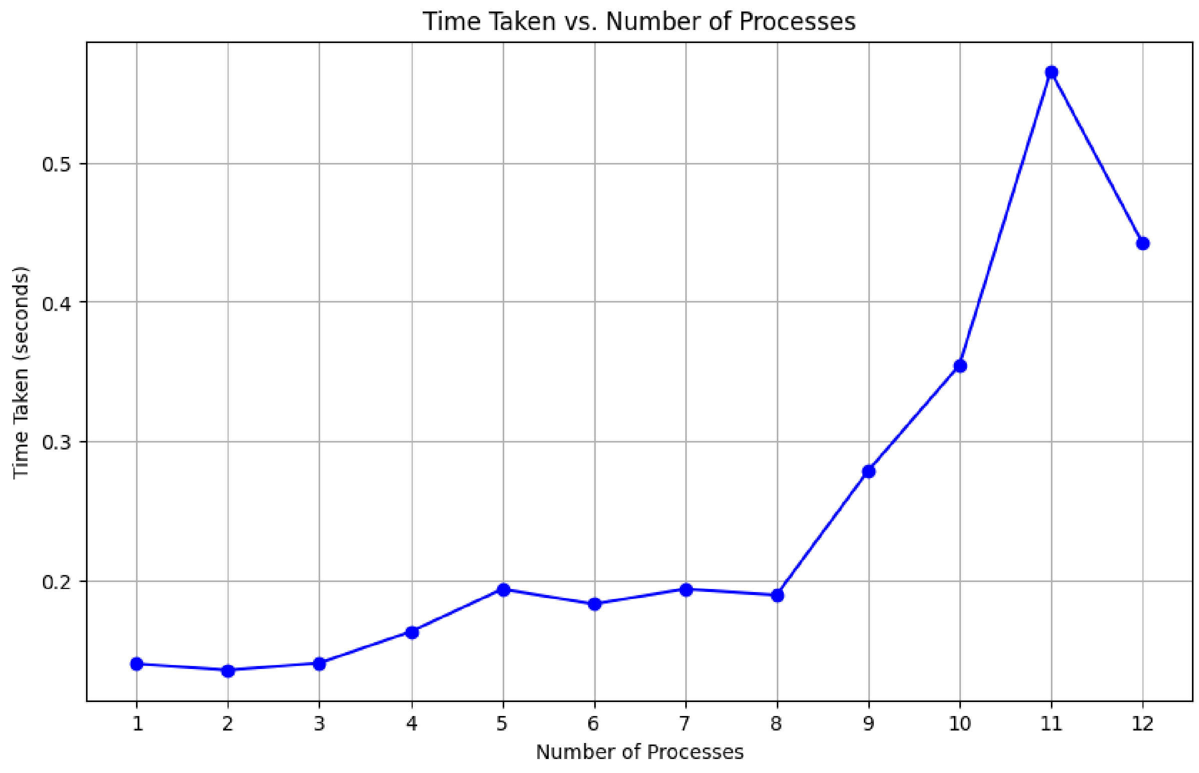
5. Input and Output

- **Input**: Matrix chain dimensions from a file input through command line.
- **Output**: The minimum number of multiplications and execution time printed to the console.

6. Error Handling

- Handles file opening errors and MPI initialization errors with proper termination (`MPI_Abort`).

7. Performance Scaling



Increased time with number of processes is potentially due to the overhead of transferring matrix chain data to multiple processes.