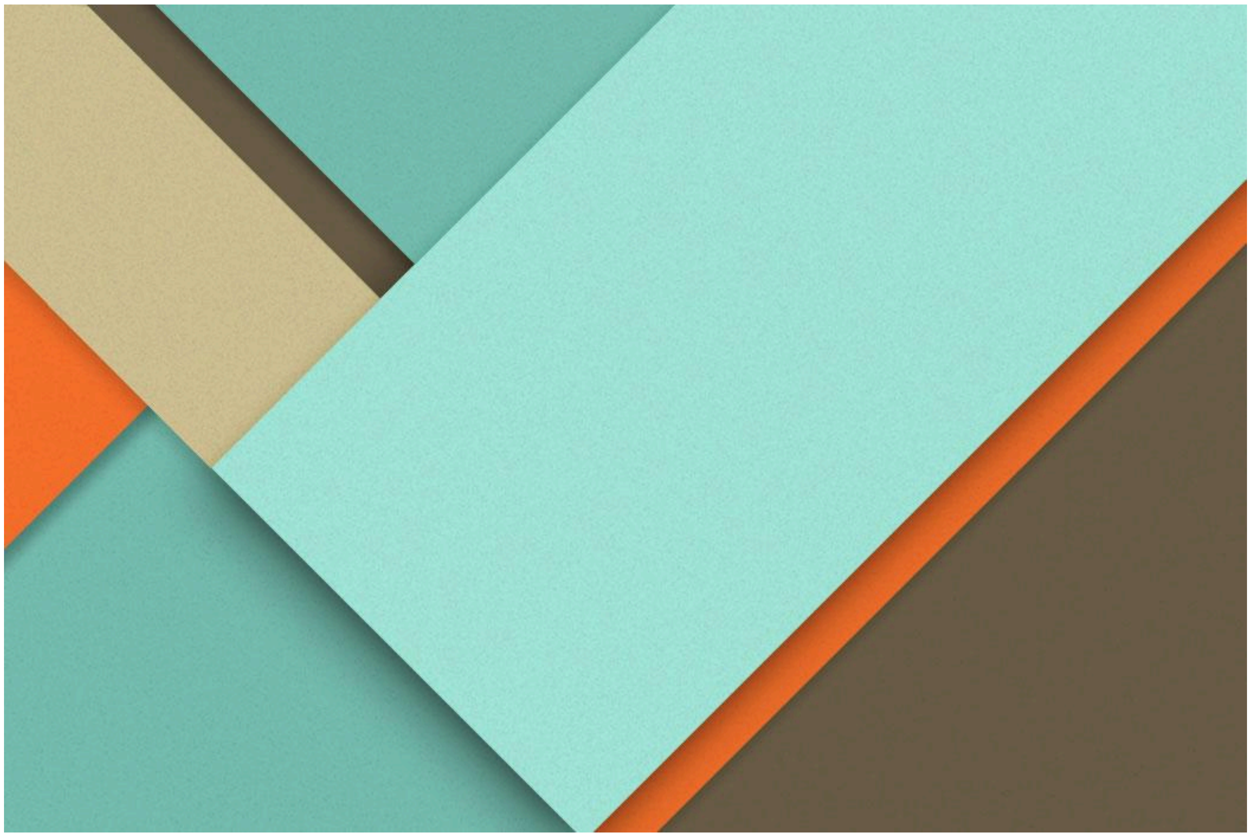


Homework 3 Report

Team Name:

Atharva Pande (2023201065)

Varun Edachali (2022101029)

3.1 K Nearest Neighbours

On running the code on a randomised input with 1000 points (N), 1000 queries (M) and 850 nearest neighbours to find (K) in rce, the processing took the following time:

No of Processes	Completion Time (ns)
1	1005029270
2	518639301
3	365378043
4	279540348
5	229166262
6	195905920
7	172412264
8	152237457
9	139929504
10	127350135
11	117891724
12	110661644

Let p be the number of processes we run with.

Solution Approach

- Broadcast the N points across the processes. (using *MPI_BCast)*
- Segment the queries into p parts, and send them to the relevant processes in O(p) time. (by getting the required bounds and using *MPI_Send* and *MPI_Recv*)
- For each query, store the result points in a max-heap of size not more than K, with the comparator checking the distance between the query point and the point in consideration. We pop from the heap if we have more than K elements.
  - This leads to a time complexity of O(N * log(K)) per query. Since we parallelise across processes, we have a total time complexity of O((M / p) * N log K).
- Get the results from each query and store them according to their ranks to ensure the output is in the appropriate order.

For efficient space complexity, each process receives only the queries relevant to it, instead of broadcasting all of the queries. This leads to a base complexity of O((M / p) + N) per process, not including the O((M / p) * K) space needed to store the results of the algorithm.

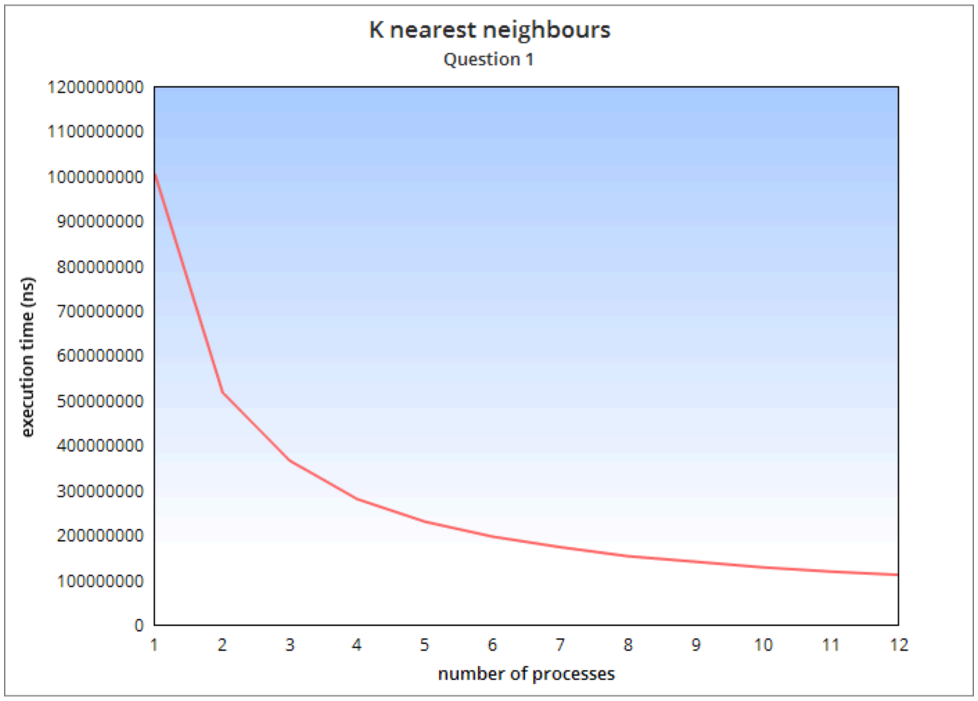
Highlights of the Program

Parallelisation is achieved by distributing queries across available processes.

We use a max-heap rather than a traditional sort approach to reduce the time complexity from O(N*log(N)) down to O(N * log(min(N, K)).

The message complexity is about O(N + M*k) as this is the extent of communication between processes.

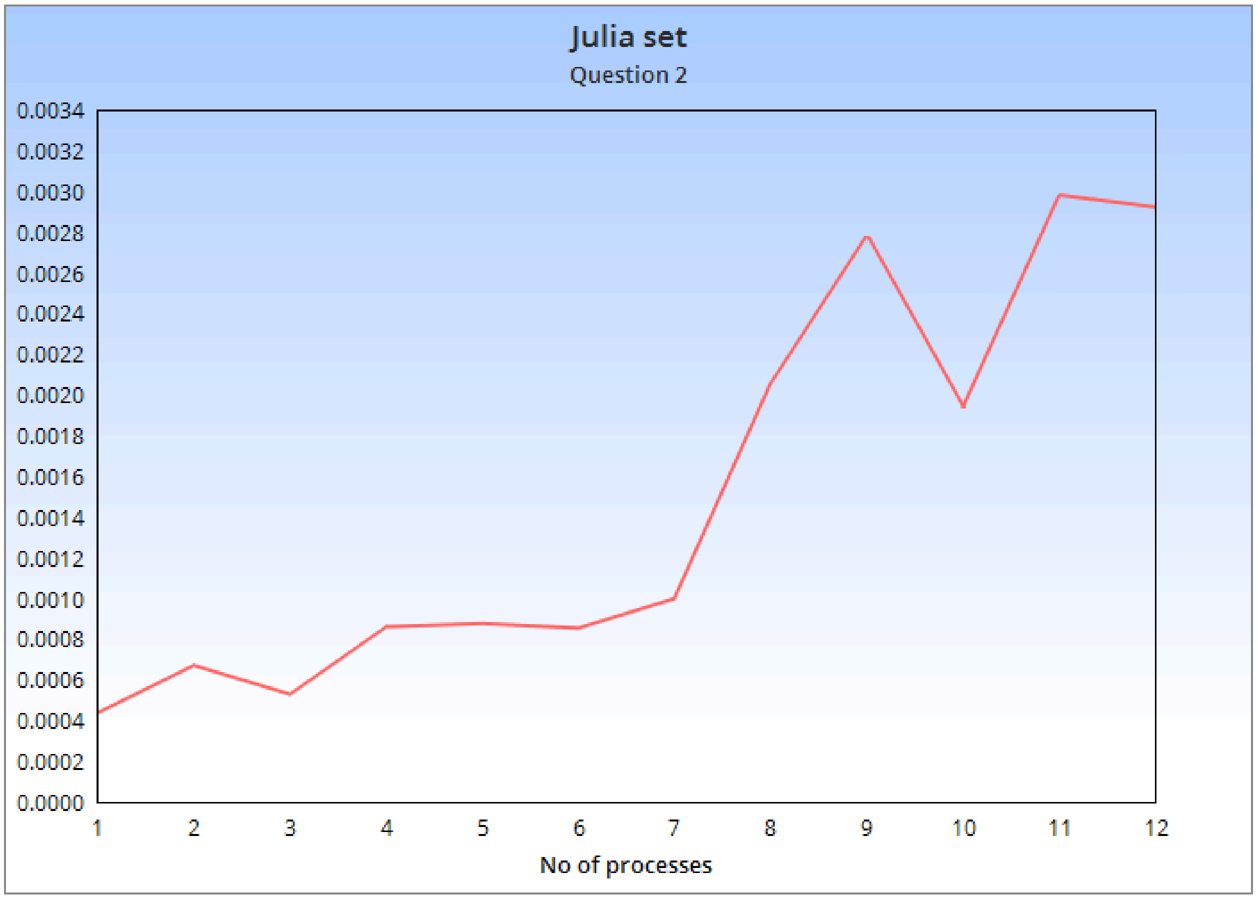
Result Analysis



Despite the execution time decreasing, we notice that the rate of decrease starts slowing down as the number of processes increases. This could be because the constant factors increase, as we have to perform more sends and receives across processes. This slows down algorithms practically, even though it is not considered in our algorithm complexity.

3.2 Julia Set

No of Processes	Completion Time
1	0.000437609
2	0.000670381
3	0.000528986
4	0.000860037
5	0.000875536
6	0.000853718
7	0.000998928
8	0.00205522
9	0.00278324
10	0.00193945
11	0.00297939
12	0.00292087



Report: MPI-Based Julia Set Computation

1. Solution Approach

Objective: The goal of this program is to compute the Julia set, a type of fractal, using parallel processing with MPI (Message Passing Interface). The Julia set is determined by iterating complex numbers and checking whether the sequence diverges. The program efficiently distributes the workload across multiple processes to speed up the computation.

Approach:

1. **Input Reading:**
   - The program reads the dimensions of the computation grid NNN (rows) and MMM (columns), the maximum number of iterations KKK, and the constant complex number ccc from an input file specified via a command-line argument.

- The root process (rank 0) is responsible for reading the input file and then broadcasting these values to all other processes.

2. **Parallel Computation:**
   - The grid is divided among the available MPI processes. Each process computes a subset of the rows in the grid, ensuring an even distribution of workload.
   - Each process calculates the Julia set for its assigned rows by iterating over the grid points and applying the Julia set formula: zn+1=zn2+cz_{n+1} = z_n^2 + czn+1=zn2+c.
   - If a complex number's magnitude exceeds 2 before reaching the maximum iteration count KKK, it is considered to have diverged. The result is stored in a local results array.
3. **Synchronization and Gathering Results:**
   - After each process completes its portion of the computation, the results are gathered by the root process using `MPI_Gatherv`. This method ensures that the results are correctly ordered and assembled into the full grid.
   - The root process then outputs the computed Julia set to the standard output.
4. **Execution Time Measurement:**
   - The program measures the time taken for the entire computation using `MPI_Wtime`. The root process prints the total execution time, providing insight into the performance of the parallel computation.

2. Code Highlights

1. **MPI Initialization and Finalization:**
   - The program begins by initializing MPI with `MPI_Init(&argc, &argv)` and ends with `MPI_Finalize()` to properly manage the parallel environment.
2. **Input Handling:**
   - The root process reads the input parameters from a file specified by the user. These parameters are then broadcasted to all processes using `MPI_Bcast` to ensure consistent data across all processes.
3. **Julia Set Computation:**
   - The computation of the Julia set is parallelized by dividing the rows of the grid among the processes. Each process works on a distinct portion of the grid, computing whether each point belongs to the Julia set based on the number of iterations required for divergence.
4. **Result Gathering and Output:**
   - The results from all processes are gathered using `MPI_Gatherv`. This function is well-suited for scenarios where processes contribute different amounts of data, as it allows the results to be correctly aligned and combined.

    - The root process then prints the Julia set, ensuring the rows are output in the correct order.

5. **Performance Measurement:**
    - Execution time is measured using `MPI_Wtime`, which captures the wall-clock time required for the computation. This metric is critical for assessing the performance gains achieved through parallelization.

3. Performance Considerations

1. **Scalability:**
    - The program is designed to scale efficiently with the number of processes. Larger grids (higher NNN and MMM) benefit more from parallelization, as the workload is distributed among more processes.
2. **Load Balancing:**
    - The program dynamically divides the grid rows among processes, ensuring that each process has a nearly equal amount of work. This approach minimizes idle time and ensures that all processes contribute effectively to the computation.
3. **Communication Overhead:**
    - While `MPI_Bcast` and `MPI_Gatherv` introduce some communication overhead, the benefits of parallel processing generally outweigh these costs, especially for large grid sizes. Efficient communication strategies are crucial for maintaining high performance.

Time Complexity is O(N * M * K / p) because we segment across processes

Space complexity is O(N * M / p) per process

The overall message complexity is O(N * M) for gathering the results.

3.3 Prefix Sum

The following times are based on an input of 1e6 random floating point numbers, running on rce.

No of Processes	Completion Time (ns)
1	1456687
2	1287549
3	1294879
4	1271922
5	1282701
6	1283944
7	1249547
8	1211948
9	1247842
10	1274885
11	1301839
12	1344226

Let p be the number of processes.

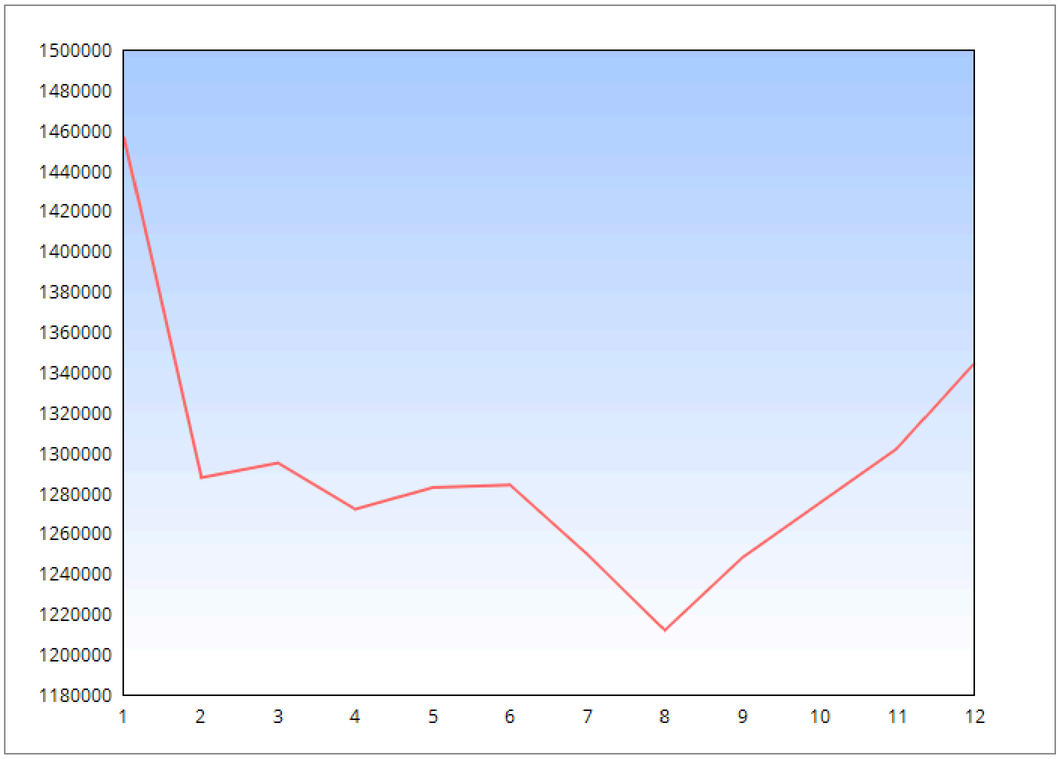
Solution Approach

- Segment the array into p nearly equal pieces, and send them to each of the processes in O(p) (by getting the required bounds and using *MPI_Send* and *MPI_Recv*)
- The p^{th} process calculates the intermediate prefix sums of the p^{th} segment in O(N/p).
- The last elements are sent back to the root process, which uses it to calculate the offsets to be added to each segment in time O(p).
- These offsets are then added back to the intermediate prefix sums in time O(N/p)
- Finally, these intermediate prefix sums are accumulated back at the root, to create the final prefix sum array.

Highlights of the Program

- The O(N / p) time complexity is achieved majorly due to the summation of the last elements of the intermediate prefix sums. These use the property that $(a_1 + a_2 + \dots + a_k) + (a_{k+1} + \dots + a_l) = (p_k + (p_l - p_k))$ where a_i is the ith element of the original array, and p_i is the ith element of the prefix sum array. This allows us to calculate the weight of each segment in O(p) time by retrieving the last element of the intermediate segments, accumulating them and sending them back.
- The space complexity is just O(n / p) per process, not counting the input values and the final array to store. Including those, it gets boosted to O(n). The message complexity is also just O(N) as we communicate the entire array.

Analysis of Results



Initially, we notice a reduction in time aligning with the analysis. However, since the sequential execution itself has a low constant coefficient (it can be executed in a single pass), the sends and receives begin behaving as bottlenecks, increasing the practical time required for execution without causing a significant advantage in processing time.

The slightly jagged results could be because of a number of factors, such as the heat of the cache, the number of cores available, and so on. Despite this, the general trend is clear.

3.4 Matrix Inversion

The following times were obtained on a random matrix of size 750, running on rce.

No of Processes	Completion Time (ns)
1	6318908516
2	4135986295
3	3067061777
4	2381702371
5	1983147027
6	1689127587
7	1484515090
8	1315933645
9	1180189886
10	1068688981
11	979102108
12	905103063

Solution Approach

- First, segment the rows (of the matrix, and the corresponding identity matrix) into p segments and distribute them to the relevant processes, in O(N) time using *MPI_Send* and *MPI_Recv*.
- Next, we perform gaussian elimination.
    - For each i of N rows:
        - If while processing row i, we have Aii = 0, we have to find the next row with a non zero value of Axi. We then swap our row with that one. This, in the worst case, takes O(n) time.
        - Set the scale as Aii. Divide all the elements in the current row by Aii. For each subsequent row, perform Rj = Rj - cRi to make all of Axi

(where x > i) 0. This is performed parallelly across chunks, requiring approximately O((N / p) * N) time.

To differentiate between the two types of messages, messages from row i requesting a row with non-zero Axi are sent with tag 1. If the next received message by i has tag 1, then we have found such a row, else we need to keep searching.

If row j receives a request with tag 0, this means we need to eliminate / reduce our values on the basis of row i.

Both messages send row i of both the matrix and the identity matrix to row j. In the first case, row j sends its own row to row i on success and sets its row to row i. In the second case, it simply performs the reduction.

- Next, we perform the back substitution
    - For each zeroingColumn from N to 1
        - For each row above the zeroingColumn row (this step is parallelised)
            - Perform Rj = Rj - cRi to reduce values A[j][i] to 0.
        - Since this is parallelised, it takes O((N / p) * N) time, with each chunk needing to perform this reduction on its rows.
- Finally, bring the rows of the identity back to the root process, taking O(N) sends and receives.

Thus, we reduce the time complexity to O(N^3 / p)

The space complexity is O(n^2 / p) per process not counting the initial whole matrices we store and those on output, in those cases, it boosts to O(N^2).

The message complexity is O(N^2), we may communicate all of the rows of the matrix for our operations.

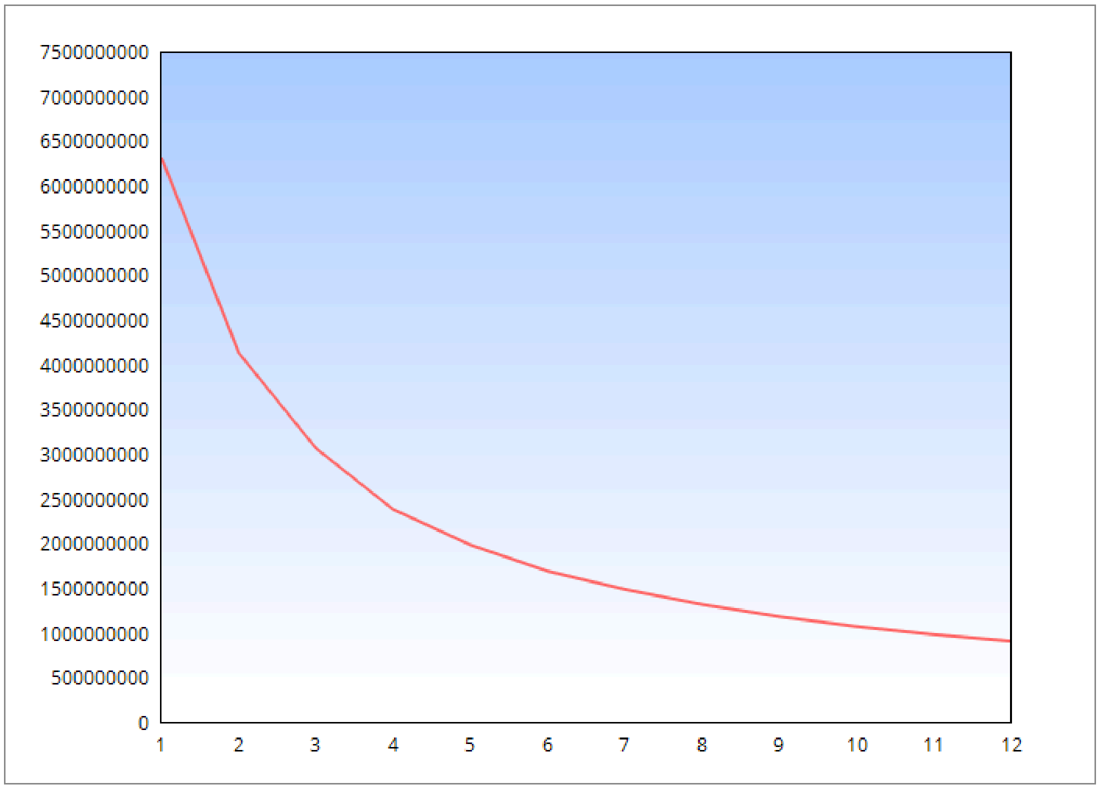
Highlights of the Approach

Matrix inversion involves a lot of steps that rows conduct independently with respect to another. For example, during gaussian elimination, the reduction steps of subsequent rows can take place independently using only the knowledge of the pivot row values. These are parallelised using message passing.

Even during back substitution, while reducing the rows above a zeroingCol, we can simply send over the necessary row, and the rest can be reduced in parallel.

A flaw of the approach is that rows often sit idle, due to improper load balancing as highlighted by this resource by University of Buffalo. This makes column distribution far more efficient than row distribution in practical cases for matrix inversion by row reduction.

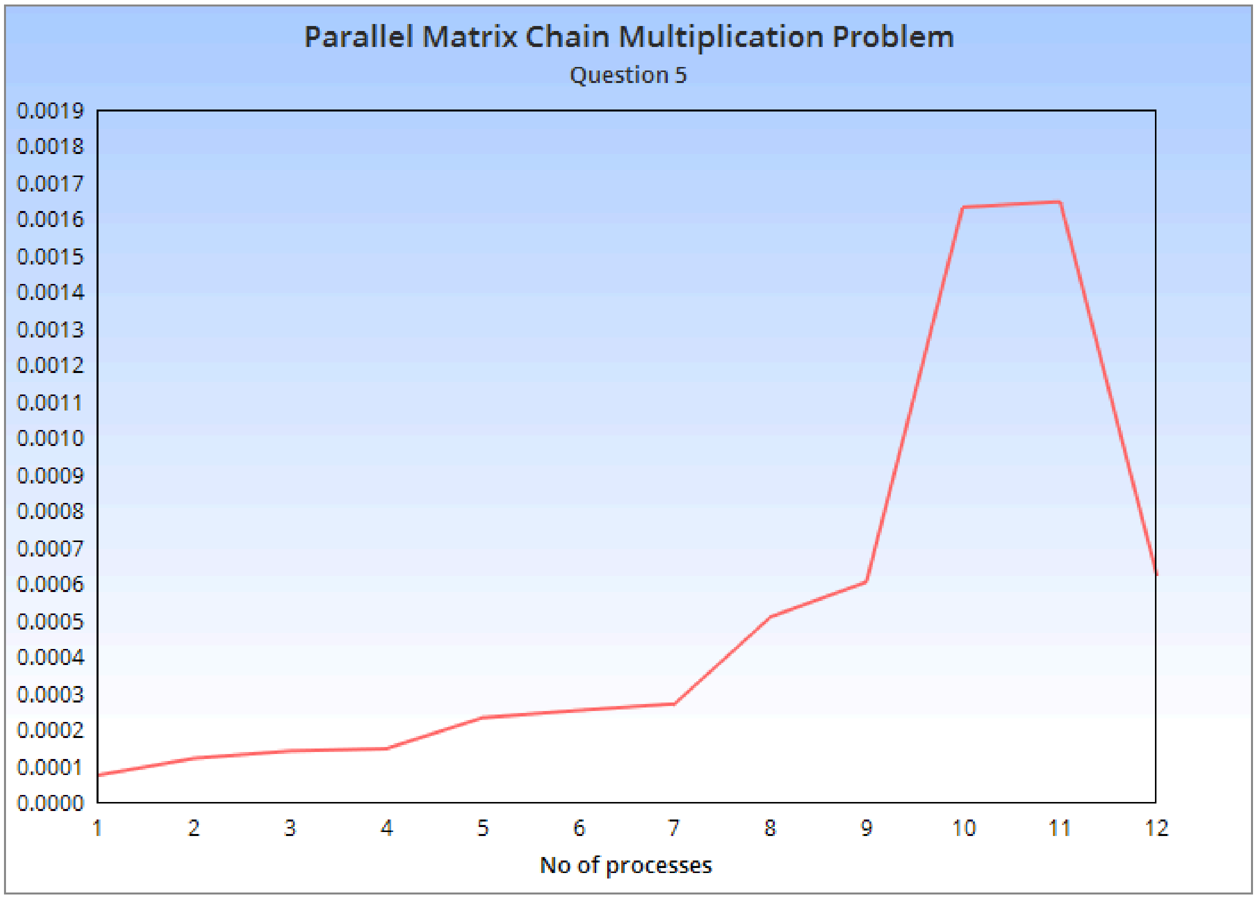
Analysis of Results



The run-times largely agree with our analysis. The rate of decrease in runtime decreases with an increase in the number of processes however, as an increased number of communication requirements leads to a higher constant factor and consequent slower overall processing.

3.5 Parallel Matrix Chain Multiplication Problem

No of Processes	Completion Time
1	0.00007299
2	0.00011935
3	0.000139867
4	0.000146167
5	0.000230946
6	0.000251127
7	0.000269507
8	0.000508277
9	0.000604124
10	0.00163168
11	0.00164738
12	0.000627153



Report on MPI-Based Matrix Chain Multiplication Solution

1. Solution Approach

Objective: The goal is to parallelize the Matrix Chain Multiplication problem using MPI (Message Passing Interface). This problem involves determining the most efficient way to multiply a chain of matrices to minimize the total number of scalar multiplications required.

Approach:

1. **Input Reading:**

   - The program reads matrix dimensions from an input file. The first line of the file contains the number of matrices, `N`, followed by `N+1` integers representing the dimensions of the matrices.
2. **Initialization:**
   - MPI is initialized to manage parallel processes.
   - The root process (rank 0) reads the input file, extracts the matrix dimensions, and broadcasts them to all other processes.
3. **Dynamic Programming (DP) Table Computation:**
   - A DP table `dp` is used to store the minimum number of scalar multiplications required for different matrix chain products.
   - The computation is parallelized based on the chain length `l` (from 2 to `N`). Each process is assigned a subset of the rows of the DP table based on its rank.
4. **Synchronization:**
   - After computing each part of the DP table, the results are synchronized across all processes using `MPI_Bcast` to ensure consistency and completeness of data.
5. **Execution Time Measurement:**
   - The time taken for the computation is measured using `MPI_Wtime`, and the results are printed by the root process.

Code Highlights:

- **MPI Initialization and Finalization:**
  - `MPI_Init(&argc, &argv)` and `MPI_Finalize()` are used to start and end MPI execution.
- **Input Handling:**
  - Input is read from a file, and `MPI_Bcast` is used to broadcast the matrix dimensions and values to all processes.
- **Dynamic Programming Computation:**
  - Parallelization is achieved by distributing the work of computing the DP table across processes. Each process computes parts of the table based on its rank.
- **Synchronization:**
  - `MPI_Bcast` ensures that all processes have the latest data from the DP table before moving to the next iteration.
- **Execution Time Measurement:**
  - Time measurement is done using `MPI_Wtime` to evaluate the performance of the parallel implementation.

2. Code Summary

- **Input File Handling:**
    - Reads matrix dimensions and broadcasts them to all MPI processes.
- **Matrix Chain Multiplication DP Computation:**
    - Utilizes a nested loop to compute the minimum number of scalar multiplications. The outer loop iterates over chain lengths, and the inner loop distributes the work based on process ranks.
- **Broadcasting Results:**
    - Synchronizes results among processes to ensure the DP table is consistent across all ranks.

3. Performance Considerations

- **Scalability:**
    - The performance scales with problem size. For small matrices, increasing the number of processes can introduce overhead that may outweigh the benefits of parallelization.
- **Communication Overhead:**
    - As the number of processes increases, the communication overhead for broadcasting and synchronization can affect performance.
- **Load Balancing:**
    - Efficient load balancing is achieved by assigning different rows of the DP table to different processes, but careful consideration is needed for larger problem sizes to avoid idle processes.

Time complexity is O(N^3 / p) since we segment across p processes

Space complexity is O(N^2 / p) as this is how much each process is responsible for.

Assuming a broadcast takes O(n) complexity throughout the report, the message complexity is O(N^3) as well due to the repeated broadcasts.