CS633 Assignment Group Number 32

Pallav Goyal (220747)
Abhishek Khandelwal (220040)
Poojal Katiyar (220770)
Ansh Agarwal (220165)

April 2025

Contents

1 Introduction

In this assignment, we address the problem of efficiently analyzing time series data defined over a 3D spatial volume (nx × ny × nz) for each time step. Specifically, we aim to compute the number of local minima, the number of local maxima, the global minimum, and the global maximum at every time step. Given the large size of the data and the need for fast computation, we implement a parallel solution using the Message Passing Interface (MPI).

To parallelize the computation, we adopt a 3D domain decomposition strategy, where the spatial volume is partitioned among a 3D Cartesian grid of MPI processes. This allows each process to handle a sub-block of the data and perform computations locally, leading to improved scalability and performance. To ensure accurate detection of extrema at subdomain boundaries, halo (ghost cell) exchange between neighboring processes is implemented.

Our solution is designed to be scalable, memory-efficient, and numerically accurate, making it well-suited for processing large 3D scientific datasets in parallel environments.

2 Code Architecture

The system follows a multi-layer architecture:

2.1 Data Decomposition

- Global domain of size $NX \times NY \times NZ$ divided into $PX \times PY \times PZ$ subdomains
- Each process gets local sub-volume of size $LX \times LY \times LZ$ where:

$$\begin{aligned} LX &= NX/PX \\ LY &= NY/PY \\ LZ &= NZ/PZ \end{aligned}$$

- Each data-point of rank *myrank* gets its local sub-volume rank in the following way :

$$\begin{aligned} \text{local\_x\_coordinate} &= \frac{\text{myrank}}{P_Y \cdot P_Z}, \\ \text{local\_y\_coordinate} &= \left(\frac{\text{myrank}}{P_Z}\right) \bmod P_Y, \\ \text{local\_z\_coordinate} &= \text{myrank} \bmod P_Z. \end{aligned}$$

- Global-to-local mapping:

$$\begin{aligned} \text{global\_x\_start} &= \text{px\_idx} \times LX, \\ \text{global\_y\_start} &= \text{py\_idx} \times LY, \\ \text{global\_z\_start} &= \text{pz\_idx} \times LZ \end{aligned}$$

3 Code Description

The implementation follows these steps:

```
1. MPI_Init and domain decomposition
2. Create MPI derived datatypes for halo exchange
3. Use Parallel IO to read input data in batches
4. Exchange boundary data with neighbors
5. Detect local extrema
6. Reduce global results
```

3.1 I/O Batch Size Optimization Function

3.1.1 Function Definition

```
int get_time(int nx, int ny, int nz, int nc) {
    unsigned long long x = 1024*1024*1024; x *= 10;
    x /= (nx*ny*nz);
    if(x > (unsigned long long)nc) return nc;
    else return (int)x;
}
```

3.1.2 Purpose

This function calculates an optimal batch size for I/O operations when processing 3D volumetric data with multiple time steps. It balances:

- Memory constraints (10GB limit to be on the safe side)
- Data dimensions (nx × ny × nz grid per process)
- Total available time steps (nc)

3.1.3 Mathematical Operation

The core calculation follows this formula:

$$\text{batch\_size} = \min\left(\left\lfloor \frac{10 \times 2^{30}}{\text{nx} \times \text{ny} \times \text{nz}} \right\rfloor, \text{nc}\right) \tag{1}$$

Where:

- 2^{30} (1GB) × 10 = 10GB memory budget
- Denominator is total grid points per process
- Result is clamped to maximum available time steps (nc)

3.1.4 Parameters

3.1.5 Behavior Examples

- **Small Grid**: For 64×64×64 grid:

$$\frac{\text{10GB}}{262,144} \approx 40,960 \text{ time steps} \tag{2}$$

Parameter	Description
nx, ny, nz	Spatial dimensions of 3D grid of the subdomain
nc	Total number of time channels/steps

Table 1: Function parameters

- **Large Grid**: For 1024×1024×1024 grid:

$$\frac{10\text{GB}}{1,073,741,824} \approx 10 \text{ time steps} \tag{3}$$

- **Upper Bound**: Returns nc if calculation exceeds available time steps

3.1.6 Typical Usage

The returned value determines how many time steps can be processed simultaneously while staying within the 10GB memory budget:

```
int batch_size = get_time(nx, ny, nz, nc);
// Use batch_size to control I/O read operations
```

3.2 HaloExchange Setup

3.2.1 Neighbor Process Calculation in 3D Domain Decomposition

For each process in the 3D Cartesian grid, we identify its six face neighbors (two in each dimension) to enable halo exchange. The neighbors are calculated based on the process's coordinates in the decomposition grid:

```
int x_low = (process_x_coordinate > 0) ? myrank - (py * pz) : MPI_PROC_NULL;
int x_high = (process_x_coordinate < px - 1) ? myrank + (py * pz) : MPI_PROC_NULL;
int y_low = (process_y_coordinate > 0) ? myrank - pz : MPI_PROC_NULL;
int y_high = (process_y_coordinate < py - 1) ? myrank + pz : MPI_PROC_NULL;
int z_low = (process_z_coordinate > 0) ? myrank - 1 : MPI_PROC_NULL;
int z_high = (process_z_coordinate < pz - 1) ? myrank + 1 : MPI_PROC_NULL;
```

3.2.2 Explanation of Neighbor Calculation

- **X-Direction Neighbors**:
  - `x_low`: Process with lower x-coordinate (left neighbor)
    - Calculated as `myrank - (py * pz)`
    - Set to `MPI_PROC_NULL` if at minimum x-boundary (`process_x_coordinate == 0`)
  - `x_high`: Process with higher x-coordinate (right neighbor)
    - Calculated as `myrank + (py * pz)`
    - Set to `MPI_PROC_NULL` if at maximum x-boundary (`process_x_coordinate == px-1`)
- **Y-Direction Neighbors**:

  - `y_low`: Process with lower y-coordinate (front neighbor)
    - Calculated as `myrank - pz`
    - Set to `MPI_PROC_NULL` if at minimum y-boundary (`process_y_coordinate == 0`)
  - `y_high`: Process with higher y-coordinate (back neighbor)
    - Calculated as `myrank + pz`
    - Set to `MPI_PROC_NULL` if at maximum y-boundary (`process_y_coordinate == py-1`)
- **Z-Direction Neighbors**:
  - `z_low`: Process with lower z-coordinate (bottom neighbor)
    - Calculated as `myrank - 1`
    - Set to `MPI_PROC_NULL` if at minimum z-boundary (`process_z_coordinate == 0`)
  - `z_high`: Process with higher z-coordinate (top neighbor)
    - Calculated as `myrank + 1`
    - Set to `MPI_PROC_NULL` if at maximum z-boundary (`process_z_coordinate == pz-1`)

3.2.3 Key Points

- `MPI_PROC_NULL` serves as a null process indicator for boundary conditions
- The rank calculation accounts for the row-major ordering of processes:
  - X-direction stride: `py * pz` (full yz-plane of processes)
  - Y-direction stride: `pz` (z-column of processes)
  - Z-direction stride: 1 (adjacent processes)
- This neighbor identification enables efficient halo exchange for distributed 3D computations

3.3 Memory Allocation for 4D Volumetric Data

3.3.1 Data Structure Design

The code implements a hybrid memory allocation strategy combining:

- A **contiguous 1D buffer** for efficient memory access and MPI communication
- A **multi-dimensional pointer structure** for intuitive indexing

3.4 Core Components

3.4.1 Linear Buffer Allocation

```
float *lin_data2 = malloc(sizeof(float) * (lx + 2) * (ly + 2) * (lz + 2) * nc);
```

- **Purpose**: Single allocation for all volumetric data including ghost cells
- **Dimensions**:
  - Spatial: $(lx + 2) \times (ly + 2) \times (lz + 2)$ (local domain + 1 ghost cell per side)

- Temporal: nc time steps/channels

- **Memory**: Contiguous block of $(lx+2)(ly+2)(lz+2)nc$ floats

Multi-dimensional Pointer Structure

```
float ****local_data = (float ****)malloc((lx + 2) * sizeof(float ***));
for (int x = 0; x < lx + 2; x++) {
    local_data[x] = (float ***)malloc((ly + 2) * sizeof(float **));
    for (int y = 0; y < ly + 2; y++) {
        local_data[x][y] = (float **)malloc((lz + 2) * sizeof(float *));
        for (int z = 0; z < lz + 2; z++) {
            int index = x*(ly+2)*(lz+2)*nc + y*(lz+2)*nc + z*nc;
            local_data[x][y][z] = &lin_data2[index];
        }
    }
}
```

3.4.2 Key Features

The memory allocation strategy provides several important advantages:

Feature	Implementation	Benefit
Ghost Cells	Additional +2 in all spatial dimensions	Enables halo region communication between MPI processes
Memory Layout	Single contiguous allocation via `lin_data2`	Improves cache locality and MPI communication performance
Access Pattern	4D pointer structure `local_data`	Provides intuitive `[x][y][z][t]` array access syntax
Index Mapping	Row-major calculation with explicit strides	Ensures correct memory access patterns

Table 2: Memory allocation characteristics and benefits

The key aspects of this implementation are:

- **Hybrid Approach**: Combines the performance benefits of contiguous memory with the convenience of multi-dimensional indexing

- **MPI Optimization**: Ghost cells are allocated within the same contiguous block for efficient halo exchanges

- **Type Safety**: Maintains proper float* pointers at the lowest level for direct data access

3.4.3 Index Calculation

The linear index for position (x, y, z) follows row-major ordering:

$$\text{index} = x \times (ly + 2)(lz + 2)nc + y \times (lz + 2)nc + z \times nc \tag{4}$$

Where:

- x, y, z are spatial indices (including ghost cells)
- nc is the number of time channels
- Strides account for ghost cells in all dimensions

3.5 Neighbor Communication Pattern

The code implements a 3D halo exchange using non-blocking MPI calls to communicate boundary data between neighboring processes in each dimension:

```
// X-direction communication
if (x_high != MPI_PROC_NULL) {
    MPI_Isend(&local_data[lx][1][1][0], 1, plane_YZ, x_high, 0,
    MPI_COMM_WORLD, &send_request_x[1]);
    MPI_Irecv(&local_data[lx+1][1][1][0], 1, plane_YZ, x_high, 1,
    MPI_COMM_WORLD, &recv_request_x[1]);
}
```

3.5.1 Key Components

Direction-Specific Communication

- **X-Direction**:
  - Sends right boundary (x=lx) to x_high neighbor
  - Receives ghost layer (x=lx+1) from x_high
  - Uses `plane_YZ` datatype created using `MPI_Type_create_subarray` for YZ-plane communication.
- **Y-Direction**:
  - Sends back boundary (y=ly) to y_high neighbor
  - Receives ghost layer (y=ly+1) from y_high
  - Uses `plane_XZ` datatype created using `MPI_Type_create_subarray` for XZ-plane communication.
- **Z-Direction**:
  - Sends top boundary (z=lz) to z_high neighbor
  - Receives ghost layer (z=lz+1) from z_high
  - Uses `plane_XY` datatype created using `MPI_Type_create_subarray` for XY-plane communication.

Direction	Send Position	Recv Position	Datatype
X-high	[lx][1..ly][1..lz]	[lx+1][1..ly][1..lz]	plane_YZ
X-low	[1][1..ly][1..lz]	[0][1..ly][1..lz]	plane_YZ
Y-high	[1..lx][ly][1..lz]	[1..lx][ly+1][1..lz]	plane_XZ
Y-low	[1..lx][1][1..lz]	[1..lx][0][1..lz]	plane_XZ
Z-high	[1..lx][1..ly][lz]	[1..lx][1..ly][lz+1]	plane_XY
Z-low	[1..lx][1..ly][1]	[1..lx][1..ly][0]	plane_XY

Table 3: Halo exchange communication pattern

3.5.2 Communication Pattern

3.5.3 Synchronization

After initiating all non-blocking communications, the code waits for completion:

```
if (x_high != MPI_PROC_NULL) {
    MPI_Wait(&recv_request_x[1], MPI_STATUS_IGNORE);
    MPI_Wait(&send_request_x[1], MPI_STATUS_IGNORE);
}
```

3.5.4 Optimization Features

- **Non-blocking Calls**: Overlaps communication with computation
- **Boundary Checks**: Skips communication for edge processes (MPI_PROC_NULL)
- **Derived Datatypes**: Uses predefined planes for efficient strided communication
- **Bidirectional Exchange**: Each neighbor pair exchanges data simultaneously

3.6 Algorithm for Extrema Detection

3.6.1 Core Implementation

```
for (int c = 0; c < k; c++) {
    // Initialize with first element
    sub_global_maxima[c+time] = sub_global_minima[c+time] = local_data[1][1][1][c];
    local_minima_count[c+time] = local_maxima_count[c+time] = 0;

    // Scan entire local domain (excluding ghost cells)
    for (int x = 1; x < lx + 1; x++) {
        for (int y = 1; y < ly + 1; y++) {
            for (int z = 1; z < lz + 1; z++) {
                bool mini = true, maxi = true;

                // Update global extrema candidates
                sub_global_maxima[c+time] = max(sub_global_maxima[c+time],
                                                local_data[x][y][z][c]);
                sub_global_minima[c+time] = min(sub_global_minima[c+time],
                                                local_data[x][y][z][c]);
```

3.6.2 Neighborhood Checking

```
// Check all 6 neighbors
if (x > 1 || (x_low != MPI_PROC_NULL)) {
    if (local_data[x-1][y][z][c] >= local_data[x][y][z][c])
        maxi = false;
    if (local_data[x-1][y][z][c] <= local_data[x][y][z][c])
        mini = false;
}
// Similar checks for x+1, y 1 , z 1
// ... (abbreviated for space)

if (maxi) local_maxima_count[c+time]++;
if (mini) local_minima_count[c+time]++;
```

3.6.3 Key Components

Component	Description
Domain Iteration	Triply-nested loops over local domain (1..lx, 1..ly, 1..lz)
Boundary Handling	Checks `MPI_PROC_NULL` for physical boundaries
6-Point Stencil	Compares each data-point with face-adjacent neighbors
Extrema Tracking	Maintains running counts and global min/max candidates

Table 4: Components of Extrema detection algorithm

3.6.4 Optimization Features

- **Boundary Awareness**: Uses MPI neighbor info to correctly handle domain edges
- **Space Efficiency**: Single pass through data for both min/max detection
- **Parallel Ready**: Designed for later reduction across processes
- **Cache Friendly**: Linear memory access pattern within loops

3.7 MPI Reduction Operations

3.7.1 Memory Allocation

```
// Allocate reduction buffers
long long int *local_minima_total = malloc(sizeof(long long int) * nc);
long long int *local_maxima_total = malloc(sizeof(long long int) * nc);
float *global_minima = malloc(sizeof(float) * nc);
float *global_maxima = malloc(sizeof(float) * nc);
```

3.7.2 Global Extrema Reduction

```
MPI_Reduce(local_maxima_count, local_maxima_total, nc,
           MPI_LONG_LONG_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(local_minima_count, local_minima_total, nc,
           MPI_LONG_LONG_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(sub_global_maxima, global_maxima, nc,
           MPI_FLOAT, MPI_MAX, 0, MPI_COMM_WORLD);
MPI_Reduce(sub_global_minima, global_minima, nc,
           MPI_FLOAT, MPI_MIN, 0, MPI_COMM_WORLD);
```

3.7.3 Timing Statistics Reduction

```
MPI_Reduce(&t_total, &t_total_max, 1,
           MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
MPI_Reduce(&t_comm, &t_comm_max, 1,
           MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
MPI_Reduce(&t_compute, &t_compute_max, 1,
           MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

3.7.4 Reduction Operations Summary

Data	**Operation**	**Datatype**	**Root**
Local maxima count	SUM	MPI_LONG_LONG_INT	0
Local minima count	SUM	MPI_LONG_LONG_INT	0
Global maxima	MAX	MPI_FLOAT	0
Global minima	MIN	MPI_FLOAT	0
Total time	MAX	MPI_DOUBLE	0
Comm time	MAX	MPI_DOUBLE	0
Compute time	MAX	MPI_DOUBLE	0

Table 5: MPI reduction operations summary

3.7.5 Key Characteristics

- **Collective Operation**: All processes must participate
- **Root Process**: Only rank 0 receives the final results
- **Two-Stage Reduction**:
  - Local extrema counts are summed across processes
  - Global extrema values are compared (min/max)
- **Timing Analysis**: Maximum values across processes capture worst-case performance

4 Code Compilation and Execution

4.1 Compilation

```
mpicc -o src src.c
```

4.2 Execution

```
mpirun -np <PX*PY*PZ> ./src <data_file> <PX> <PY> <PZ>
           <NX> <NY> <NZ> <NC> <output_file>
```

4.3 Example

For a 64^3 grid with 3 timesteps on 8 processes:

```
mpirun -np 8 ./src data_64_64_64_3_bin.txt 2 2 2 64 64 64 3 output_64_64_64_3_8.txt
```

5 Code Optimizations

5.1 Key Optimizations

- **Parallel I/O**: Each MPI process reads only its assigned sub-volume of the data using `MPI_File_read_at`, allowing efficient distributed loading. This eliminated the need for central coordination and scaled well with the number of processes.
- **Adaptive Batch Reading**: Instead of reading all time steps at once, each process adaptively determines the optimal number of time stamps to read at a time based on the local subdomain size. This was calculated to maximize memory usage without exceeding the working memory constraint.
- **Derived Datatypes**: MPI subarrays were defined to represent faces of 3D subdomains (`plane_XY`, `plane_XZ`, `plane_YZ`), reducing halo exchange overhead and simplifying communication logic.
- **Non-blocking Communication**: Halo data exchange was implemented using `MPI_Isend` and `MPI_Irecv`, allowing communication to overlap with computation on interior grid points. This reduced overall communication latency and improved concurrency.

6 Results

6.1 Scaling Performance for Optimized Code:

- To provide a statistically sound representation of the system's performance, each configuration was executed five times. The results from these runs were then averaged to smooth out anomalies and transient system variations.

Timing Variable	Definition	What it Measures
`t_total`	Total execution time of the entire simulation loop	Full loop duration including I/O and compute
`t_read`	Total time spent on reading data from the file using MPI I/O	I/O time due to `MPI_File_read_at` operations
`t_main`	Total time spent on computation and communication after reading data	Processing time (computation + communication)

Table 6: Description of MPI Timing Variables

- We present the average execution times across various values of `np` for the datasets `data_64_64_64_3_bin.txt` and `data_64_64_96_7_bin.txt`, as shown below.

Table 7: Timings for the `data_64_64_64_3_bin.txt`

Processes	Read Time	Main Code Time	Total Time
8	0.0094108	0.0082630	0.0169462
16	0.0140786	0.0046908	0.0181484
32	0.0156940	0.0031602	0.0181090
64	0.0316980	0.0052790	0.0329334

Table 8: Timings for the `data_64_64_96_7_bin.txt`

Processes	Read Time	Main Code Time	Total Time
8	0.0198205	0.0275463	0.0451513
16	0.0225020	0.0142244	0.0353216
32	0.0244424	0.0083746	0.0316374
64	0.0491926	0.0079434	0.0530390

- For the timings run on input file `data_64_64_64_3_bin.txt`, we observe the following performance trends:

  The **Total Time** initially increases very slowly, remaining almost constant for a lower number of processes. However, as the number of processes (`np`) increases further, the total time begins to increase more significantly. This is due to the growing overhead from communication and I/O contention that starts to outweigh the benefits of parallelization as the data size is not that large so parallelization benefits are not seen.

  The **Main Code Time** decreases with increasing `np`, as the computational workload is divided among more processes. However, beyond a certain point, the main code time shows a slight increase. This is due to increasing communication overhead, particularly during halo exchange operations. As seen in the MPI code snippet, a 6-point stencil communication pattern is used (`MPI_Isend`/`MPI_Irecv` in all six directions).With more processes, each process exchanges smaller messages more frequently. Smaller messages suffer from higher latency-to-bandwidth ratio, making communication less efficient. As the number of processes grows, more messages are injected into the network simultaneously, leading to contention.

The **Read Time** increases consistently with the number of processes. While each process reads a smaller portion of the file, the overall read time increases due to intensified I/O contention and synchronization overhead. When multiple processes simultaneously access the same file, especially on shared filesystems, they compete for disk bandwidth and file system locks. This leads to queuing delays, increased latency, and degraded throughput. Moreover, small scattered reads by many processes can prevent effective prefetching and caching by the operating system, further amplifying the overhead.

- The graph for input data_64_64_64_3_bin.txt is shown below:

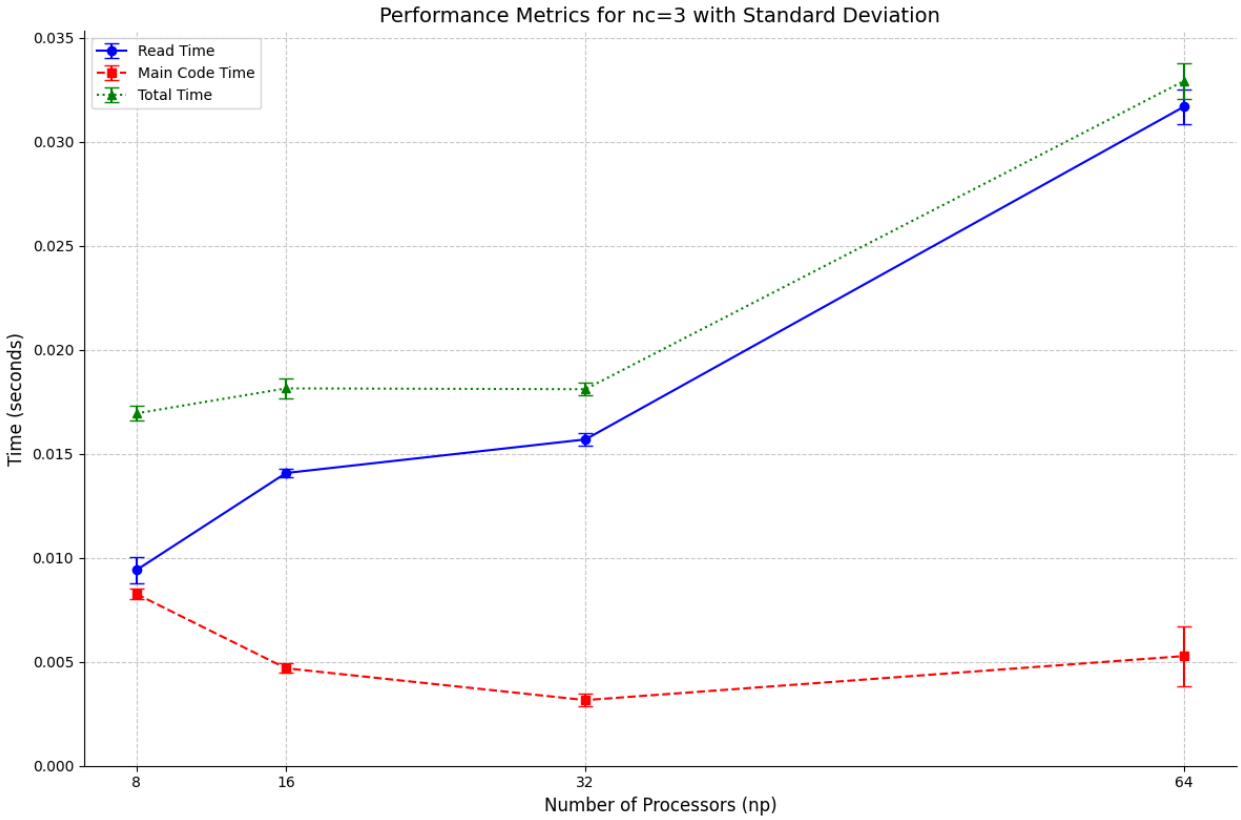

Figure 1: Time vs Number of Processes

- For the timings run on the input file `data_64_64_96_7_bin.txt`, we observe the following performance trends:

  Firstly, since the data size is larger compared to the previous input, the overall execution time is generally higher.

  The **Read Time** increases consistently with the number of processes. This is due to the same reason as before. Given the larger data size, this effect becomes more pronounced, leading to higher read times.

  The **Main Code Time** shows a clear decrease as the number of processes increases. This indicates effective parallelization, as the increased computational load due to the larger data size benefits from being distributed across more processes. The performance gain from par-

allelizing computation outweighs the communication overhead introduced by `MPI_Isend` and `MPI_Irecv`. However, as `np` increases, the reduction in main code time becomes less significant, suggesting that communication overhead is gradually becoming more dominant.

The **Total Time** decreases as the number of processors increases up to `np = 32`, and then increases again. This indicates that with a larger data size, parallelization yields more benefits. For each input file, there exists an optimum number of processes that minimizes the sum of **Read Time** and **Main Code Time**. As increasing `np` leads to higher read time and lower main code time, identifying this optimal point helps in efficiently optimizing overall performance.

For example, for the input file `data_64_64_64_3_bin.txt`, `np = 8` is optimal among the given four process counts. In contrast, for `data_64_64_96_7_bin.txt`, `np = 32` is optimal.

- The graph for input data_64_64_64_7_bin.txt is shown below:

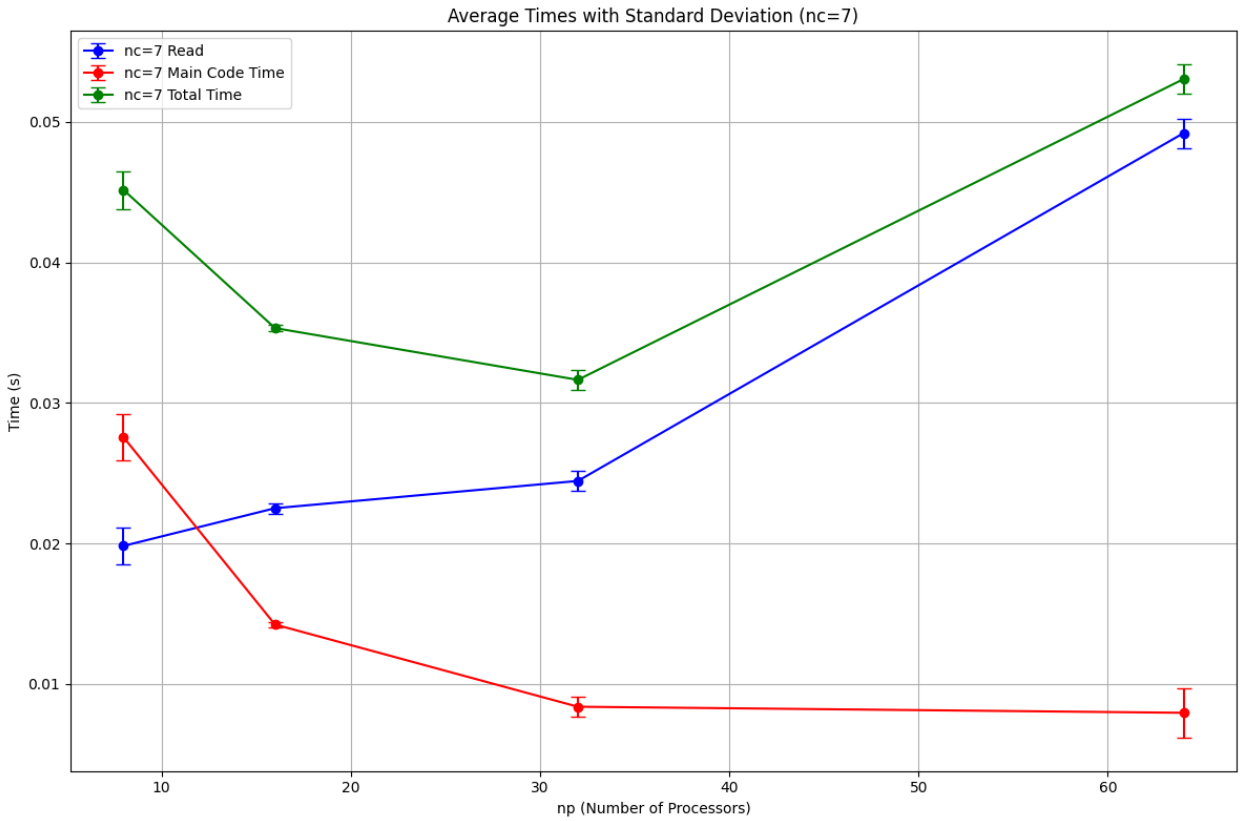

Figure 2: Time vs Number of Processes

6.2 Unoptimized code done using Sequential I/O

- Initially when we started working on the assignment we implemented various data distribution strategies:
  - **Single `MPI_Send/MPI_Recv` for each data point:** Simple but inefficient, as each data point requires a separate communication operation, leading to high overhead.

- **Using `MPI_Pack` and `MPI_Unpack`:** Reduces communication overhead by packing multiple data points into a single message, making it more efficient than sending individual data points.
- **Using `MPI_Scatter`:** Optimized for large datasets by distributing chunks of data to processes in a single operation, minimizing communication overhead and simplifying code.

- Although we followed the best data distribution strategy but still it was inefficient as we were doing sequential I/O (meaning reading from 1 process and then distributing to others).
- So the unoptimized code that we will be comparing is where data is read sequentially at rank 0 and distributed using `MPI_Scatter`.
- The rank 0 process computes each data point's target process using block-wise decomposition and fills a flattened `send_buffer` accordingly.
- Subarrays are created to describe each process's portion of the global data. These subarrays enable efficient scattering using `MPI_Scatter`.
- The plots for unoptimized code for input files data_64_64_64_3 .txt and data_64_64_96_7 .txt are:

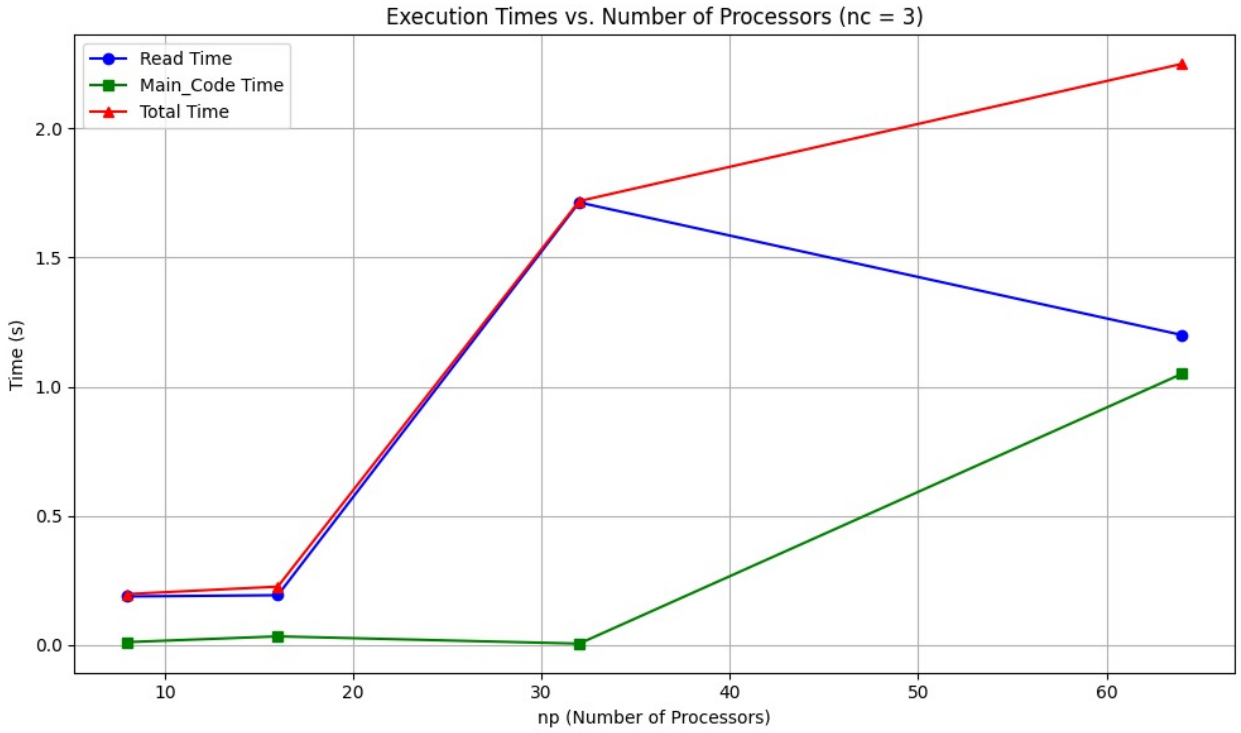

Figure 3: Average Total time vs Number of Processes(unoptimized)

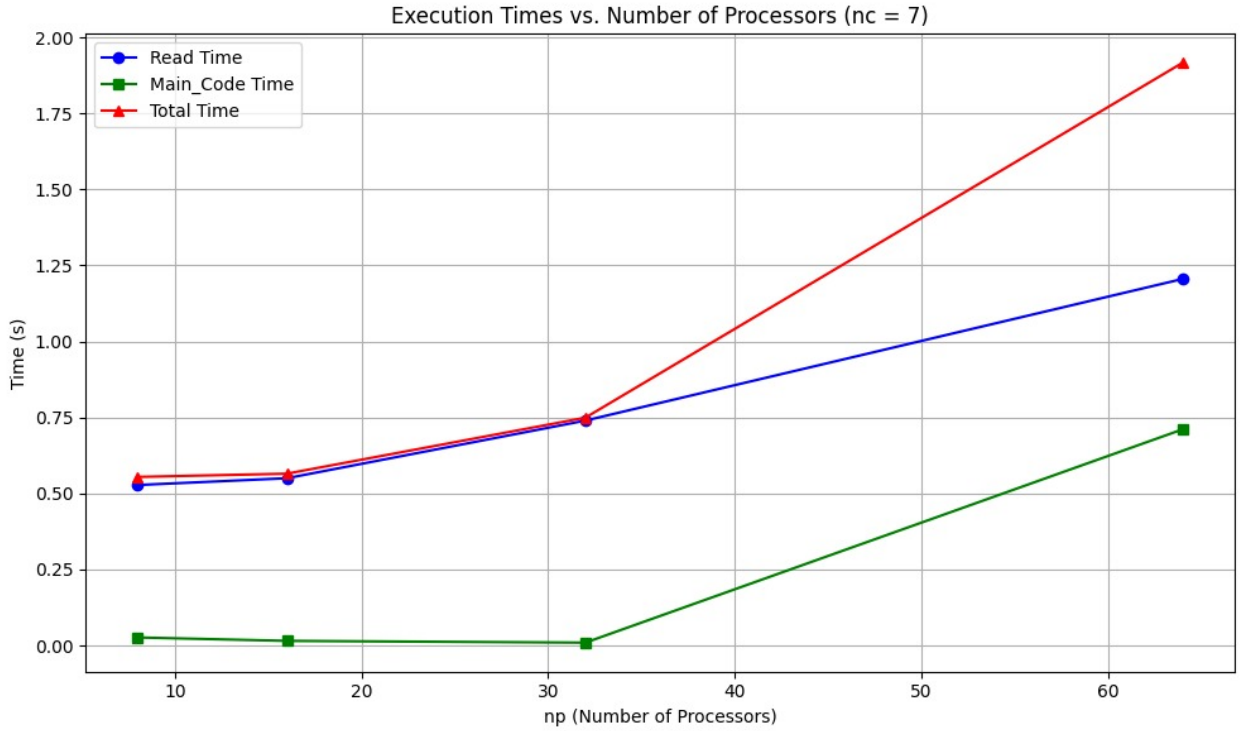

Figure 4: Average Total time vs Number of Processes(unoptimized)

- From the figure above it is quite evident how the Read time is of an order 10-50 times more as compared to when reading using Parallel I/O

Table 9: Comparision between Timings for the `data_64_64_64_3_bin.txt` for Read for unoptimised and optimised code (in seconds)

Processes	Unoptimised Code Timing	Optimised Code Time
8	0.187912	0.0094108
16	0.192609	0.0140786
32	1.713589	0.0156940
64	1.200247	0.0316980

Table 10: Comparision between Timings for the `data_64_64_96_7_bin.txt` for Read for unoptimised and optimised code (in seconds)

Processes	Unoptimised Code Timing	Optimised Code Time
8	0.527858	0.0198205
16	0.550168	0.0225020
32	0.739710	0.0244424
64	1.205197	0.0491926

6.3 Code Efficiency improvements implemented:

- **Memory Constraints:** Given the constraints, we ensured that our code is optimized to handle the worst-case scenario efficiently. To achieve this, the code has been adjusted to process the number of timestamps according to the provided values, ensuring memory usage remains within the limits. The logic for managing this can be observed in the get_time function(whose detailing is provided in section 3.1.1), which efficiently handles the timestamps without exceeding the memory capacity.

- **Sequential I/O Optimized Code:** Even when using sequential I/O, we employed various data distribution methods, such as Pack/Unpack followed by send/recv and MPI_Scatter. We implemented the optimal data distribution strategy with MPI_Scatter to further enhance the code's efficiency.

- **Parallel I/O (Most Optimized Code):** As shown in section 6.1 and 6.2, how the use of Parallel I/O considerably improves the efficiency of code by reducing the read time considerably.

7 Conclusions

7.1 Team Contributions

- **Pallav**: Worked on implementing efficient MPI communication and halo exchange. Defined MPI subarray data types for the XY, XZ, and YZ planes using `MPI_Type_create_subarray` and `MPI_Type_commit`, enabling non-contiguous memory access for halo regions. Contributed to optimizing the parallel I/O to reduce communication overhead during data processing.

- **Abhishek**: Focused on optimizing parallel file I/O and explored various methods for sending and receiving halo regions, including the evaluation of collective and point-to-point communication strategies. Played a key role in debugging critical sections of the code to ensure correctness and improve performance.

- **Poojal**: Investigated data distribution methods such as `MPI_Scatter` and manual buffer packing during the use of sequential I/O. Implemented the core logic for computing local minima and maxima. Performed runtime analysis to compare the impact of different I/O and communication strategies on performance.

- **Ansh**: Carried out detailed performance analysis and scaling tests. Conducted strong and weak scaling experiments across different problem sizes and processor counts. Evaluated speedup and efficiency metrics to guide further optimizations and validate improvements in the parallel implementation.

All members actively contributed to different parts of the project. Through extensive trial and error, collaborative debugging, and iterative optimization—especially transitioning from sequential I/O to parallel I/O and implementing MPI-based halo exchange—the team successfully developed an optimized parallel solution for computing the assignment.

Submission Details

The submission consists of the following components:

- **Output Files:** Named in the format
  `output_<nx>_<ny>_<nz>_<nc>_<px*py*pz>_<iter>.txt`
  where each placeholder represents the corresponding simulation parameter.
- **Job Script:** A SLURM job script named `job.sh` that requests 2 nodes. The number of tasks per node is set such that the total number of processes is $\frac{np}{2}$. Naming convention in the zipped folder for various configurations is job_<nc>_<np>.
- **Source Code:** Contained in a single file named `src.c`.
- **Report:** The final report is submitted as `Group32.pdf`.
- **Plot Scripts:** Python scripts used to generate plots included in the report.