Distributed Systems Homework 3

Team 62
Vineeth Bhat (2021101103) and Mohammed Omer (2024909002)

September 2, 2024

Introduction

Questions 1 and 3 have been done by M. Omer.
Questions 2 and 4 have been done by V. Bhat.
Question 5 was done together.

Complexities and Experiments have been added to their respective question sections. Time has been measured in seconds.

For sequential output statistics, please consider the output of the program with one process in use.

Q1

Approach:

1. Read the input data on the root process (rank 0).
2. Distribute the points in set P evenly among all processes.
3. Broadcast the query points (set Q) to all processes.
4. Each process calculates the K nearest neighbors for its subset of P for all query points.
5. Gather the results from all processes to the root process.
6. The root process selects the final K nearest neighbors for each query point.
7. Output the results.

Explanation:

- We define structures `Point` and `DistancePoint` to represent 2D points and points with their distances.
- The `euclideanDistance` function calculates the Euclidean distance between two points.

- The `findKNearestNeighbors` function finds the K nearest neighbors for a given query point from a set of points.

In the main function:

1. We initialize MPI and get the rank and size of the communicator.
2. The root process (rank 0) reads the input data.
3. We broadcast N, M, and K to all processes.
4. We distribute the points P evenly among all processes using `MPI_Scatter`.
5. We broadcast the query points Q to all processes.
6. Each process computes the K nearest neighbors for all query points using its subset of P.
7. We gather the results from all processes using `MPI_Gather`.
8. The root process combines the results, selects the final K nearest neighbors for each query point, and prints the output.

Time Complexity: $O\left(M \cdot \frac{N}{p} + N \cdot \log(K)\right)$, where p is the number of processes.
Message Complexity: $O(N \cdot \log p + K \cdot p)$ for broadcasting, scattering, and gathering data.
Space Complexity: $O\left(\frac{N}{p} + K \cdot p\right)$ in total, with $O\left(\frac{N}{p}\right)$ per process for local storage.

Experiments

We generate 3 large test cases with N between 20,000 and 30,000, Q between 1 and 10,0000 and K between 1 and N.

Here are the results:

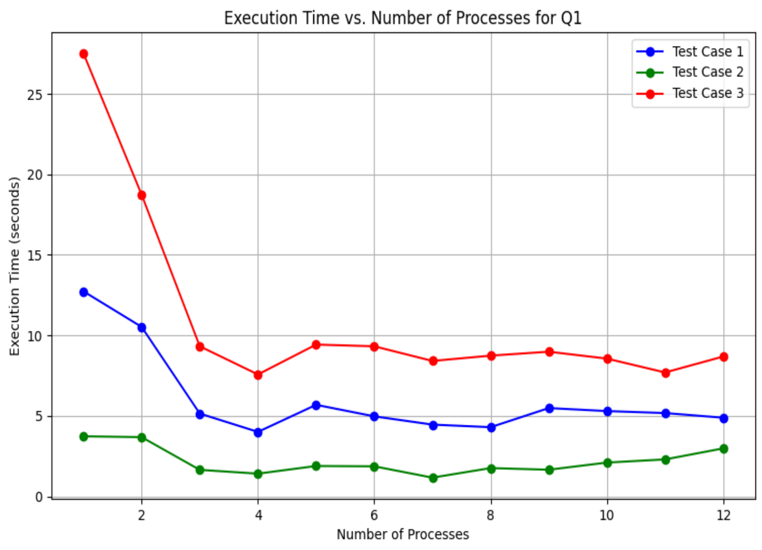


Table 1: Q1 Execution Time for Different Number of Processes and Test Cases

Number of Processes	Test Case 1	Test Case 2	Test Case 3
1	12.7354	3.73234	27.511
2	10.5264	3.68008	18.737
3	5.14391	1.64886	9.3207
4	4.00589	1.41245	7.57254
5	5.68634	1.8885	9.429
6	4.97133	1.86841	9.3225
7	4.45812	1.16548	8.41325
8	4.29813	1.75949	8.74151
9	5.48444	1.65654	8.9862
10	5.2985	2.101	8.5577
11	5.17502	2.29821	7.6966
12	4.88654	2.9823	8.6866

General Trend: As the number of processes increases, the execution time for Test Cases 1 and 2 generally decreases. This suggests that the distributed approach scales well with more processes, which helps to reduce execution time.

Fluctuations: The rate of decrease slows down as the number of processes increases, indicating diminishing returns in execution time reduction with additional processes. This can be attributed to the number of cores being limited to 8 and high message passing overhead.

Q2

Approach:

1. Read the input data on the root process (rank 0).
2. Distribute the input parameters (grid dimensions, iteration count, threshold, and complex constant) to all processes.
3. Each process calculates its portion of the Julia set using a distributed approach.
4. Gather the results from all processes to the root process.
5. The root process combines and prints the final Julia set output.

Explanation:

- The `read_file` function reads the input parameters from a file.
- The `square_complex_number` and `julia_next_iteration` functions handle the Julia set iterations.
- The `compute_for_point_per_process` function checks if a point is within the Julia set.
- The `attach_grid_coordinates` function maps grid indices to the complex plane.
- The `compute_answer_per_proc` function generates the results for its assigned portion of the grid.

In the main function:

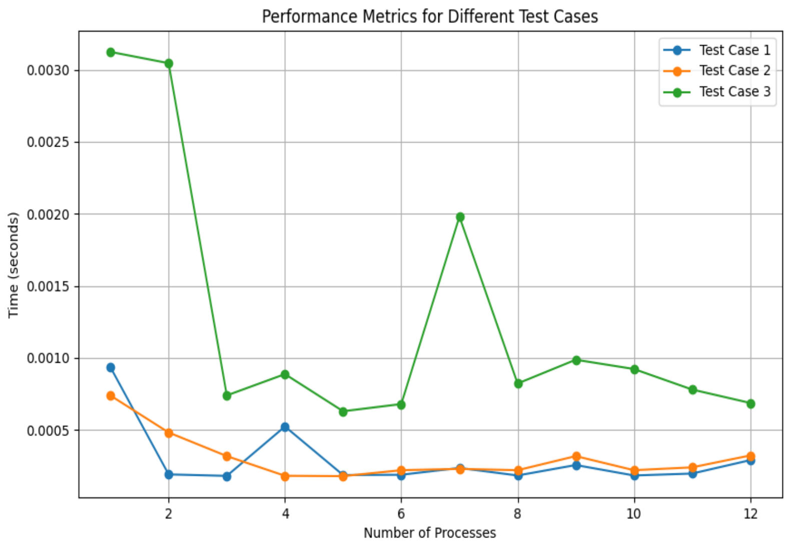
1. Initialize MPI and obtain the rank and size of the communicator.
2. The root process (rank 0) reads the input data from a file.
3. Distribute the input parameters to all processes using `MPI_Bcast`.
4. Each process computes the portion of the Julia set assigned to it using the `compute_answer_per_proc` function.
5. Gather the results from all processes using `MPI_Gatherv` and combine them at the root process.
6. The root process prints the final combined output.

Time Complexity: $O\left(\frac{N \cdot M \cdot K}{p}\right)$, where p is the number of processes.
Message Complexity: $O(N \cdot M + p)$
Space Complexity: $O\left(\frac{N \cdot M}{p}\right)$ per process while it is $O\left(N \cdot M\right)$ for process 0.

Experiments

We generate 3 large test cases with all parameters set to 100 and randomly generated constant points.

Here are the results:



Number of Processes	**Test Case 1**	**Test Case 2**	**Test Case 3**
1	0.000935109	0.000738968	0.00312329
2	0.000192	0.000482805	0.00304508
3	0.000181678	0.000318904	0.000739883
4	0.000524222	0.000182317	0.000887769
5	0.000187313	0.000179921	0.000630128
6	0.000189524	0.000221168	0.000680131
7	0.000236779	0.000230217	0.00198017
8	0.000184465	0.000221168	0.0008226
9	0.00025698	0.000318809	0.0009876
10	0.000184465	0.00022113	0.00092312
11	0.0001978	0.00024083	0.000780851
12	0.0002909	0.000323268	0.000687599

Table 2: Performance metrics for different test cases across varying numbers of processes.

General Trend: The execution time tends to decrease with an increase in the number of processes, indicating that parallelization can be effective in reducing execution time.

Fluctuations: The fluctuations can be attributed to device inefficiencies and context switching time (not all cores were available). Possibly due to high messaging overhead.

Q3

Approach:

1. Distribute the input array evenly among all processes.
2. Each process computes the local sum of its portion.
3. Perform a parallel prefix sum on the local sums.
4. Each process adds its prefix sum to its local elements.
5. Gather the results to the root process.

Explanation:

- The `parallelPrefixSum` function computes the local prefix sum and then combines the results using a parallel algorithm.

In the main function:

1. We initialize MPI and get the rank and size of the communicator.
2. The root process (rank 0) reads the input data.
3. We broadcast N to all processes.
4. We calculate the local array size for each process, handling cases where N is not evenly divisible by the number of processes.
5. We distribute the array using `MPI_Scatterv` to handle uneven distribution.
6. Each process computes its local prefix sum using the `parallelPrefixSum` function.
7. We gather the results from all processes using `MPI_Gatherv` to handle potentially different local array sizes.
8. The root process prints the final prefix sum array.

Time Complexity: $O\left(\frac{N}{p} + p\right)$, where N is the total number of elements and p is the number of processes.
Message Complexity: $O(N + p)$ for scattering, gathering, and allgather operations.
Space Complexity: $O(N)$ in total, with $O\left(\frac{N}{p}\right)$ per process for local storage.

Experiments

We generate 3 large test cases with number of elements between 2,000,000 and 3,000,000.

Here are the results:

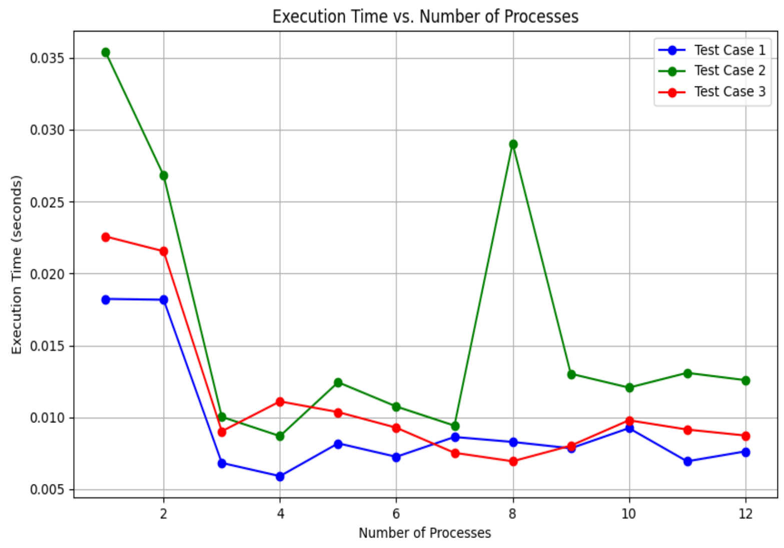


Table 3: Execution Time for Different Number of Processes and Test Cases

Number of Processes	**Test Case 1**	**Test Case 2**	**Test Case 3**
1	0.0182306	0.0354022	0.0225724
2	0.0181721	0.0268139	0.0215432
3	0.00682764	0.010028	0.00902637
4	0.0059083	0.00869353	0.0111029
5	0.00818877	0.0124341	0.0103615
6	0.00725241	0.0107508	0.0092764
7	0.00862968	0.00941917	0.00753092
8	0.00828574	0.0290467	0.00694065
9	0.00784255	0.0130268	0.00802622
10	0.00924513	0.0120623	0.00979171
11	0.00693789	0.0130892	0.00914822
12	0.00762791	0.0125822	0.00873181

General Trend: As the number of processes increases, the execution times in general reduce. This suggests that the distributed approach scales well with more processes, which helps to reduce execution time.

Fluctuations: The rate of decrease slows down as the number of processes increases, indicating diminishing returns in execution time reduction with additional processes. This can be attributed to the number of cores being limited to 8 and high message passing overhead.

Q4

Approach:

1. Read the input matrix data on the root process (rank 0).
2. Broadcast the matrix size N and matrix data to all processes.
3. Initialize the identity matrix on all processes.
4. Perform Gaussian elimination with partial pivoting in a distributed manner:
   (a) Identify the owner process for each pivot row.
   (b) Find and broadcast the next non-zero row to handle row swapping.
   (c) Broadcast and update pivot rows across all processes.
   (d) Eliminate rows below the pivot in parallel.
5. Perform back substitution to complete the inversion:
   (a) Broadcast and update pivot rows from top to bottom.
   (b) Perform backward elimination in parallel.
6. Gather and print the final inverted matrix on the root process.

Explanation:

- The `read_file` function reads the input matrix from a file.
- The `allocate_square_array` function allocates memory for a square matrix.
- The `swap_rows_in_square_matrix` function swaps rows in the matrix.
- The `gaussian_elimination` function performs Gaussian elimination to transform the matrix into its reduced row echelon form.
- The `back_substitution` function performs back substitution to obtain the final inverse matrix.

In the main function:

1. Initialize MPI and get the rank and size of the communicator.
2. Read the matrix data on the root process and broadcast it to all processes.
3. Initialize the identity matrix on all processes.

4. Execute Gaussian elimination and back substitution in parallel.

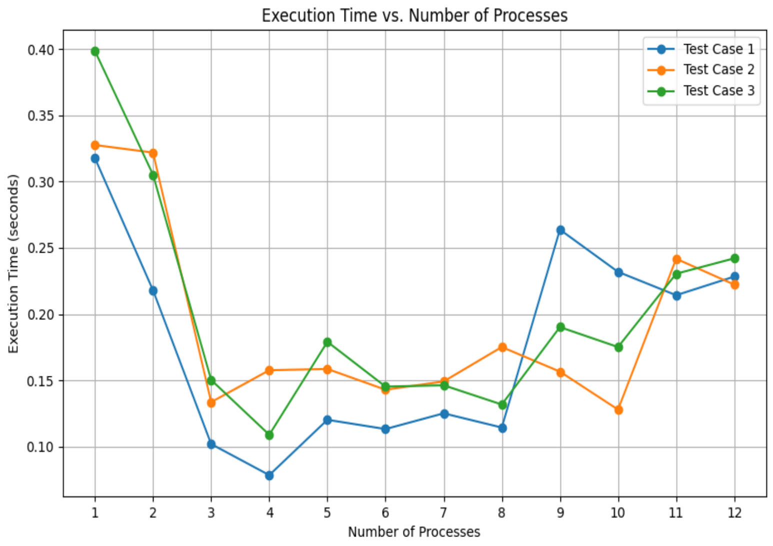
5. The root process prints the final inverted matrix.

Time Complexity: $O\left(\frac{N^3}{p}\right)$, where p is the number of processes.
Message Complexity: $\theta(N^2 \cdot log(p))$ on an average; $O(N^2 \cdot p$ in the theoretical worst case stemming from row swaps (which will not happen as not every diagonal element while calculating the r.r.e.f. can become 0.);
Space Complexity: $O\left(N^2\right)$ per process.

Experiments

We generate 3 large test cases with randomly generated square matrices of size 300 to 400.

Here are the results:



General Trend: The execution time tends to decrease with an increase in the number of processes, indicating that parallelization can be effective in reducing execution time.

Fluctuations: The fluctuations can be attributed to device inefficiencies and context switching time (not all cores were available). Possibly due to high messaging overhead and system memory limitations.

Number of Processes	Test Case 1 (s)	Test Case 2 (s)	Test Case 3 (s)
1	0.317729	0.327561	0.398546
2	0.217968	0.321839	0.305183
3	0.101994	0.133610	0.150336
4	0.078529	0.157670	0.108975
5	0.120272	0.158617	0.179098
6	0.113283	0.142943	0.145298
7	0.125152	0.149193	0.146316
8	0.114436	0.175171	0.131802
9	0.263741	0.156623	0.190193
10	0.231865	0.127984	0.175172
11	0.214250	0.241757	0.230450
12	0.228303	0.222259	0.242074

Table 4: Execution times for different test cases across varying numbers of processes for Q4.

Q5

Approach:

1. Read matrix chain dimensions from the input file on the root process (rank 0).
2. Broadcast the matrix dimensions to all processes.
3. Initialize the dynamic programming (DP) array to store the minimum multiplication costs.
4. Perform parallel matrix chain multiplication using a distributed approach:
   (a) Distribute the computation of subproblems across processes.
   (b) Each process computes results for a portion of the DP array.
   (c) Use MPI's `MPI_Allgatherv` to gather and distribute the results.
5. Output the final result (minimum cost of matrix chain multiplication) on the root process.

Explanation:

- The `read_file` function reads matrix chain dimensions from an input file.
- The DP array is initialized to store intermediate results.
- The matrix chain multiplication is performed in parallel, distributing computations and gathering results using MPI.
- The final result is output by the root process.

In the main function:

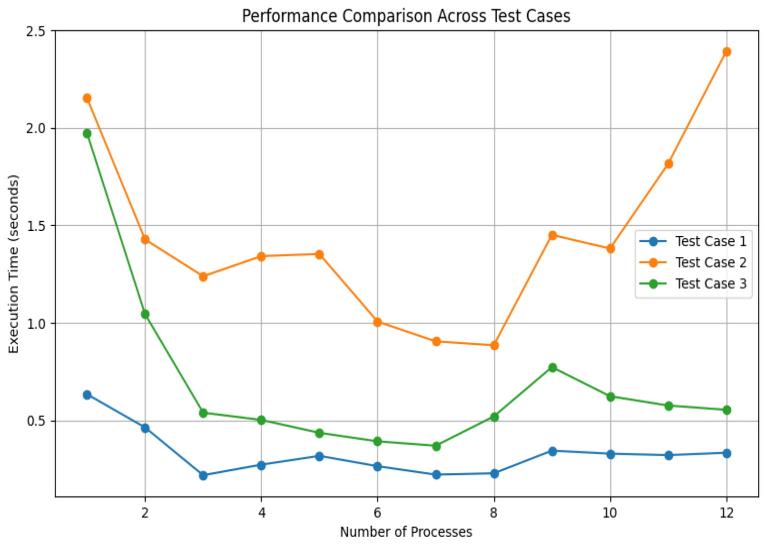
1. Initialize MPI and get the rank and size of the communicator.
2. Read matrix dimensions on the root process and broadcast to all processes.
3. Initialize the DP array and perform the matrix chain multiplication in parallel.
4. Gather and print the final result on the root process.

Time Complexity: $O(N^3/p)$, where N is the number of matrices and p is the number of processes.
Message Complexity: $O(N^2)$ per iteration (in DP calculation) due to gather operations (absolute worst case).
Space Complexity: $O(N^2)$ per process.

Experiments

We generate between 500 to 1000 dimensions.

Here are the results:



General Trend: As the number of processes increases, the execution times in general reduce. This suggests that the distributed approach scales well with more processes, which helps to reduce execution time.

Fluctuations: The rate of decrease slows down as the number of processes increases, indicating diminishing returns in execution time reduction with additional processes. This can be attributed to the number of cores being limited to 8 and high message passing overhead.

Upshot towards the end for one of the test cases can be treated as an outlier.

Number of Processes	**Test Case 1 (s)**	**Test Case 2 (s)**	**Test Case 3 (s)**
1	0.633845	2.15472	1.975222
2	0.463317	1.42799	1.04469
3	0.218107	1.23886	0.539058
4	0.272285	1.34242	0.501738
5	0.317482	1.35323	0.435592
6	0.26448	1.0064	0.391875
7	0.221116	0.905052	0.369312
8	0.227836	0.884728	0.519555
9	0.344149	1.45131	0.772607
10	0.32876	1.38117	0.623261
11	0.321255	1.81598	0.575745
12	0.333444	2.39303	0.553798

Table 5: Performance across different numbers of processes for three test cases