CS633 Assignment Group Number 31

Anaswar K B (220138)
Anisurya Jana (220152)
Khushi Gupta (220531)
Pathe Nevish Ashok (220757)
Wadkar Srujan Nitin (221212)

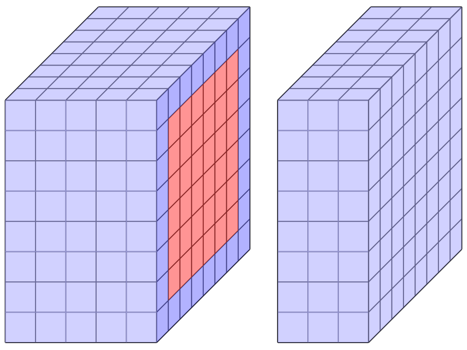
April 13, 2025

1 Code Description

The code implements a parallel algorithm for processing a time series dataset of a 3D volume. The solution is designed using MPI and employs a 3D domain decomposition strategy. In this section, we provide a detailed explanation of the design, highlighting domain decomposition, boundary data exchange using a 7-point stencil, and local/global computations. Additionally, we briefly incorporate MPI_Pack/MPI_Unpack routines in our boundary exchange functions.

1.1 Domain Decomposition and Grid Point Distribution

The overall 3D grid, defined by the dimensions `NX` (X-dimension), `NY` (Y-dimension), and `NZ` (Z-dimension), is decomposed among multiple processes arranged in a 3D process grid specified by `PX`, `PY`, and `PZ`. The decomposition is carried out as follows (the red part indicates the local sub-domain portion whereas the violet part indicates the exchanged portion):



- **Process Grid Allocation:** The total number of processes is $P = PX \times PY \times PZ$. Each process is assigned a unique rank and its coordinates in the process grid are determined by:

$$x_{\text{coord}} = \text{rank} \mod PX, \quad y_{\text{coord}} = \left(\frac{\text{rank}}{PX}\right) \mod PY, \quad z_{\text{coord}} = \frac{\text{rank}}{(PX \times PY)}$$

- **Sub-domain Size Calculation:** The sub-domain dimensions (m_x, m_y, m_z) for each process are computed such that grid points are distributed uniformly. ***In cases where the global dimensions are not exactly divisible by the number of processes in that dimension, the process at the boundary gets the remaining grid points.*** This is done for each spatial dimension, ensuring that every process handles a contiguous block of points.
- **XYZ Order Distribution:** The grid points are allocated to processes in an XYZ order such that the data is partitioned with the X-index varying fastest, followed by Y, and then Z. Each process is assigned a contiguous block of data corresponding to its sub-domain, with grid points arranged following the natural ordering in the three-dimensional space.
- **Local Array Allocation:** Each process allocates a 4D local data array for its sub-domain denoted by `local_data[size_x][size_y][size_z][nc]`. The array dimensions are augmented by a two-layer padding (halo or ghost cells) along each spatial dimension. This padding is required to store neighboring values from adjacent sub-domains and is essential for the correct evaluation of local extrema.

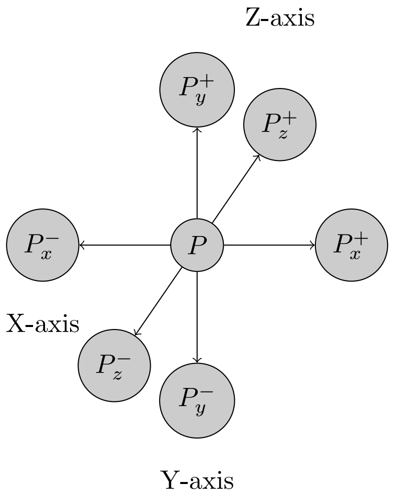
1.2 Parallel I/O Handling

Each process reads its local subdomain directly from the global file using `MPI_File_open` and `MPI_File_read_at`. Offsets are calculated based on the process's coordinates in the 3D grid. This avoids bottlenecks of serial I/O and ensures scalable input performance. Edge cases for boundary processes are handled correctly by computing their local sizes accordingly.

1.3 Boundary Data Exchange Using 7-Point Stencil

An integral part of ensuring accurate local extrema computation is the exchange of boundary (halo) data between neighboring processes. This code uses a 7-point stencil for performing the computation at each grid point. The 7-point stencil involves the grid point itself and its six immediate neighbors in the X, Y, and Z directions. Specifically, for a grid point P at (x, y, z), the stencil includes:

- P at (x, y, z) — the center,
- P_x^- at $(x-1, y, z)$ — left neighbor,
- P_x^+ at $(x+1, y, z)$ — right neighbor,
- P_y^- at $(x, y-1, z)$ — neighbor below,
- P_y^+ at $(x, y+1, z)$ — neighbor above,
- P_z^- at $(x, y, z-1)$ — back neighbor,
- P_z^+ at $(x, y, z+1)$ — front neighbor.



In practice, each process sends the required boundary layers using dedicated `Send` and `Recv` routines that handle the six directions. This exchange ensures that for every interior grid point near the boundary, all six neighbors are available for comparison. The even-odd communication pattern is adopted to avoid deadlocks during this process.

1.4 Local Computation of Extremum Values

Once the boundary data is in place, each process computes:

- **Local Extrema Counts:** The algorithm iterates over each interior point (excluding the halo cells) and compares its value with the six neighboring points (adjacent in X, Y, and Z). If a point is **less than or equal to** all six neighbors, it is counted as a local minimum; similarly, if it is **greater than or equal to** all six neighbors, it is marked as a local maximum.
- **Global Extremes:** During the same traversal, each process determines the global minimum and maximum values of its subdomain for every time step. These values are stored in dedicated local arrays.

The comparisons are carefully structured to account for edge conditions where a process represents a boundary of the global domain.

1.5 Global Aggregation Using MPI Reduction

After computing the local counts and extreme values, the code aggregates the results across all processes:

- **Count Reduction:** Using `MPI_Reduce` with the `MPI_SUM` operator, the local counts of minima and maxima are summed to yield the global counts.
- **Extreme Value Reduction:** `MPI_Reduce` is also used with `MPI_MIN` and `MPI_MAX` operators to compute the overall global minimum and maximum.

These reduction operations ensure that rank 0 has the complete and accurate results for the entire 3D domain for every time step.

1.6 Timing and Performance Measurement

The assignment requires measurement of several performance metrics:

- **Data Distribution Time:** The time taken from the start of processing (after MPI initialization) until the data has been successfully distributed among all processes and the halo exchange has also been performed.
- **Main Computation Time:** The interval during which each process performs local computation of extrema, and the subsequent reduction operations.
- **Total Execution Time:** The complete duration from initialization to the final output.

2 Code Optimization

We apply two main optimizations in our implementation: **Parallel I/O** and an **Odd-Even Halo Exchange Pattern** in each dimension.

2.1 Parallel I/O

Parallel I/O is implemented using MPI I/O routines. Each process opens the shared input file concurrently with `MPI_File_open` and reads its assigned block of data using `MPI_File_read_at`. The offset for each process is calculated based on its starting indices and sub-domain dimensions in the global 3D grid.

The offset is computed as:

$$\text{offset} = (st\_z \times nx \times ny \times nc + st\_y \times nx \times nc + st\_x \times nc) \times \text{sizeof(float)}$$

where:

- st_x, st_y, st_z are the (global) starting indices for the X, Y, and Z dimensions of the sub-domain.
- nx and ny are the global dimensions in the X and Y directions.
- nc is the number of time steps (columns).
- sizeof(float) converts the element count to the proper byte offset.

This calculation ensures that each process reads exactly the block of data corresponding to its sub-domain in the correct XYZ order.

2.2 Odd-Even Halo Exchange

Halo exchanges are performed in each of the three directions using the odd-even 1D exchange based on the process's coordinate in that dimension (as discussed in class). For example, in the x-direction, processes with even x-coordinates send first and receive after, while odd ones receive first and then send. This ensures parallelism in execution which would otherwise have been sequential without the odd-even separation.

2.3 MPI_Pack Integration

`MPI_Pack` and `MPI_Unpack` routines have been integrated into our Send and Recv functions to pack boundary data into contiguous buffers before transmission. Although this approach was intended to optimize communication, performance measurements indicated that the use of `MPI_Pack` *did not* yield a significant improvement. Thus, while `MPI_Pack` is included in the code description, its impact on performance was negligible and did not result in overall code optimization.

3 Code Compilation and Execution Instructions

1. Compilation
Compile the MPI files using `mpicc`:

```
# Optimised version
mpicc src.c -o a

# Unoptimised version
mpicc src_unoptimised.c -o a_unopt
```

2. Preparing Sample Test Cases
Run the following script to prepare the test cases:

```
./prepare_samples.sh
```

3. Running Jobs in Each Run Folder
Inside each `Run#x` folder within the `Sample_Test_Cases` directory, run:

```
cd Sample_Test_Cases/Run#1
sbatch sample1.sh
sbatch sample2.sh

cd ../Run#2
sbatch sample1.sh
sbatch sample2.sh
# Repeat for other Run#x folders
```

4. Generating Data for Opt vs Unoptimised
Use the job scripts in the `Opt vs Unopt` directory to generate data for performance comparison between the optimised and unoptimised binaries (`a` and `a_unopt`).

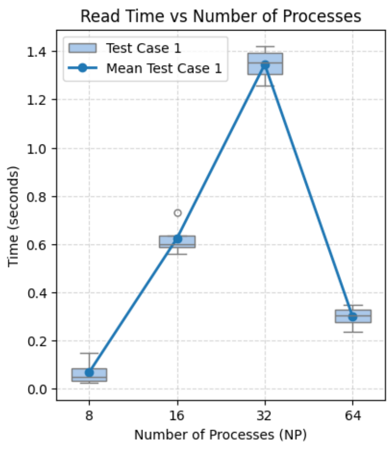
5. Generators for Large Files
The `Generators` folder contains scripts to generate large input files for the job scripts used in the `Large Files` setup. If the generator's output is too small, concatenate the file multiple times:

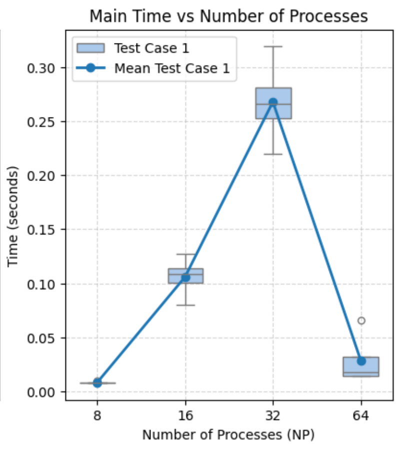
```
for i in {1..n}; do cat small_file >> large_input_file; done
```

6. Plotting Results
The Plot Scripts are in the file `Plot_scripts.ipynb`

4 Results

4.1 Test Case 1 (`data_64_64_96_3.bin.txt`)





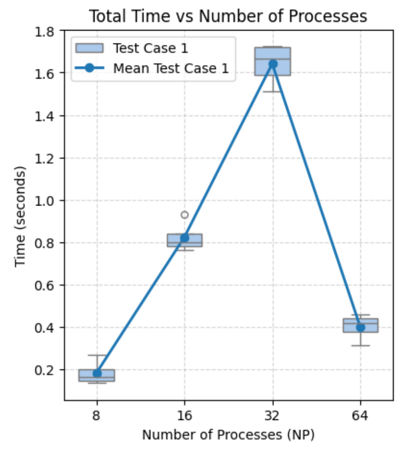
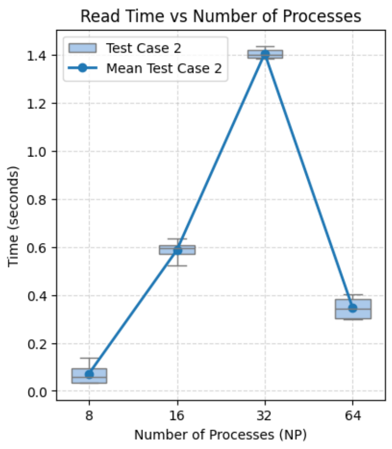
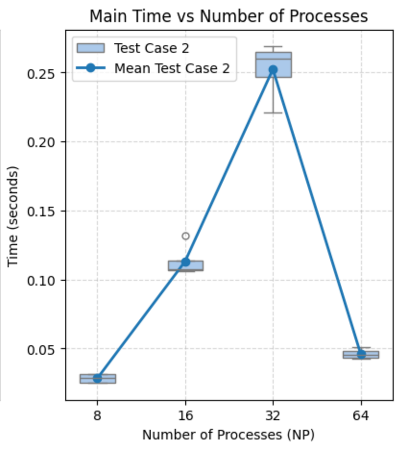


Table 1: Timing Results for Test Case 1 (seconds)

NP	Read Run 1	Read Run 2	Main Run 1	Main Run 2	Total Run 1	Total Run 2
8	0.024100	0.061245	0.007836	0.007812	0.134730	0.174919
16	0.596905	0.558874	0.079800	0.106966	0.786549	0.763940
32	1.419834	1.258020	0.318964	0.219985	1.719661	1.511722
64	0.349063	0.320610	0.020591	0.014465	0.459671	0.431441

4.2 Test Case 2 (`data_64_64_96_7.bin.txt`)





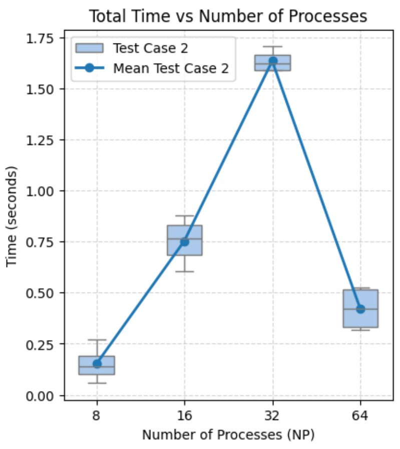
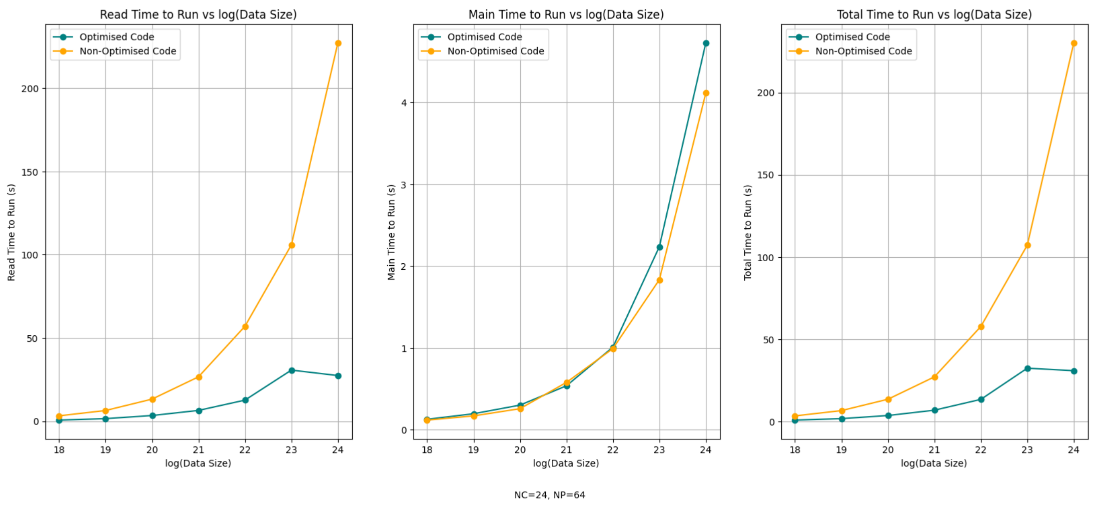


Table 2: Timing Results for Test Case 2 (seconds)

NP	Read Run 1	Read Run 2	Main Run 1	Main Run 2	Total Run 1	Total Run 2
8	0.033054	0.033278	0.025135	0.025060	0.058341	0.165621
16	0.520485	0.635947	0.107887	0.131968	0.606572	0.875888
32	1.385812	1.382044	0.268976	0.255678	1.588262	1.652380
64	0.298941	0.402264	0.042853	0.043158	0.319234	0.526404

4.3 Optimised Vs Unoptimised Code

We find a significant dip in the times for Optimised Code (i.e with Parallel I/O) than sequential reads in the case for Unoptimised Code



4.4 Surface Plots

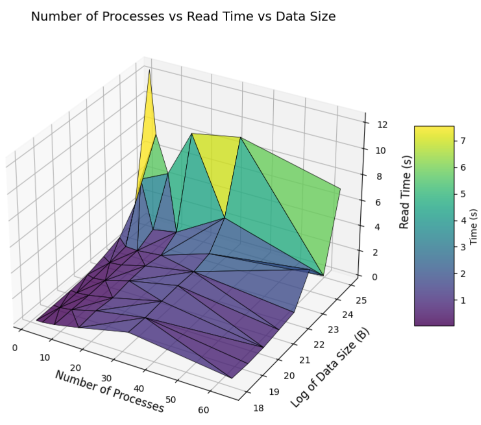
We analysed the dependence of Data Size and Number of Processes on the time for different stages

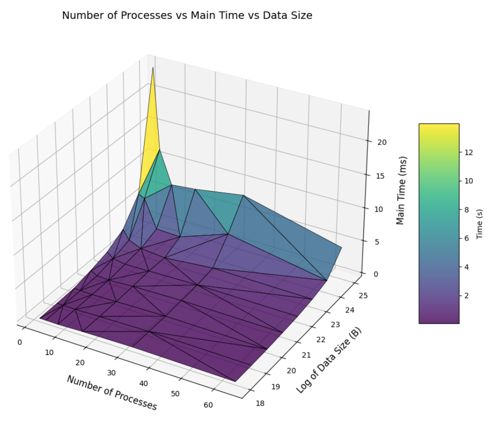
5 Discussion and Analysis of Results

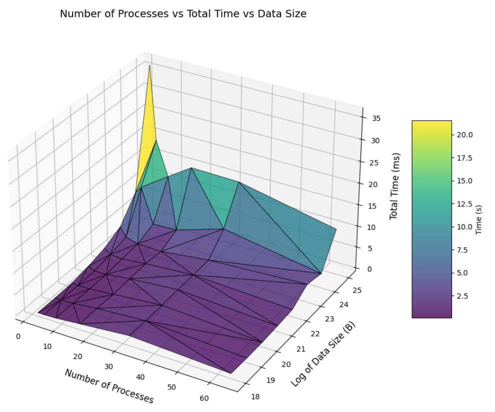
This section provides a detailed discussion of the observations and analysis of our parallel code's performance and efficiency. Both the strengths and limitations of the current implementation are evaluated, and potential improvements are discussed.

5.1 Observations from Execution

Execution Time Trends: We observe that as the number of processes increases, the execution time initially tends to rise when the data size is kept constant. This increase is likely due to the growing overhead from inter-process communication, as the computation time itself is not a dominant factor in the total execution time. However, a notable drop in execution time occurs when scaling from 32 to 64 processes. This suggests that the benefits of parallel I/O begin to outweigh the communication overhead at higher process counts, leading to improved overall performance.







Scalability: Initially, with smaller data sizes, parallelizing the compute process does not lead to a significant reduction in total execution time, as communication overhead dominates. However, when we tested the code on larger datasets (as illustrated in the surface plots), we observed a clear decrease in execution time with an increase in number of processes. This indicates that our solution scales effectively with data size.

5.2 Code Efficiency and Inefficiencies

Features:

- **Efficient Domain Decomposition:** The chosen 3D decomposition model ensures that each process handles a contiguous sub-domain. In case the global dimensions are not divisible by the number of processes in that direction, the remaining grid points are given to the last process.
- **Optimized Parallel I/O:** With parallel I/O, processes can concurrently read their respective data segments from the global file. This avoids a bottleneck at the root process and is particularly beneficial when dealing with large datasets.
- **Effective Use of Halo Exchange:** The 7-point stencil implemented with an odd-even halo exchange pattern ensures that the exchanges happens in parallel reomving the bottleneck of network exchange.

Areas for Improvement:

- **Overhead in Data Distribution:** Although parallel I/O has been incorporated to enhance scalability, the overhead incurred during the offset computations and MPI initialization increases the initial run times. To further reduce the overheads, we can explore more advanced MPI I/O strategies, such as collective I/O operations.
- **Communication Overhead:** The communication costs associated with exchanging boundary data, especially at higher process counts, suggest that overlapping communication with computation could be an effective strategy. The use of non-blocking MPI routines (e.g., `MPI_Isend` and `MPI_Irecv`) might help in masking the latency and improving overall performance.
- **Limited Impact of MPI_Pack:** Although MPI_Pack and MPI_Unpack routines were integrated to ensure contiguous data communication, the measured performance benefit was marginal. It appears that for the relatively small sizes of the exchanged halo regions, the overhead of packing and unpacking can outweigh the benefits. An alternative approach would be to utilize MPI derived data types (such as `MPI_Type_vector`) for direct non-contiguous data transfers, which may simplify the code and reduce unnecessary overhead.

6 Conclusions

The implementation of the parallel code using MPI was divided into several parts. The contributions for each part are detailed below:

- **Sequential I/O:** Reading entire data from process 0 and using MPI_Scatterv() for exchanging data. *(Anaswar K B,Anisurya Jana)*
- **Parallel I/O:** Implementing parallel reads using MPI_file_read_at() functions and calculating appropriate offset for each process . *(Srujan Wadkar, Nevish Pathe)*
- **3D domain decomposition and data structuring:** Structuring the received data in a 3D array for further exchange and edge case handling. *(Khushi Gupta, Nevish Pathe)*
- **3D Halo Exchange:** Performing Halo exchange along each dimension keeping in mind parallelism and even-odd distinction. *(Anaswar K B, Srujan Wadkar)*
- **Local Computation and Handling Computed Data:** Computing the count of local minima/maxima and sub-domain minima/maxima and using MPI_Reduce() for applying appropriate function. *(Nevish Pathe, Khushi Gupta)*
- **Testing and Experimentation:** Writing job scripts, running code on multiple input files, debugging and testing. *(Srujan Wadkar, Anaswar K B, Nevish Pathe)*
- **Plot Making and Analysis:** Making plots from the data inferred via testing and experimentation and drawing inferences based on it. *(Khushi Gupta, Anisurya Jana)*
- **Report Writing:** Everyone contributed equally for structuring and writing the report.