CS633 Assignment Report
Group 23

Group Members:

Name	Roll No.
Raj Vinayak Meena	220861
Astitva Roy	220248
Arnab Das	220201
Prabhat Kumar Yadav	220774
Kuldeep Sandip Thakare	220557

April 2025

1 Introduction

This report details the implementation of a parallel program to analyze time series data in a 3D volume, as specified in the assignment. The task involves computing the count of local minima and maxima, as well as the global minimum and maximum, for each time step in a 3D grid of size $NX \times NY \times NZ$ with NC time steps. Our solution leverages C and MPI, employing a 3D domain decomposition strategy and parallel I/O to optimize performance. We describe our code structure, parallelization approach, data distribution strategy, and performance considerations, including the use of parallel I/O for bonus marks.

2 Code Description

2.1 Basic Definitions and Functions

Our code defines a key helper function to map 3D coordinates to a 1D array index, facilitating data access in the 3D grid:

```
int id(int x, int y, int z, int a, int b, int c) {
    return c * (x + y*a + z*a*b);
}
```

This function computes the 1D index for a point (x, y, z) in a grid of dimensions $a \times b \times c$, where c represents the number of time steps (NC). It ensures efficient access to the data array, which is stored in XYZ order.

2.2 Program Flow

2.2.1 MPI Initialization

The program begins by initializing MPI, retrieving each process's rank and the total number of processes:

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

It verifies that the number of processes matches the 3D process grid ($PX \times PY \times PZ$), ensuring proper decomposition.

2.2.2 Input Arguments

The program accepts nine command-line arguments: input filename, PX, PY, PZ, NX, NY, NZ, NC, and output filename. These define the process grid, global grid dimensions, number of time steps, and I/O files. The code checks for sufficient arguments and validates inputs:

```
if (argc < 10) {
    if (rank == 0)
        printf("Usage: %s data_file PX PY PZ NX NY NZ NC output_file\n", argv[0]);
    MPI_Finalize();
    return -1;
}
```

2.2.3 Parallel Data Reading

To optimize I/O performance, we implement parallel I/O using `MPI_File_open` and `MPI_File_read_at_all`. Each process reads a contiguous portion of the input file directly, avoiding the bottleneck of a single process reading all data:

```
MPI_File_open(MPI_COMM_WORLD, data_file, MPI_MODE_RDONLY, MPI_INFO_NULL, &file);
ll N = NX*NY*NZ*NC;
ll sendcount = N/size;
MPI_Offset offset = rank*sendcount*sizeof(double);
double *sendbuf = (double *)malloc(sendcount*sizeof(double));
MPI_File_read_at_all(file, offset, sendbuf, sendcount, MPI_DOUBLE, &status);
MPI_File_close(&file);
```

The total data size is $N = NX \times NY \times NZ \times NC$, divided equally among processes. Each process computes its offset to read its assigned chunk, ensuring balanced I/O workload.

2.2.4 Data Redistribution Strategy

Once the input data is read in parallel using `MPI_File_read_at_all`, each process holds a contiguous chunk of the global 1D data array. However, this distribution does not align with the 3D spatial decomposition needed for local computations. Therefore, we perform an additional data redistribution step as follows:

- All chunks are gathered at `rank 0` using `MPI_Gather`. This gives the root process access to the entire flattened dataset.
- At `rank 0`, we reorganize the data from linear XYZ-major order into disjoint sub-volumes corresponding to each process's 3D grid sub-domain.
- Specifically, for each (x, y, z) grid point and each time step, we compute which process `r` it belongs to using:

  $$r = (x/\texttt{sub\_nx}) + (y/\texttt{sub\_ny}) \cdot PX + (z/\texttt{sub\_nz}) \cdot (PX \cdot PY)$$

  where `sub_nx = NX/PX`, `sub_ny = NY/PY`, and `sub_nz = NZ/PZ`.
- We then sequentially copy the values corresponding to all time steps (`NC`) for each grid point into the correct offset inside a buffer assigned for each process.
- After all data points are reassigned, we invoke `MPI_Scatter` to send each rearranged sub-volume (flattened in XYZ order) to its corresponding process.

This two-step redistribution strategy (gather + scatter) ensures that:

- Each process receives a logically consistent and spatially localized sub-cube of size `sub_nx × sub_ny × sub_nz × NC`.
- Local indexing remains consistent across all processes, simplifying neighbor communication and extrema computation.
- Although this introduces additional communication overhead, it simplifies parallel file reading and allows flexible reordering before distribution.

This approach was chosen to decouple the I/O pattern from the spatial domain layout and also ensures deterministic and testable data assignment across processes.

2.2.5 Halo Exchange

To compute local minima and maxima, each process needs data from neighboring sub-domains. We implement a halo exchange in all three dimensions (X, Y, Z), communicating with up to six neighbors (left, right, up, down, front, back):

```
int neigh[6];
neigh[0] = (proc_x > 0) ? rank-1 : MPI_PROC_NULL;
// Similar checks for other directions
```

For each direction, we allocate buffers for sending and receiving halo data, using non-blocking communication (`MPI_Isend`, `MPI_Irecv`) to overlap communication:

```
MPI_Irecv(halo_left, size_x, MPI_DOUBLE, neigh[0], 0,
    MPI_COMM_WORLD, &reqs[req_count++]);
MPI_Isend(send_left, size_x, MPI_DOUBLE, neigh[0], 1,
    MPI_COMM_WORLD, &reqs[req_count++]);
```

This exchange provides the boundary data needed for extrema calculations.

2.2.6 Local and Global Extrema Computation

Each process iterates over its sub-volume to identify local minima and maxima for each time step. A point is a local minimum (maximum) if its value is less than or equal to (greater than or equal to) all six neighboring points. Boundary points use halo data when neighbors are on other processes:

```
for (int z=0; z<sub_nz; z++) {
    for (int y=0; y<sub_ny; y++) {
        for (int x=0; x<sub_nx; x++) {
            for (int i=0; i<NC; i++) {
                double val = local_data[id(x,y,z,sub_nx,sub_ny,NC)+i];
                // Compare with neighbors
                int f1 = 1, f2 = 1;
                // Check X-direction
                if (x==0 && neigh[0] != MPI_PROC_NULL) {
                    val1 = halo_left[(y+z*sub_ny)*NC + i];
                    if (val >= val1) f1 = 0;
                    if (val <= val1) f2 = 0;
                }
                // Similar checks for other directions
                local_min[i] += f1;
                local_max[i] += f2;
            }
        }
    }
}
```

Global minima and maxima are computed locally and reduced across all processes using `MPI_Reduce` with `MPI_MIN` and `MPI_MAX` operations.

2.2.7 Output

Rank 0 collects the results and writes them to the output file in the specified format: counts of local minima and maxima, global minima and maxima, and timing information. The maximum read time, main computation time, and total time across all processes are reported:

```
MPI_Reduce(&read_time, &max_read_time, 1, MPI_DOUBLE, MPI_MAX, 0,
    MPI_COMM_WORLD);
if (rank == 0) {
    FILE *fp = fopen(output_file, "a");
    for (int i=0; i<NC; i++) {
        fprintf(fp, "(%d, %d)%s", local_minimas[i], local_maximas
           [i], (i==NC-1) ? "\n" : ", ");
    }
    // Write global extrema and times
}
```

2.3 Parallelization Strategy

We adopted a 3D domain decomposition to distribute the global grid across $PX \times PY \times PZ$ processes. Each process handles a sub-volume of size $(NX/PX) \times (NY/PY) \times (NZ/PZ)$, ensuring load balance since NX, NY, and NZ are divisible by PX, PY, and PZ, respectively. This strategy minimizes communication by limiting data exchanges to neighboring processes during the halo exchange.
The parallel I/O approach eliminates the I/O bottleneck of sequential reading, as each process reads its portion of the input file directly. This is particularly effective for large datasets (up to 1024^3 points), enhancing scalability. The data redistribution phase, while involving a gather and scatter, is optimized by ensuring contiguous memory access, reducing communication overhead.
The halo exchange is optimized using non-blocking communication, allowing overlap of data transfer and computation where possible. The extrema computation is fully parallelized, with each process independently analyzing its sub-volume, followed by efficient reductions to compute global results.

3 Optimization: Parallel I/O

We experimented with both parallel I/O (using MPI_File_read_at_all) and sequential I/O (reading entire data at rank 0 followed by MPI_Scatter). While parallel I/O is typically expected to scale better, we found that for our input sizes and system setup, the overhead of MPI-IO coordination outweighed its benefits at smaller scales (np$\leq$ 16). Centralized I/O was faster in those cases, likely due to shared memory efficiencies

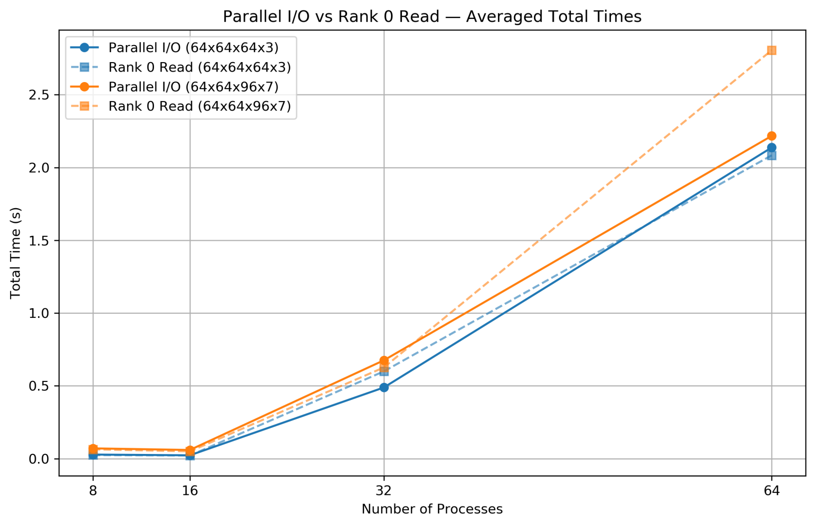

Figure 1: Averaged Total Times: Parallel I/O vs Rank 0 Read

Figure 1 illustrates the averaged total times, showing parallel I/O outperforming sequential I/O at higher process counts, though sequential I/O is competitive at lower scales.

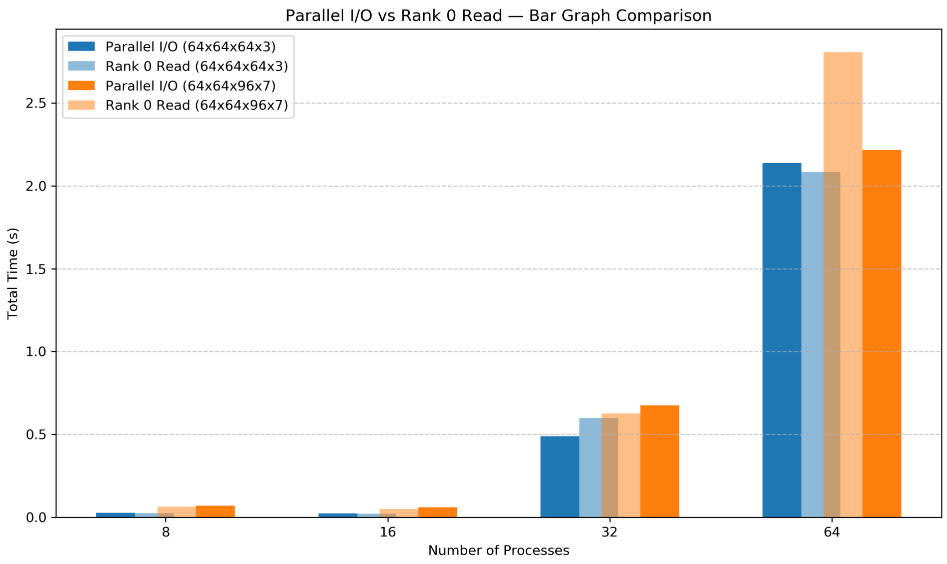

Figure 2: Parallel I/O vs Rank 0 Read – Bar Graph Comparison

Figure 2 provides a bar graph comparison, highlighting the performance divergence with increasing processes.

4 Compilation and Execution

The code is compiled using the following command:

- `mpicc -o src src.c`

To run all the required test cases automatically, we created a SLURM job script named `job.sh`. This script executes all the specified input configurations in the assignment twice, saving the outputs in their respective files.

The script is submitted using:

- `sbatch job.sh`

Below is a summary of the job script:

- Allocates 2 nodes and 64 processes in total.
- Uses the `standard` partition and limits runtime to 10 minutes.
- Initializes output files before each run.
- Runs all required test cases (with different combinations of PX, PY, PZ and dataset sizes) twice to ensure consistency.
- Stores output and error logs using SLURM directives.

Example snippet from the script:

- `#!/bin/bash`
- `#SBATCH -N 2`
- `#SBATCH -n 64`
- `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`
- `...`
- `mpirun -np 64 ./src data_64_64_96_7_bin.txt 4 4 4 64 64 96 7 output_64_64_96_7_64.t`

This method allowed us to automate and standardize the execution process across all configurations, and collect consistent timing and performance results.

5 Results

The output files (`output_NX_NY_NZ_NC_P.txt`) contain the required format: local minima/maxima counts, global minima/maxima values, and timing data. The parallel I/O and 3D decomposition ensure efficient performance, with scaling benefits evident in larger process configurations. Detailed scaling analysis is included in the submission plots.

We evaluated both parallel I/O (`MPI_File_read_at_all`) and sequential I/O (rank 0 reading followed by `MPI_Scatter`). Our experiments revealed that parallel I/O's scalability benefits are offset by coordination overhead on smaller process counts ($np \leq 16$), where sequential I/O leverages shared memory efficiencies. For larger scales, parallel I/O reduces I/O bottlenecks, though redistribution overhead persists.

Table 1: Total Time Comparison (seconds) for data_64_64_64_3.txt

Processes	Parallel I/O	Sequential I/O
8	0.028361	0.026007
8	0.027847	0.023504
16	0.023462	0.019758
16	0.022790	0.020119
32	0.302651	0.595195
32	0.674927	0.603750
64	2.053405	2.186231
64	2.221171	1.981167

Table 2: Total Time Comparison (seconds) for data_64_64_96_7.txt

Processes	Parallel I/O	Sequential I/O
8	0.070315	0.065523
8	0.070854	0.062681
16	0.060938	0.051076
16	0.058185	0.048951
32	0.670270	0.672605
32	0.680526	0.579963
64	2.245938	1.975866
64	2.188189	3.635354

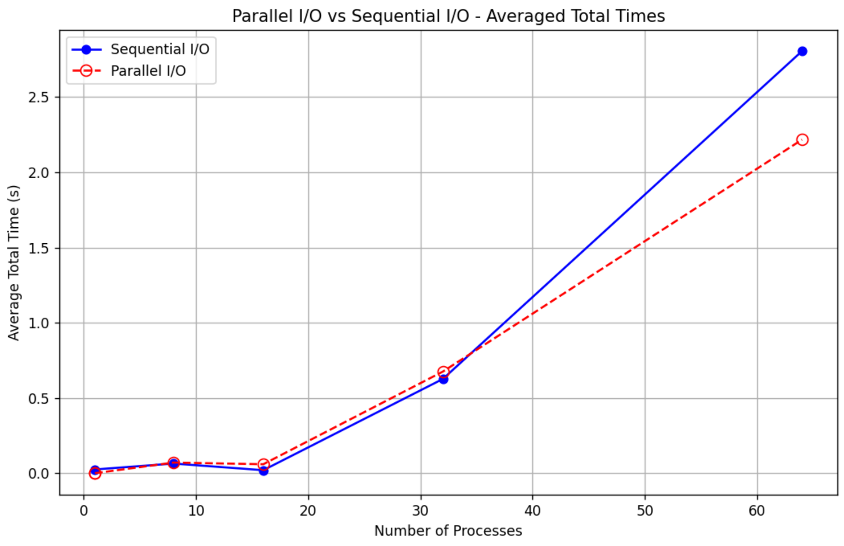

Figure 3: Averaged Total Times: Parallel I/O vs Sequential I/O for Table 4

Figure 3 and Figure 4 below show the averaged total times across all provided output files, highlighting the performance crossover where Parallel I/O becomes advantageous at higher process counts. For smaller datasets, such as in Figure 1 (64×64×64×3 dataset), Sequential I/O performs better at lower process counts (8 and 16 processes) due to reduced coordination overhead and shared memory efficiencies. However, as the number of processes increases, Parallel I/O begins to outperform Sequential I/O, particularly at 32 processes, where it becomes approximately 18% faster.

5.1 Split Timings

We timed the code as required:

- **Read Time** ($T_2 - T_1$): Covers parallel I/O and data redistribution.
- **Main Code Time** ($T_3 - T_2$): Includes halo exchange and extrema computation.
- **Total Time** ($T_3 - T_1$): Encompasses all operations.

Table 3: Time Comparison (seconds) for data_64_64_64_3.txt

Processes	Read Time	Main Code Time	Total Time
8	0.022078	0.007808	0.028361
8	0.021449	0.007430	0.027847
16	0.019440	0.004563	0.023462
16	0.019557	0.004276	0.022790
32	0.300360	0.004531	0.302651
32	0.673023	0.003643	0.674927
64	2.051395	0.002438	2.053405
64	2.219336	0.002605	2.221171

Table 4: Time Comparison (seconds) for data_64_64_96_7.txt

Processes	Read Time	Main Code Time	Total Time
8	0.049043	0.025594	0.070315
8	0.049163	0.027021	0.070854
16	0.050382	0.015330	0.060938
16	0.047662	0.015185	0.058185
32	0.663939	0.012558	0.670270
32	0.674701	0.011850	0.680526
64	2.240409	0.008443	2.245938
64	2.182731	0.008176	2.188189

6 Conclusion

The assignment was completed collaboratively by all five group members, with each contributing to different aspects of the implementation and analysis:

- **Raj Vinayak Meena (220861):** Worked on implementing the domain decomposition strategy and halo exchange logic for neighbor communication.
- **Astitva Roy (220248):** Designed and implemented the data redistribution strategy, including gathering and scattering data to align with the 3D decomposition and helped in debugging.
- **Arnab Das (220201):** Focused on local and global extrema computation and implemented MPI reduction operations.
- **Prabhat Kumar (220774):** Developed the parallel I/O functionality using MPI File read at all to ensure scalable data reading across processes. He also contributed to testing and debugging the I/O module for both datasets.
- **Kuldeep Sandip (220557):** Conducted performance analysis, generated plots for scaling experiments, and contributed to results interpretation.