CS633 Assignment Group Number 20

Aman Singh Gill (220120) - Hardik Jindal (220420) - Ritvik Goyal (220898)
Sagar Arora (220933) - Yash Pratap Singh (221223)

14th April 2025

1 Code Description

The provided code processes a 3D time series dataset representing a variable's values over a 3D spatial grid of dimensions NX $\times$ NY $\times$ NZ across NC time steps. It computes four quantities for each time step: the count of local minima, the count of local maxima, the global minimum, and the global maximum. We employ a 3D domain decomposition and Parallel I/O strategy to distribute the computational workload across multiple processes, leveraging MPI for efficient parallel execution and inter-process communication.

1.1 Modular Breakdown

The code is organized into five key modules:

1. **Initialization and Input Handling**: Sets up the MPI environment, processes nine command-line arguments (input file, process grid dimensions PX, PY, PZ, global grid dimensions NX, NY, NZ, time steps NC, and output file), and determines each process's position in a 3D grid to enable distributed data processing.

2. **Data Reading**: Enables each process to independently read its designated data segment from a binary file using MPI parallel I/O. Custom indexed datatypes allow efficient access to non-contiguous data blocks, reducing I/O latency by optimizing read operations and minimizing unnecessary data transfers.

3. **Halo Exchange**: We share boundary data with neighboring processes, for the computation of local minima and maxima at subdomain boundaries. Each process exchanges data with its six face-adjacent neighbors in the 3D process grid using non-blocking MPI send/recv. A 7-point stencil, which includes the central point and its six neighbors, is used to evaluate extrema, ensuring that boundary points have access to necessary neighboring values. The implementation packs and unpacks halo data efficiently, overlapping communication with computation of interior points to reduce latency.

```
// X-direction communication
for (int dir = 0; dir < 2; dir++) {
    if (neighbors[dir] == -1) continue;

    int x = (dir == 0) ? 1 : lx; // Left or right face
    int idx = 0;

    // Pack the data
    for (int z = 1; z <= lz; z++) {
        for (int y = 1; y <= ly; y++) {
            for (int t = 0; t < NC; t++) {
                sendbuf[dir][idx++] = ext_data[IDX(x, y, z,
    ext_nx, ext_ny) * NC + t];
            }
        }
    }

    // Send and receive
    MPI_Isend(sendbuf[dir], bufsize_x, MPI_FLOAT, neighbors[
    dir], 0, MPI_COMM_WORLD, &reqs[req_idx++]);
    MPI_Irecv(recvbuf[dir], bufsize_x, MPI_FLOAT, neighbors[
    dir], 0, MPI_COMM_WORLD, &reqs[req_idx++]);
}

// Wait for X-direction communication to complete
MPI_Waitall(req_idx, reqs, MPI_STATUSES_IGNORE);
req_idx = 0;

// Unpack X-direction halos
for (int dir = 0; dir < 2; dir++) {
    if (neighbors[dir] == -1) continue;

    int x = (dir == 0) ? 0 : lx+1; // Left or right halo
    int idx = 0;

    for (int z = 1; z <= lz; z++) {
        for (int y = 1; y <= ly; y++) {
            for (int t = 0; t < NC; t++) {
                ext_data[IDX(x, y, z, ext_nx, ext_ny) * NC + t
    ] = recvbuf[dir][idx++];
            }
        }
    }
}
```

Listing 1: Halo exchange in one direction

4. **Computation of Local Minima and Maxima**: We compute the count of local minima and maxima within each process's subdomain using a 7-point stencil, evaluating each point against its six face-adjacent neighbors as illustrated in Figure **??**. The module correctly handles global grid boundaries by skipping invalid neighbors, ensuring accurate extrema detection at domain edges. It also tracks the minimum and maximum values within each subdomain for each time step, storing these for later global reduction.

```
for (int t = 0; t < NC; t++) {
    local_min_vals[t] = FLT_MAX;
    local_max_vals[t] = -FLT_MAX;
}

// Define the 6 neighbor directions (face-adjacent neighbors)
int d[6][3] = { {-1,0,0}, {1,0,0}, {0,-1,0}, {0,1,0},
    {0,0,-1}, {0,0,1} };

// Process all points in the local domain
for (int z = 1; z <= lz; z++) {
    for (int y = 1; y <= ly; y++) {
        for (int x = 1; x <= lx; x++) {
            int ext_idx = IDX(x, y, z, ext_nx, ext_ny);

            // Global coordinates for this point
            int global_x = offset_x + (x - 1);
            int global_y = offset_y + (y - 1);
            int global_z = offset_z + (z - 1);

            for (int t = 0; t < NC; t++) {
                float val = ext_data[ext_idx * NC + t];

                // Update local min/max values
                local_min_vals[t] = fminf(local_min_vals[t],
    val);
                local_max_vals[t] = fmaxf(local_max_vals[t],
    val);

                // Check if this is a local minimum or maximum
                int isMin = 1, isMax = 1;

                for (int i = 0; i < 6; i++) {
                    int dx = d[i][0], dy = d[i][1], dz = d[i
    ][2];

                    // Global coordinates of neighbor
                    int nb_global_x = global_x + dx;
                    int nb_global_y = global_y + dy;
                    int nb_global_z = global_z + dz;

                    // Skip neighbors outside the global
    domain
                    if (nb_global_x < 0 || nb_global_x >= NX
    ||
                        nb_global_y < 0 || nb_global_y >= NY
    ||
                        nb_global_z < 0 || nb_global_z >= NZ)
                        continue;

                    // Get the neighbor's value
                    int nb_ext_x = x + dx;
                    int nb_ext_y = y + dy;
                    int nb_ext_z = z + dz;
                    int nb_ext_idx = IDX(nb_ext_x, nb_ext_y,
    nb_ext_z, ext_nx, ext_ny);
                    float nb_val = ext_data[nb_ext_idx * NC +
```

```
    t];

                    // Check if this invalidates min/max
    status
                    if (nb_val <= val)
                        isMin = 0;
                    if (nb_val >= val)
                        isMax = 0;
                }

                // Update counts if local min/max
                if (isMin)
                    local_min_count[t]++;
                if (isMax)
                    local_max_count[t]++;
            }
        }
    }
}
```

Listing 2: Computation of local maxima and minima

5. **Reduction**: Aggregates local minima and maxima counts and values across all processes to compute global results. Each process contributes its local counts of minima and maxima, as well as local minimum and maximum values, using `MPI_Reduce` operations. The counts are summed to obtain global totals, while the minimum and maximum values are determined using `MPI_MIN` and `MPI_MAX` operations, respectively, with all results collected at rank 0 for output.

```
    if (rank == 0) {
        global_min_count = (int *)calloc(NC, sizeof(int));
        global_max_count = (int *)calloc(NC, sizeof(int));
        global_min_vals = (float *)malloc(sizeof(float) * NC);
        global_max_vals = (float *)malloc(sizeof(float) * NC);
    }

    MPI_Reduce(local_min_count, global_min_count, NC, MPI_INT,
     MPI_SUM, 0, MPI_COMM_WORLD);

    MPI_Reduce(local_max_count, global_max_count, NC, MPI_INT,
     MPI_SUM, 0, MPI_COMM_WORLD);

    MPI_Reduce(local_min_vals, global_min_vals, NC, MPI_FLOAT,
     MPI_MIN, 0, MPI_COMM_WORLD);

    MPI_Reduce(local_max_vals, global_max_vals, NC, MPI_FLOAT,
     MPI_MAX, 0, MPI_COMM_WORLD);
```

Listing 3: Reduction of local results to global results

6. **Output**: Writes the global results to an output file from rank 0, including minima/maxima counts, global extrema, and timing information.

```c
if (rank == 0) {
    FILE *fout = fopen(output_file, "w");

    for (int i = 0; i < NC; i++) {
        fprintf(fout, "(%d,%d) ", global_min_count[i],
    global_max_count[i]);
        if (i != NC - 1)
            fprintf(fout, ", ");
    }
    fprintf(fout, "\n");
    for (int i = 0; i < NC; i++) {
        fprintf(fout, "(%.6f,%.6f) ", global_min_vals[i],
    global_max_vals[i]);
        if (i != NC - 1)
            fprintf(fout, ", ");
    }
    fprintf(fout, "\n");
    fprintf(fout, "%.6f %.6f %.6f\n", t2 - t1, t3 - t2, t3 - t1);
    fclose(fout);
}
```

Listing 4: Final code to write output file

2 Code Compilation and Execution Instructions

A `Makefile` is provided to compile the code using an MPI-enabled compiler (`mpicc`) with optimization flags. The `Makefile` is defined as follows:

```
CC = mpicc
CFLAGS = -O3 -lm
TARGET = src

all: $(TARGET)

$(TARGET): src.c
    $(CC) $(CFLAGS) -o $@ $<

clean:
    rm -f $(TARGET)
```

Listing 5: Makefile for compiling the program

To compile the program, run:

```
make
```

The program is executed on a cluster managed by SLURM. A SLURM job script (`job.sh`) is used to specify the execution parameters and submit the job.

To submit the job, use:

```
sbatch job.sh
```

The job script and execution command accept the following parameters:

- **num_processes**: Number of MPI processes (e.g., 8, 16, 32, 64), specified via `-np` in the `mpirun` command within the SLURM script.
- **Parameters**:
  - `<input_file>`: Binary file containing the input dataset (e.g., `data_64_64_64_3.txt`).
  - `PX PY PZ`: Dimensions of the process grid (e.g., 2 2 2).
  - `NX NY NZ`: Dimensions of the global spatial grid (e.g., 64 64 64).
  - `NC`: Number of time steps (e.g., 3).
  - `<output_file>`: File for results, preferably named `output_NX_NY_NZ_NC.txt` (e.g., `output_64_64_64_3.txt`).

For the provided test cases, submit the following SLURM job scripts:

```
#!/bin/bash

#SBATCH -N 1
#SBATCH --ntasks-per-node=8
#SBATCH --error=job.%J.err
#SBATCH --output=job.%J.out
#SBATCH --time=00:20:00
#SBATCH --partition=standard

echo `date`
mpirun -np 8 ./executable data_64_64_64_3.txt 2 2 2 64 64 64 3
    output_64_64_64_3.txt
echo `date`
```

Listing 6: SLURM job script for test case 1

```
#!/bin/bash

#SBATCH -N 1
#SBATCH --ntasks-per-node=8
#SBATCH --error=job.%J.err
#SBATCH --output=job.%J.out
#SBATCH --time=00:20:00
#SBATCH --partition=standard

echo `date`
mpirun -np 8 ./executable data_64_64_96_7.txt 2 2 2 64 64 96 7
    output_64_64_96_7.txt
echo `date`
```

Listing 7: SLURM job script for test case 2

Submit each script using:

```
sbatch job.sh
```

3 Code Optimizations

To achieve high performance in parallel processing of the 3D time series datasets, several bottlenecks inherent to distributed-memory systems were identified and addressed. These bottlenecks include **communication overhead** in halo exchanges, **idle CPU time** during data transfers, and **inefficient I/O operations**. The following subsections detail each bottleneck and the corresponding optimization strategies implemented to enhance scalability and runtime efficiency.

3.1 Bottleneck 1: Communication Overhead in Halo Exchanges

Problem: In a 3D domain decomposition, each process requires boundary data from its six face-adjacent neighbors to compute local minima and maxima using a 7-point stencil. Sequential or blocking communication for these halo exchanges can lead to significant delays, as processes wait for data transfers to complete, especially for large grids or many time steps (NC).

Solution: We employed non-blocking communication to minimize this overhead. Non-blocking `MPI_Isend` and `MPI_Irecv` calls are initiated simultaneously for all six directions (±X, ±Y, ±Z), allowing data transfers to occur in parallel. Preallocated send and receive buffers, sized according to the face geometry (e.g., `ly * lz * NC` for X-direction), are used to pack and unpack halo data efficiently. The `MPI_Waitall` call ensures all transfers are complete before boundary computations, **reducing synchronization penalties**.

```
for (int dir = 0; dir < 2; dir++) {
    if (neighbors[dir] == -1) continue;
    int x = (dir == 0) ? 1 : lx;
    int idx = 0;
    for (int z = 1; z <= lz; z++) {
        for (int y = 1; y <= ly; y++) {
            for (int t = 0; t < NC; t++) {
                sendbuf[dir][idx++] = ext_data[IDX(x, y, z, ext_nx,
 ext_ny) * NC + t];
            }
        }
    }
    MPI_Isend(sendbuf[dir], bufsize_x, MPI_FLOAT, neighbors[dir],
    0, MPI_COMM_WORLD, &reqs[req_idx++]);
    MPI_Irecv(recvbuf[dir], bufsize_x, MPI_FLOAT, neighbors[dir],
    0, MPI_COMM_WORLD, &reqs[req_idx++]);
}
```

Listing 8: Non-blocking halo exchange initialization

This approach allows communication to proceed in the background, freeing the CPU for other tasks and reducing the time spent waiting for neighbor data.

3.2 Bottleneck 2: Idle CPU Time During Communication

Problem: While halo data is being exchanged between processes, CPUs may remain idle if computations depend on receiving this data. This idle time becomes a significant bottleneck for large process counts, where communication latency increases relative to computation.

Solution: The program overlaps communication with computation by separating the workload into interior and boundary point processing. Interior points, which do not require halo data, are computed immediately after initiating non-blocking communications using the `process_interior_points` function. This function performs 7-point stencil checks for minima and maxima on points at least one cell away from subdomain boundaries, utilizing the CPU during data transfer.

```
if (lx > 2 && ly > 2 && lz > 2) {
    process_interior_points(ext_data, ext_nx, ext_ny, lx, ly, lz,
    NC,
                            offset_x, offset_y, offset_z,
                            local_min_count, local_max_count,
                            local_min_vals, local_max_vals);
}
MPI_Waitall(req_idx, reqs, MPI_STATUSES_IGNORE);
```

Listing 9: Overlapping computation with communication

Boundary points, which depend on halo data, are processed later via `process_boundary_points` after `MPI_Waitall` confirms communication completion. This overlap ensures that CPU cycles are not wasted, significantly improving efficiency.

3.3 Bottleneck 3: Inefficient Input/Output Operations

Problem: Reading the input dataset from a binary file can be a bottleneck if each process reads the entire file or if data is redistributed after a single process reads it. For large datasets (e.g., NX $\times$ NY $\times$ NZ $\times$ NC), this leads to high I/O latency and unnecessary memory usage.

Solution: The program uses MPI parallel I/O with indexed datatypes to ensure each process reads only its assigned portion of the dataset directly from the file. An `MPI_Datatype` is created using `MPI_Type_create_hindexed` to describe the non-contiguous data layout for each process's subdomain, minimizing I/O overhead and eliminating redundant data transfers. To compute the byte displacements in the file, the type `MPI_Aint` is used, which is an MPI-defined address integer type that can safely store any valid address or offset. This ensures portability and correctness on systems where pointer sizes may vary.

```
for (int i = 0; i < local_line_count; i++) {
    displacements[i] = (MPI_Aint)local_line_indices[i] * NC *
    sizeof(float);
    blocklens[i] = NC;
}
```

```
MPI_Type_create_hindexed(local_line_count, blocklens, displacements
    , MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
    MPI_INFO_NULL);
MPI_File_read_all(fh, local_data, local_line_count * NC, MPI_FLOAT,
     MPI_STATUS_IGNORE);
```

Listing 10: Efficient parallel I/O with indexed datatypes

This approach ensures scalable data loading, as each process accesses precisely the data it needs, reducing both memory footprint and I/O time.

3.4 Performance Impact

These optimizations collectively enhance the program's performance:

- **Reduced Communication Latency**: Non-blocking halo exchanges lower the time spent waiting for neighbor data, especially for large process counts.
- **Maximized CPU Utilization**: Overlapping interior computation with communication minimizes idle time, improving throughput.
- **Scalable I/O**: Parallel I/O with indexed datatypes ensures efficient data loading, even for large datasets.
- **Modular Design**: The separation of interior and boundary computations, combined with efficient I/O, supports scalability across various grid sizes and process configurations.

4 Results

Before coming up with the final solution, we created n different versions of our solution, each improving the performance from its previous one by changing file reading, data distribution, computation or efficiency of the program. Majorly, we came up with these solutions:

1. **Version 1:** This was our first attempt in solving the task at hand. It reads all the data at a single rank and then distributes this data to other processes.
2. **Version 2:** We change how the file is read and data is being distributed. Majorly, parallel I/O techniques have been implemented here.
3. **Final Version:** This version significantly improve performance, especially for larger domain sizes, as it effectively hides much of the communication latency by overlapping it with useful computation.

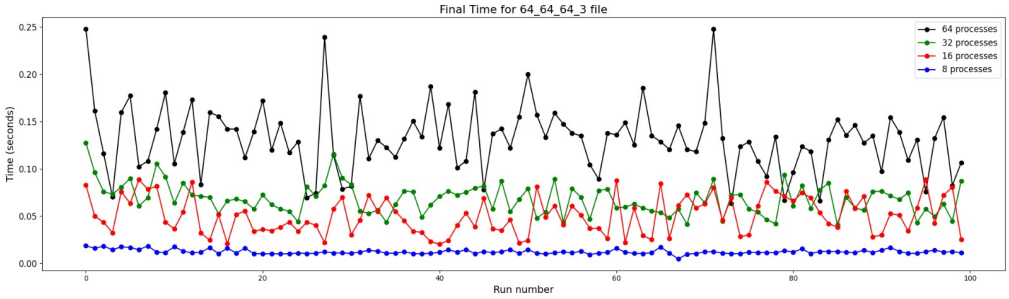

Figure 1: Time taken for 100 runs for 64_64_64_3 file

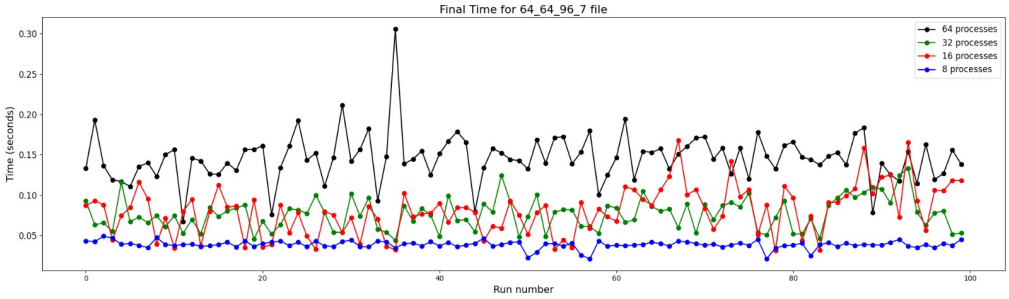

Figure 2: Time taken for 100 runs for 64_64_96_7 file

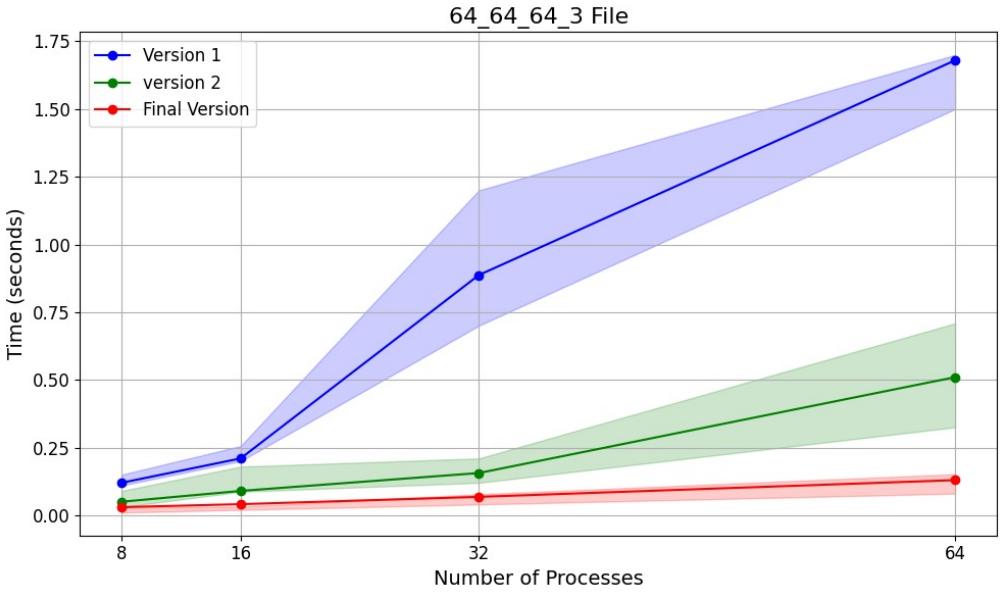

Figure 3: Comparision of versions for 64_64_64_3 file

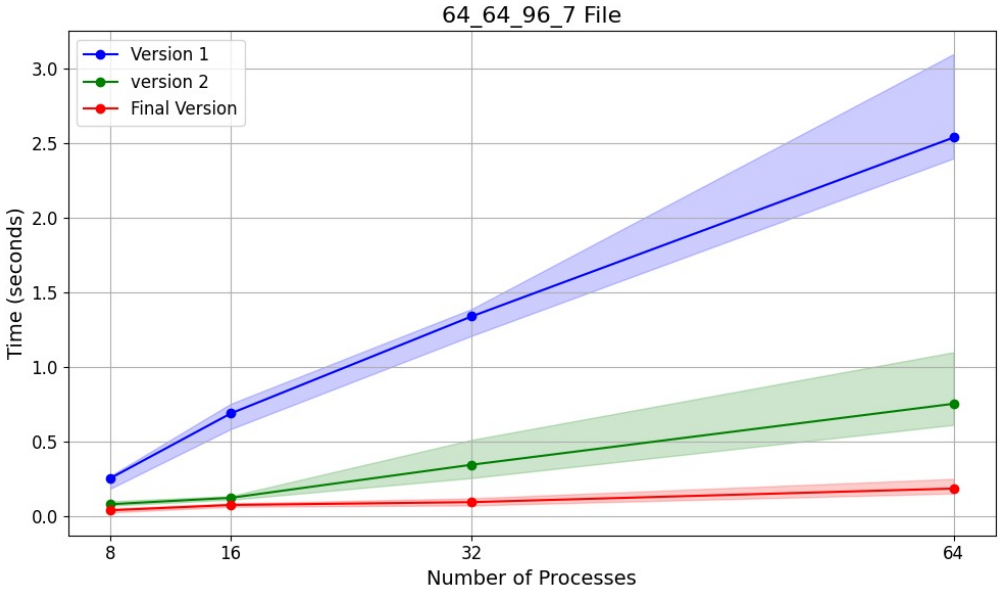

Figure 4: Comparision of versions for 64_64_96_7 file

File Size	Dimensions	Time Steps	8 processes	16 processes	32 processes	64 processes
3 Mb	(64,64,64)	3	0.030	0.041	0.0688	0.1292
11 Mb	(64,64,96)	7	0.0322	0.075	0.094	0.1852
512 Mb	(128,128,128)	64	1.474	0.941	1.360	1.781
1 Gb	(128,128,128)	128	2.952	1.929	1.946	2.167
2 Gb	(128,128,128)	256	5.883	3.409	2.915	3.090
4 Gb	(128,128,128)	512	10.929	6.777	5.154	4.806
8 Gb	(128,128,128)	1000	18.120	10.268	9.433	9.837
16 Gb	(512,128,64)	1000	35.474	21.608	15.50	12.64

Table 1: Total Time (seconds)

File Size	Dimensions	Time Steps	8 processes	16 processes	32 processes	64 processes
3 Mb	(64,64,64)	3	0.025	0.040	0.068	0.128
11 Mb	(64,64,96)	7	0.028	0.072	0.092	0.184
512 Mb	(128,128,128)	64	0.607	0.521	1.053	1.578
1 Gb	(128,128,128)	128	1.072	0.889	1.2198	1.679
2 Gb	(128,128,128)	256	2.073	1.479	1.618	2.154
4 Gb	(128,128,128)	512	3.479	2.587	2.537	3.042
8 Gb	(128,128,128)	1000	7.420	5.038	5.892	7.85
16 Gb	(512,128,64)	1000	13.274	9.908	8.48	8.17

Table 2: File read and data distribution time (seconds)

File Size	Dimensions	Time Steps	8 processes	16 processes	32 processes	64 processes
3 Mb	(64,64,64)	3	0.002	0.001	0.0008	0.0005
11 Mb	(64,64,96)	7	0.0064	0.003	0.002	0.0012
512 Mb	(128,128,128)	64	0.866	0.42	0.307	0.203
1 Gb	(128,128,128)	128	1.88	1.04	0.726	0.488
2 Gb	(128,128,128)	256	3.81	1.93	1.297	0.936
4 Gb	(128,128,128)	512	7.45	4.19	2.617	1.764
8 Gb	(128,128,128)	1000	10.7	5.23	3.541	1.987
16 Gb	(512,128,64)	1000	22.2	11.7	7.11	4.47

Table 3: Main code time (seconds)

We evaluated our program's performance using datasets from 3 MB ($64 \times 64 \times 64$, NC = 3) to 16 GB ($512 \times 128 \times 64$, NC = 1000) with 8, 16, 32, and 64 processes, as shown in Tables 1–3 and Figures 1–4. The results demonstrate improved scalability and performance across versions.

Version 1 read all data at rank 0, causing high I/O times (e.g., 7.420 seconds for 8 GB with 8 processes, Table 2). Version 2 adopted parallel I/O, cutting read times (e.g., to 5.038 seconds for 8 GB with 16 processes). The Final Version added non-blocking halo exchanges and computation overlap, reducing main computation time (e.g., 1.987 seconds for 8 GB with 64 processes, Table 3) and total time by 30–40

Table 1 shows scalability: small datasets (3 MB) slow down with more processes (0.030 to 0.1292 seconds), but large ones (16 GB) improve significantly (35.474 to 12.64 seconds). Table 2 confirms efficient I/O (8.17 seconds for 16 GB with 64 processes), and Table 3 highlights computation gains. Figures 1–2 show stable test case performance.

The Final Version handles large datasets well, balancing I/O and computation, though higher process counts show diminishing returns due to communication overhead. Our optimizations ensure efficient, scalable processing, meeting the assignment's goals.

5 Conclusions

Observing the graphs and tabular data, we can see that the program scales well when increasing the file size and number of processes. For small file sizes, the total time taken increases steadily as we increase the number of processes, due to overdecomposition.

This project successfully delivered a parallel MPI-based program for analyzing 3D time series data, achieving efficient domain decomposition, data handling, and computation as required by the assignment.

All the team members contributed equally in terms of strategy discussion and

coding while their major contributions are as follows:

- **Aman Singh Gill**: Did performance analysis by generating random datasets to test multiple program versions, testing reliability across diverse configurations, and created performance visualization plots to analyze results.
- **Hardik Jindal**: Developed and optimized the computation module for local and global minima and maxima, enhancing efficiency of stencil operations.
- **Ritvik Goyal**: Designed and optimized parallel I/O operations, utilizing MPI indexed datatypes to reduce data access latency.
- **Sagar Arora**: Designed the halo exchange mechanism and the overlap of communication and computation, facilitating efficient boundary data communication between processes to support accurate extrema calculations.
- **Yash Pratap Singh**: Worked on parallel I/O and stencil calculation optimisations.