Distributed Systems HW-3 Report

Team Number: 55

Team Members: Mitansh Kayathwal, Pradeep Mishra

Roll Numbers: 2021101026, 2023801013

Note: For message complexity, the time taken for the message transfer($log(p)$) has also been included in the report. Also, the total combined message complexity across all processes has been presented in this report.

Question - 1

Solution Approach

There were 2 approaches that could have been taken to distribute the code for this problem.

For this question, consider the following **notation**:

n ⇒ Number of coordinates of set - P

m ⇒ Number of query points

k ⇒ The number of k-Nearest Neighbours required

p ⇒ The number of processes

Approach-1

We can distribute the k-NN algorithm, i.e. distribute the data points (n) across the different processes, estimate the best k-points from the data-points at every process, send them to the root process i.e. process with rank 0 and combine them at the root process. This is done for every query point iteratively, we take an approach to outputing the

Asymptotics

Computational Time Complexity: $O((nm/p)log(n/p) + mpklog(pk))$

Each process receives n/p data points and needs to find the nearest neighbors for m query points. Finding the nearest neighbors locally for each query point takes $O((n/p)log(n/p))$ per query point, so for m query points, we get a factor of m. Rest of time is for merging at root process.

Message Complexity: $O(pm + n + pk)$

Distributing the queries require $O(n)$, $O(pk)$ is for broadcasting messages to the root and an additional $O(pm)$ for broadcasting the queries to all processes.

Space Complexity:

1. **Per Process:** $O(n/p + m + k)$
2. **Root Process:** $O(n/p + m + pk)$

Space for storage of data points, queries as well as local results across processes.

Approach-2

We can distribute the query points (m) across different processes, in this scenario the k-NN for different query points are handled independently and after every process computes the k-NN for it's assigned query points, all of them are broadcast to the root process where they are combined and output.

Asymptotics

Computational Time Complexity: $O((nm/p)log(n) + mk)$

Each process handles m/p query points and has access to all n data points. Finding the nearest neighbors for each query point takes $O(nlog(n))$ so the total time is increased by a factor of m/p due to distribution of query points.

Message Complexity: $O(pmk + np + m)$

Distributing the n data points requires $O(np)$ messages. An additional $O(pmk)$ for gathering data at the root process for output and $O(m)$ for distributing query points.

Space Complexity:

1. **Per Process:** $O(mk/p + n)$
2. **Root Process:** $O(mk + n)$

Space for storage of data points, queries as well as local results across processes.

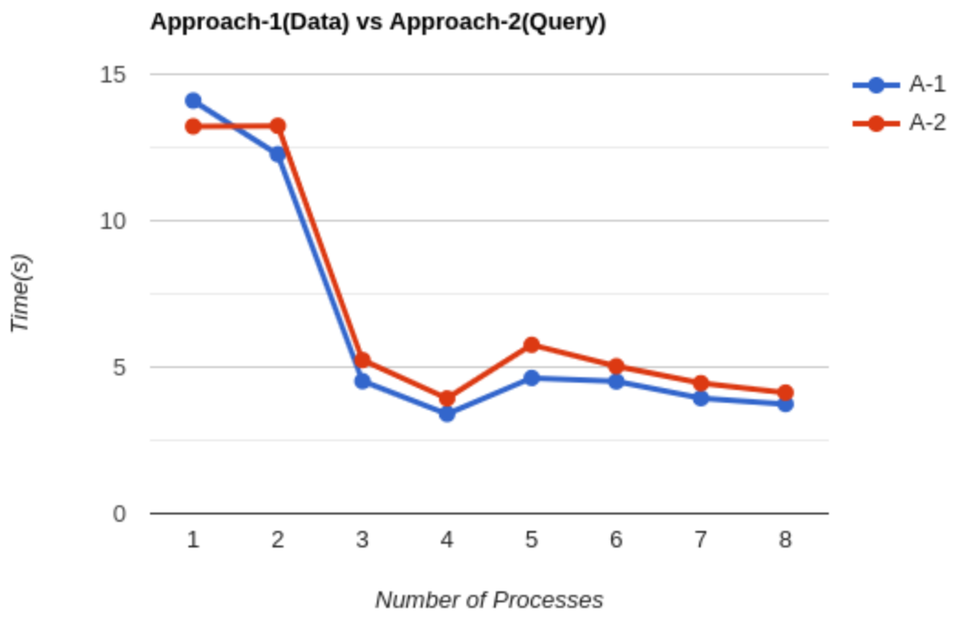
After doing a comparative experimental analysis on large test-cases, I found that **Approach-1** outperforms **Approach-2**, a sample graph which was obtained can be found here(It's only plotted till 8 processes, as k was 43 in the test-case considered, and there was a considerable overhead after 8 processes due to asymptotics, hence the reason). This was also due to compute restriction, it can be accepted that at more higher asymptotics, **Approach-1** will perform better at higher scale due to almost similar Time Complexity, however the message complexity of **Approach-2** is considerably higher as it's dependent on **both m and k**.

Key Highlights (For Approach-1)

Each process independently calculates distances between query points and a subset of data points, sorting the results to determine the k-nearest neighbors. This local sorting reduces the amount of data that needs to be communicated between processes, enhancing overall efficiency. After local k-nearest neighbors are determined, the results from all processes are aggregated and re-sorted at the

root process to determine the final k-nearest neighbors. Thus, thus improves the overall computational overhead

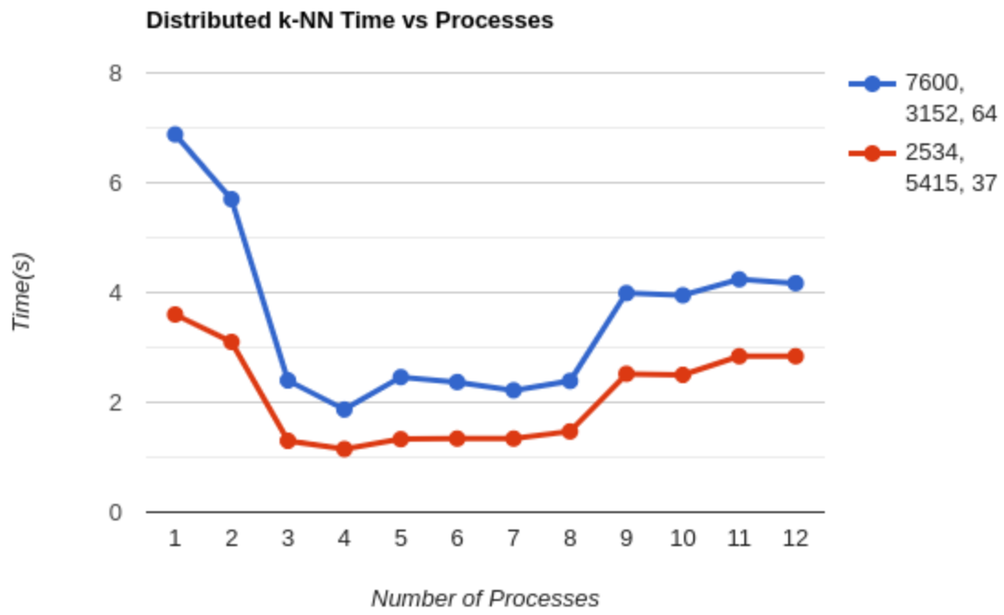
Graph for **asymptotics** for a test-case with **n = 9264, m = 4731, k = 43:**



The scale of the graph is not a great indicator, but overall, distributing data is performing much better than distributing queries. Overall, to cite actual numbers the data can be found here:

Number of Processes	Time for Approach-1(s)	Time for Approach-2(s)
1	14.10	13.22
2	12.27	13.24
3	4.52	5.24
4	3.40	3.93
5	4.63	5.76
6	4.51	5.03
7	3.94	4.45
8	3.74	4.13

Thus, I have used **Approach-1** in my actual code file, **1.cpp.** The description of the approach and asymptotics have already been provided above. On scale testing, this is the graph obtained across various different test-cases:



The graph shows the tuple (n,m,k) at the top-right for which the time plots were drawn.

This is the time measured across different number of processes, we see a sharp improvement in the time as we increase the number of processes. The slight increase in time is due to the increased overhead of communication due to relatively smaller k. Due to the approach we have taken, the additional distribution of k points across processes is causing more number of computations and thus combinations at the root process. It's still well below the brute-force benchmark of $p = 1$.

Question - 2

Solution Approach

The approach taken in this question was to distribute the grid points. I visualize the 2D-matrix as a flattened linear matrix to distribute across processes. This allows to contiguously calculate the indices of the allotted points to every process very easily. I then compute the **real** and **imaginary** parts after every iteration, and check with the threshold. I have used **square of the Euclidean Distance** in my code to avoid any floating point errors due to taking **square-root.**

Key Highlights

The program distributes the computation of the Julia Set across multiple processes using MPI, significantly speeding up the task by dividing the workload among processes.

Asymptotics

For this consider the following **notation**:

n ⇒ Number of rows in the grid

m ⇒ Number of columns in the grid

p ⇒ The number of processes

Computational Time Complexity: $O(nmk/p)$

The time complexity is due to division of points across the p processes, thus an improvement by a factor of p. The remaining complexity is for the computation of the Julia-Set algorithm.

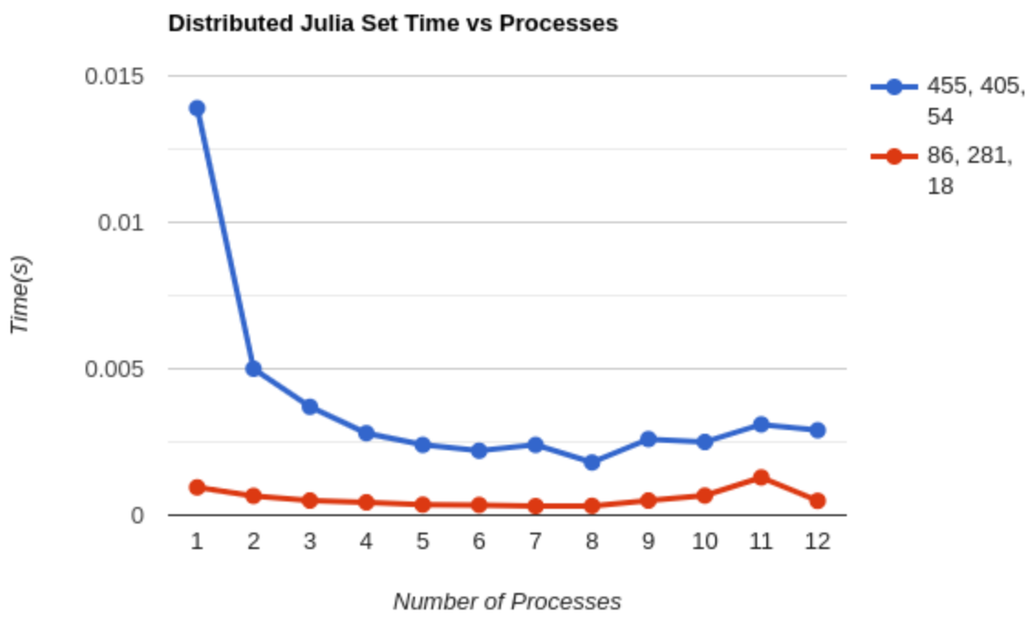
Message Complexity: $O(nm + p)$

This is due to each process sending it's part of the computation back to the main process.

Space Complexity:

1. **Per Process:** $O(nm/p)$
2. **Root Process:** $O(nm)$

This is for storing the results in an array format, hence every process stores it's chunked data, hence reduced by a factor of p, and root has entire data.



The graph shows the tuple (n,m,k) at the top-right for which the time plots were drawn.

At scale(blue-line), it's very evident that the parallelization is giving amazing results, we see a clear gain with $p > 1$ processes.

With a smaller test-case as well, there is a considerable gain, a slight increase is noticed towards the end but that is largely due the larger communications taking place(only $p = 11$, can be considered an outlier).

We see much better benchmarks for higher number of processes than $p = 1$.

Question - 3

Solution Approach

The approach taken in this question was to initially distribute the initial array across the different processes. Now, every process calculates it's own prefix sum concurrently. After this is done, we now need that every process having **rank r** needs to know the total prefix sums of all the processes from **process 1 through r-1,** to achieve this we use the **Send** and **Recv** functions provided by MPI. This helps us to synchronize the total sub-sums of the different processes and thus compute the correct answers for the particular array chunk. Finally, I broadcast the chunks back to the **root** process where they are combined and output by the program.

For this consider the following **notation**:

n ⇒ Number of elements in the array

p ⇒ The number of processes

Key Highlights

Key highlights include efficient data distribution across different processes. Along with this, the communication to send the partial sums between processes has been handled in a way to minimize the total message complexity. The program uses **MPI_Send** and **MPI_Recv** to propagate the accumulated sum from one process to the next.

Asymptotics

Computational Time Complexity: $O(n/p + p)$

Every process does it's distributed computation and hence an improvement by a factor of p, an additional $O(p)$ is incurred due to each row having to access at most $p - 1$ previous prefix sums.

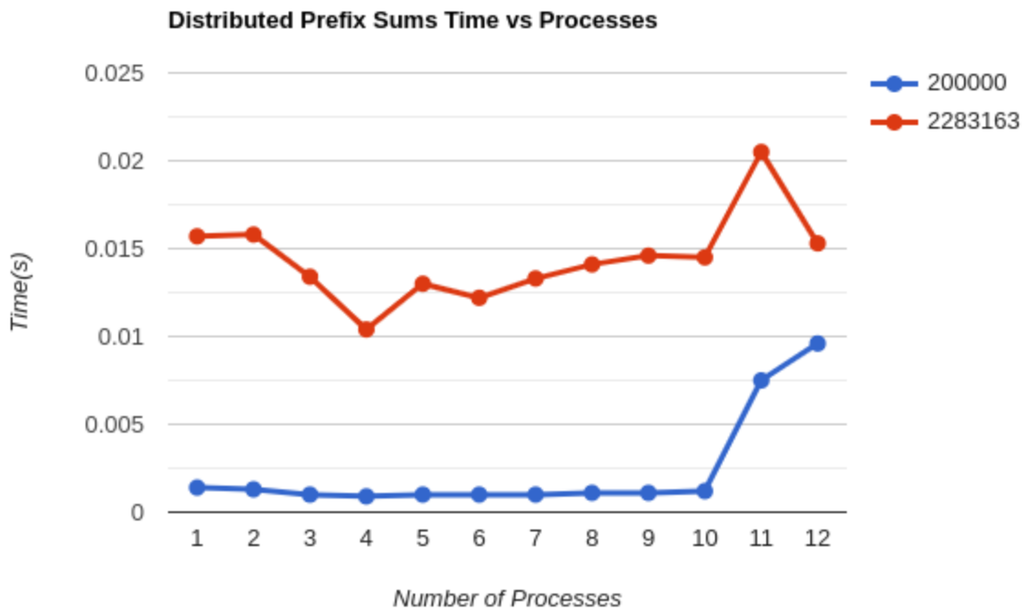
Message Complexity: $O(n)$

This is due to scattering of the initial array and gathering of the final array.

Space Complexity:

1. **Per Process:** $O(n/p + p)$
2. **Root Process:** $O(n + p)$

Space complexity at every process for storing it's data chunk, at the root process the complexity is due to storage of entire initial and final arrays.



The graph shows the value of n at the top-right for which the time plots were drawn.

We see that with increase in N, we start to observe the benefits of parallelization. We see a slight increase at higher processes, and this is mostly due to the end synchronization overhead of all the processes, where they are waiting for **Send** and **Recv** calls to complete before the final computation. It's still well below the brute-force benchmark of $p = 1$.

Question - 4

Solution Approach

The main approach was **Gauss-Jordan Elimination** using RREF. The matrices mat and identity I are initialized on all processes. The matrix mat is then populated with input values by the root process (rank 0). For each pivot row i, the process with rank i % size (where size is the total number of processes) is responsible for normalizing that row, i.e., making the diagonal element 1 by dividing the entire row by the pivot element. Every pivot is assigned to a process which can be inherently thought of a simulator for that particular pivot row. Each process handles the elimination of elements below the pivot row in parallel. Specifically, if the row index j is such that j%size==rank, then that process performs the elimination for row j using the broadcasted pivot row. Now, we have an upper triangular matrix

After this, the same idea is applied for the backward elimination (making the matrix upper triangular and then diagonal). Again, the rows are broadcasted after being updated, and the processes work in parallel to eliminate the elements above the pivots.

Key Highlights

The program divides the work of row operations across multiple processes, enabling simultaneous updates to different parts of the matrix. Each process is

responsible for handling specific rows based on its rank, effectively parallelizing the matrix inversion process. For each step in the algorithm, the process responsible for the current pivot row performs the operations and broadcasts the results to other processes. If a pivot element is zero (which would prevent division), the program checks subsequent rows for a suitable pivot. If found, it swaps rows and continues the elimination, ensuring the algorithm can proceed even in cases of numerical instability. This ensures fault tolerance in the program as well.

Asymptotics

For this consider the following **notation**:

n ⇒ Dimension of the 2-D array

p ⇒ The number of processes

Computational Time Complexity: $O(n^3/p)$

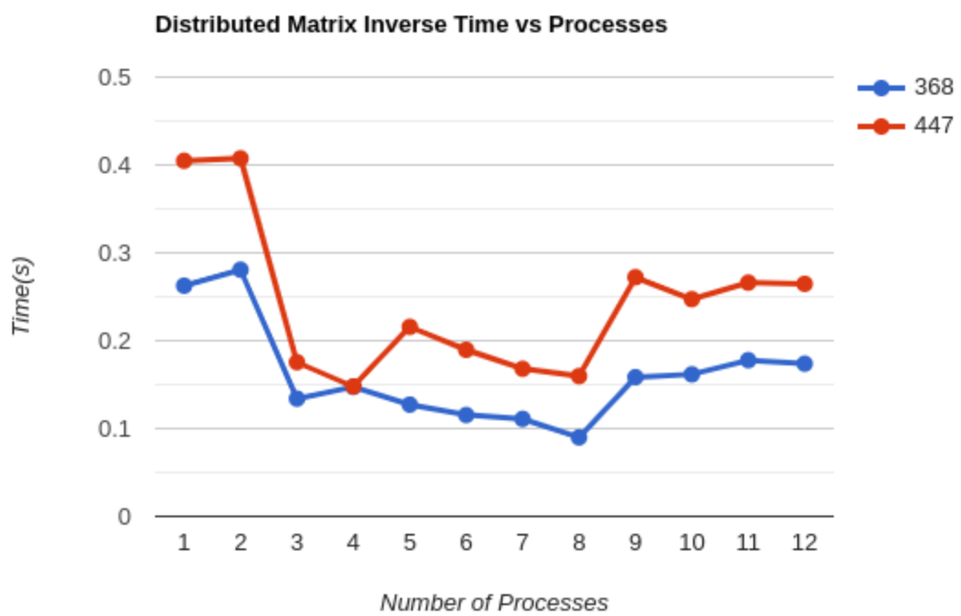
Each row is processed once, and each operation on a row involves O(n) work for each row. Since rows are distributed among p processes, the time complexity is improved by a factor of p.

Message Complexity: $O(n^2 logp)$

Each row is broadcasted n times, leading to a message complexity of O(n^2logp)

Space Complexity: $O(n^2)$ per process

This is due to the **mat**-array and **identity**-array being maintained at every process.



The graph shows the value of n at the top-right for which the time plots were drawn.

We see that the distribution of matrix inverse mostly works well. The slight increase at $p = 2$ is probably due to increased broadcasts required. At higher number of processes as well, a slight increase in communication overhead is

causing the time taken to increase. It's still well below the brute-force benchmark of $p = 1$.

Question - 5

Solution Approach

The main loop iterates over lengths of matrix chains, represented by slider_len. For each length, the possible matrix chains are divided among the processes. This is optimal because for any higher slider_len value we will need solutions to the smaller values of slider_len. The matrix chains are divided into blocks, and each process works on a specific block of chains determined by its rank. The block size and any remainder are calculated to ensure a balanced distribution of work among the processes. Each process computes the DP values for its assigned block of chains, updating the $dp[i][j]$ values using the recurrence relation as mentioned in the code. For every slider_len value, we gather all the $dp[i][j]$ values which were updated in this iteration across the processes using **MP_AlllGatherv** method.

Key Highlights

The program uses **MPI_Allgatherv** to efficiently gather the partial results computed by each process and distribute them among all processes. This ensures that all processes have the necessary data to continue with the next step of the computation. The matrix chain computations are divided into blocks, and the program carefully distributes these blocks among processes to ensure a balanced workload. This minimizes idle time and maximizes the use of computational resources. The program is designed to scale with the number of processes. As the number of processes increases, the workload is further divided, reducing the time taken for the computation, provided that communication overhead remains manageable.

Asymptotics

For this consider the following **notation**:

n ⇒ Dimension of the 2-D array

p ⇒ The number of processes

Computational Time Complexity: $O(n^3/p)$

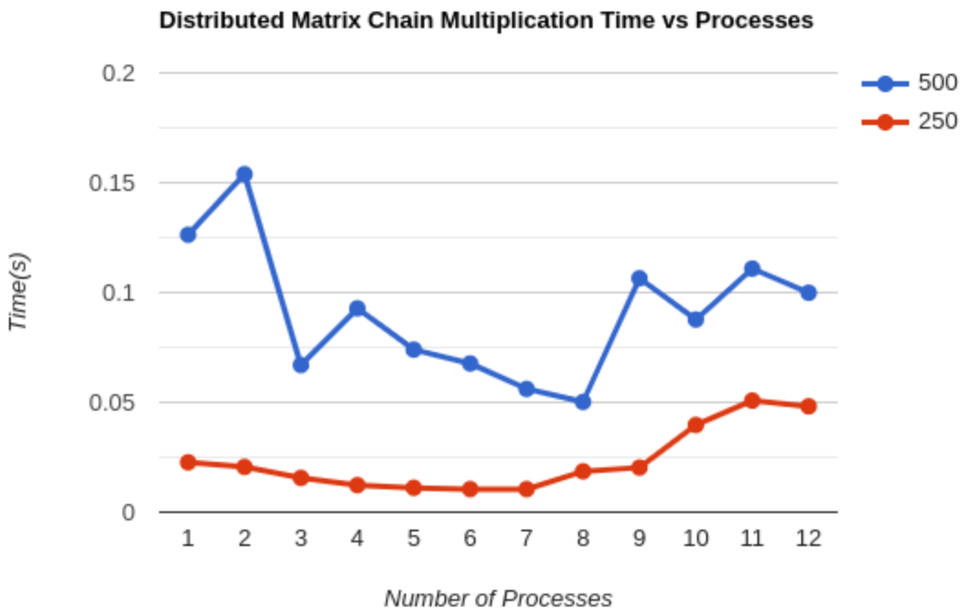
The sequential MCM DP algorithm has a time complexity of O(n^3), where n is the number of matrices. The factor of p comes by parallelizing the i - loop in the code.

Message Complexity: $O(n^2 logp)$

This is because we use **AllGatherv** method for broadcasting the messages for all dp[i][j]'s.

Space Complexity: $O(n^2)$ per process

This is due to the dp-array being maintained at every process.



The graph shows the value of n at the top-right for which the time plots were drawn.

When the problem is split between two processes, there is an initial overhead related to communication, task division, and synchronization between processes. This overhead can outweigh the benefits of parallel execution, especially when there are only two processes. Post this, we see a significant decrease in the computational time till $p = 8$. After this, we see a slight increase in time due to message complexity overhead. However, as seen with this, we can conclude that at scale the distribution is working perfectly.