CS633 Assignment Group 41

Lakshika (210554), Rohit Jangid (210872), Sharvani Jadhav (210960), Vanshika Gupta (211146)

April 13, 2025

1 Code Description

The submitted code is implemented in C language using MPI library and follows a modular approach for high-performance parallel computation. The primary goal is to compute the count of local minima, count of local maxima, global minima and global maxima in a 3D time-series dataset using 3D spatial decomposition. Below is modular breakdown of the code:

1.1 Initialization and Argument Parsing

The program starts by initializing MPI and reading command-line arguments:

```
<file> PX PY PZ NX NY NZ NC <output_file>
```

These specify the input file, process grid, data dimensions, time steps, and output file name.

1.2 3D Domain Decomposition

The 3D domain is divided among processes based on the provided values of PX, PY, and PZ. Each process determines its rank in X, Y, and Z directions using sub-communicators:

```
MPI_Comm_split(...);
```

The data is decomposed evenly assuming NX, NY, and NZ are divisible by PX, PY, and PZ respectively.

1.3 Parallel File Reading (Optimized I/O)

Each process reads its respective sub-block from the binary file using calculated offsets based on its location in the domain. This eliminates the need for a root process to perform all I/O and improves scalability:

```
fseek(fp, offset_in_floats * sizeof(float), SEEK_SET);
fread(buffer, sizeof(float), chunk_size, fp);
```

1.4 Halo Exchange Using Non-blocking Communication

To accurately compute local extrema near the boundaries, halo exchange is done using non-blocking sends and receives (‘MPI_Isend‘, ‘MPI_Irecv‘) with derived datatypes (‘MPI_Type_vector‘) to transfer faces of sub-domains.

1.5 Minima and Maxima Computation

Each rank loops through its local volume and checks values in all 6 directions (X±1, Y±1, Z±1) to determine local minima and maxima. Values of points of neighbouring process are recived via halo exchange and the minima maxima is computed for the corner cases with their help.

1.6 Global Reduction

Global extrema and total local counts are aggregated using collective operations using MPI_MIN, MPI_MAX for extrema values and MPI_SUM for sum of counts

```
MPI_Reduce(...);
```

1.7 Output Generation and Timing

The root process (rank 0) writes results in the expected format:

- Line 1: Local minima and maxima counts per timestep
- Line 2: Global minimum and maximum values per timestep
- Line 3: Read time, compute time, and total execution time

Timing is recorded using ‘MPI_Wtime()‘ at key checkpoints, and the maximum across ranks is reported.

1.8 Optimizations Implemented

- **Parallel I/O**: Each process reads its data chunk in parallel, reducing I/O bottleneck.
- **Non-blocking Communication**: Enables overlapping computation and communication during halo exchanges.
- **MPI Derived Datatypes**: Efficiently transfers faces of 3D data without manual packing/unpacking.

These algorithm choices contribute to strong scalability and efficient memory use, especially for larger datasets and higher core counts.

2 Code Compilation and Execution Instructions

Compile the code using the provided makefile or via the following command:

```
mpicc -o test test.c
```

Execute using the provided MPI command line format:

```
mpirun -np <number_of_processes> ./test <input_file> PX PY PZ NX NY NZ NC <output_file>
```

Example:

```
mpirun -np 64 ./test data_64_64_64_3.txt 4 4 4 64 64 64 3 output_64_64_64_3_64_1.txt
```

The naming convention for the output file is as follows:

```
output_NX_NY_NZ_NC_<number of processes>_<iteration number>
```

3 Code Optimizations

The main bottlenecks identified and optimized in the provided code were:

1. **File I/O Bottleneck**: The original approach was sequential file reading by rank 0 followed by distribution. This was optimized using parallel file reading where each rank directly reads its data chunk, significantly improving I/O efficiency.
2. **Communication Bottleneck**: Non-blocking MPI communication (‘MPI_Isend‘ and ‘MPI_Irecv‘) was employed for efficient halo exchange, reducing communication overhead and idle waiting times.

4 Results

The scalability was tested for various process configurations (np=8,16,32,64). Below are results for two key test cases:

4.1 Test Case 1: (NX,NY,NZ,NC) = (64,64,64,3)

The output results are:

```
Local Minima and Maxima counts per time step:
(37988,37991),(37826,38005),(37788,37978)

Global Minima and Maxima per time step:
(-48.2500,33.6300),(-51.4500,33.3500),(-48.5500,33.3500)
```

# Processes	Run	Read Time	Compute Time	Total Time
8	Iteration 1	0.006342	0.009374	0.017534
	Iteration 2	0.007409	0.010195	0.018461
	Average	**0.006876**	**0.009785**	**0.017998**
16	Iteration 1	0.006396	0.005539	0.013305
	Iteration 2	0.006619	0.005491	0.013561
	Average	**0.006508**	**0.005515**	**0.013433**
32	Iteration 1	0.007538	0.005083	0.014458
	Iteration 2	0.008285	0.005215	0.014841
	Average	**0.007912**	**0.005149**	**0.014650**
64	Iteration 1	0.017530	0.003184	0.021912
	Iteration 2	0.007848	0.003291	0.012598
	Average	**0.012689**	**0.003238**	**0.017255**

Table 1: Execution times for different numbers of processes

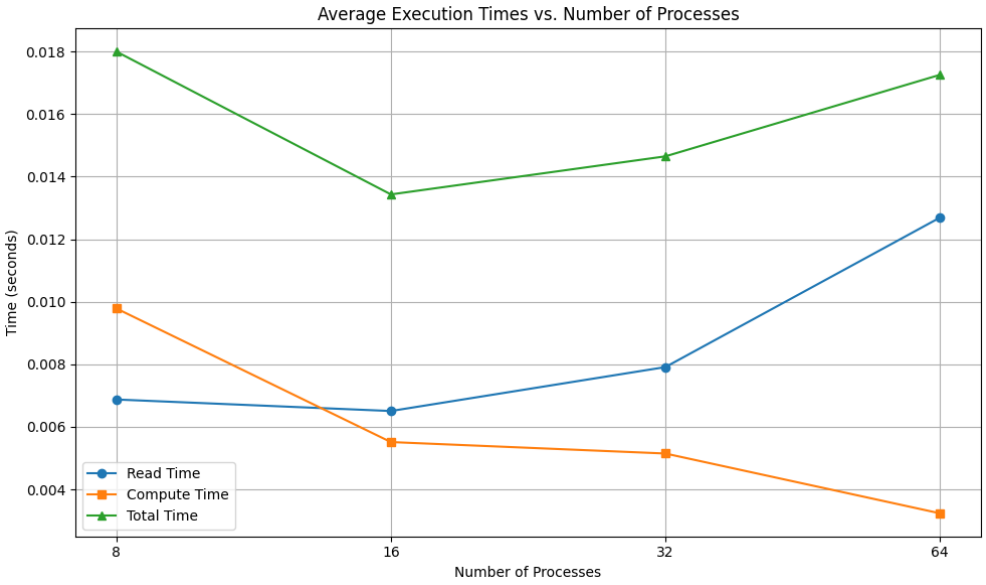

Figure 1: Graph of execution time

4.2 Test Case 2: (NX,NY,NZ,NC) = (64,64,96,7)

The output results are:

```
Local Minima and Maxima counts per time step:
(56875,56848),(56703,56965),(56680,56847),(56601,56980),
(56769,56937),(56657,56950),(56862,56996)

Global Minima and Maxima per time step:
(-48.2500,33.6300),(-51.4500,33.3500),(-48.5500,33.3500),
(-43.1300,32.0000),(-53.5500,34.0600),(-49.6800,34.1800),
(-53.5500,34.3400)
```

# Processes	Run	Read Time	Compute Time	Total Time
8	Iteration 1	0.011329	0.030761	0.043913
	Iteration 2	0.007409	0.010195	0.018461
	Average	**0.009369**	**0.020478**	**0.031187**
16	Iteration 1	0.010324	0.015646	0.027012
	Iteration 2	0.012165	0.015550	0.029160
	Average	**0.011245**	**0.015598**	**0.028086**
32	Iteration 1	0.010475	0.010781	0.022585
	Iteration 2	0.012122	0.011719	0.025328
	Average	**0.011299**	**0.011250**	**0.023957**
64	Iteration 1	0.024267	0.006342	0.031903
	Iteration 2	0.021461	0.006040	0.028960
	Average	**0.022864**	**0.006191**	**0.030432**

Table 2: Execution times for different numbers of processes

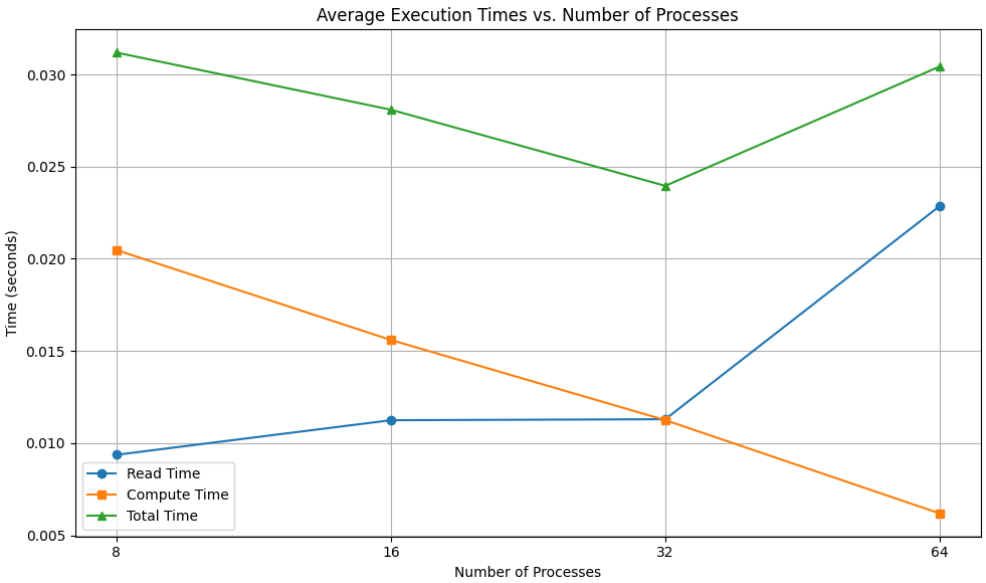

Figure 2: Graph of execution time

4.3 Performance Analysis

The parallel strategy shows good scalability with an increasing number of processes, particularly evident from the optimized parallel file I/O. The main computation time is significantly lower than file reading time, indicating that parallel I/O provided a clear advantage.

Test Case 1: (NX, NY, NZ, NC) = (64, 64, 64, 3)

This case involves a smaller time-series dataset (only 3 time steps), so compute workload is relatively low. Here's what we observe:

- **Read Time:** The read time improves from 8 to 16 and worsens from 16 to 64 processes. It decreases from 8 to 16 because of parallel I/O but then the system overhead increases. The increased overhead is because the system attempts to handle more simultaneous requests, leading to contention for resources like disk access and communication between processes. For instance, it increases from 0.006508 (16 processes) to 0.012689 (64 processes),
- **Compute Time:** Compute time benefits significantly from increased parallelism, decreasing from 0.009785 (8 processes) to 0.003238 (64 processes), a more than 3x speedup.
- **Total Time:** The total time shows a consistent drop from 0.017998 (8 processes) to 0.014650 (32 processes), then increases to 0.017255 at 64 processes. This suggests that I/O contention begins to balance out the gain from more parallel computation at 16 processes.

Test Case 2: (NX, NY, NZ, NC) = (64, 64, 96, 7)

This dataset has higher computational demands (7 time steps and larger Z-dimension), so it scales more effectively.

- **Read Time:** The read time worsens, from 0.009369 (8 processes) to 0.011245 (16 processes), to 0.011299 (32 processes) and finally to 0.022864 (64 processes) , likely due to I/O contention.
- **Compute Time:** The compute time steadily drops from 0.020478 (8 processes) to 0.006191 (64 processes), showing excellent parallel scalability. This suggests that the larger number of time steps (NC=7) provides enough work to efficiently utilize more cores.
- **Total Time:** Total execution time drops significantly from 0.031187 (8 processes) to 0.023957 (32 processes) and then increases slightly to 0.030432 (64 processes).Here, again the I/O contention begins to balance out the gain from parallel computation.

General Observations:

- For both test cases, compute time decreases with increasing number of processes, validating that the domain decomposition and communication strategies scale well.
- Read time benefits from parallel I/O in comparison to sequential read. Still, the time increases as the number of processes increases consistently at higher process counts—likely due to increased I/O contention on shared filesystems.
- The impact of NC (number of time steps) is clear—larger NC in Test Case 2 makes better use of additional processes, while in Test Case 1, speedup flattens earlier.
- At 64 processes, an increase in total time is likely due to increased overhead in synchronization and communication.

5 Conclusions

The implemented parallelization strategy using 3D domain decomposition, direct parallel file reading, and efficient halo exchange successfully reduced computational time and improved scalability. Each group member significantly contributed as follows:

- **Lakshika:** Conducted performance tests and ran the jobs on SLURM, reported the times, prepared saturation analysis, domain decomposition logic.
- **Rohit:** Implemented parallel file I/O mechanism, data distribution strategy in sequential case, data structures and domain decomposition logic
- **Sharvani:** Code for halo exchange communication, finding local and global minima and maxima with edge case, MPI_Reduce logic
- **Vanshika:** Created new data structures of type-vector, created subcommunicator to find xrank, yrank, zrank, and completed the documentation.

This collaborative effort effectively addressed all required tasks of the assignment, demonstrating improved efficiency and scalability.