CS633 Assignment Group Number 01

Pragati Agrawal (220779)
Dhruv Gupta (220361)
Kundan Kumar (220568)
Anshu Yadav (220171)
Param Soni (220752)

1 Code Description

1.1 Introduction:

In the problem we need to compute the count of local minima and local maxima, and global min/max for a 3D domain for multiple time steps. The data is given as 2D matrix where each column contains data for all the points in row-major order at a particular time step. To solve this, we tried three different methods -

- **Method 1 :** Rank 0 reads all the data and distributes the respective data to all the processes. Then each process communicate the data corresponding to boundary points in its subdomain with the corresponding neighbour. After this each process perform the computation required. Then all the results are collected by Rank 0 using MPI_Reduce, which then write the output in the output file.
- **Method 2 :** Since there is an upper limit (of around 180 GB on param-rudra, depends on system used) on the amount of memory which can be dynamically allocated by a process, it may not be possible to read complete data at once if data size is large. Therefore, rank 0 reads data corresponding to only 1 time step at a time and distributes it. All processes do all the communications and computations for that data, and then reuse the same memory for the next time step. Everything else remains same as Method 1.
- **Method 3 (optimized) :** We utilize parallel IO, i.e., now all the processes read their data in parallel. We also combine the ideas in methods 1 & 2. If the data size is large, the data is read and processed one time step at a time. But for small data, all the data is read at once. The threshold for deciding the boundary to switch between the two is defined using a macro MAX_RAM_READ which is set to 150 GB, slightly less than 180 GB.

Each method involves 3D decomposition. This report primarily discusses the final optimized version in detail.

1.2 Code Flow Diagram:

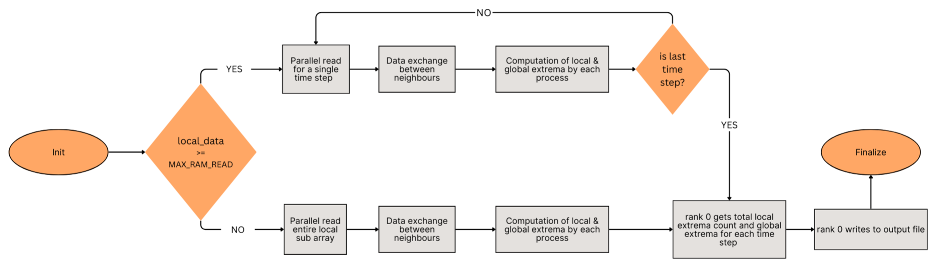
Init
local_data >= MAX_RAM_READ
YES
NO
Parallel read for a single time step
Data exchange between neighbours
Computation of local & global extrema by each process
is last time step?
NO
YES
Parallel read entire local sub array
Data exchange between neighbours
Computation of local & global extrema by each process
rank 0 gets total local extrema count and global extrema for each time step
rank 0 writes to output file
Finalize

Figure 1: Flow Chart showing the Code flow

1.3 Definitions:

- **local_nx, local_ny, local_nz:** Number of grid points in the local subdomain along the X, Y, Z axis respectively.
- **local_data:** A 1D float array that stores the 4D data for the local subdomain after 3D spatial decomposition. It has dimensions $local\_nx \times local\_ny \times local\_nz \times local\_nt$ and is accessed in a flattened format. Note that $local\_nt = 1$ when data is read one timestep at a time, and $local\_nt = NC$ when whole data is read at once.
- **local_min_val[NC], local_max_val[NC]:** Arrays storing the local minimum value and maximum value respectively found in the local subdomain for each of the NC timesteps. This is later reduced across all processes to compute the global min, max value.
- **local_min_cnt[NC], local_max_cnt[NC]:** Arrays storing the number of the local minimas, local maximas respectively for each of the NC timesteps in the local subdomain of the process. This is later reduced across all processes to compute the global count.
- **global_min_val[NC], gloabal_max_val[NC]:** Arrays storing the global minimum value and maximum value respectively found in the full 3D domain for each of the NC timesteps.

- **global_min_count[NC], global_max_count[NC]:** Results of `MPI_Reduce` across all `local_min_cnt` arrays , `local_max_cnt` arrays respectively. Represent the total number of local minimas, local maximas respectively for each timestep.
- **local_data_size:** Total number of float values in the local subdomain handled by a single MPI process.
- **total_data_size:** Total number of float values in the full 4D dataset before decomposition.
- **X, Y, Z:** The coordinate of the process along the X, Y, Z axis respectively in the 3D process grid. These are computed as:

$$X = rank \bmod PX$$

$$Y = \left(\frac{rank}{PX}\right) \bmod PY$$

$$Z = \left\lfloor \frac{rank}{PX \times PY} \right\rfloor$$

- **P:** Total number of MPI processes used for the 3D decomposition:

$$P = PX \times PY \times PZ$$

- **x_left, x_right, y_front, y_back, z_top, z_bottom:** Data3D structs for storing the neighbors communicated data respective faces. It consist of pointer to 1D array data, x , y, z axis values which are stored as XYZ format in data array.

1.4 Parallel I/O Read after 3D Decomposition

Pseudocode:

Algorithm 1 Parallel file set view and read using Type_create_subarray

```
Inititalisation{
    // Initialise local and global variables
    ...
}
MPI_File_open($in\_file$) // Open file
// Create and commit subarray type using MPI_Type_create_subarray
// Structures to create Type_create_subarray

int $sizes[4] = \{NZ, NY, NX, NC\}$;
    // $local\_nt = 1$ if reading one time step at a time, $NC$ otherwise
int $subsizes[4] = \{local\_nz, local\_ny, local\_nx, local\_nt\}$;
int $starts[4] = \{0, 0, 0, 0\}$;
MPI_Datatype $subarray\_type$;
MPI_Type_create_subarray(4, $sizes$, $subsizes$, $starts$,MPI_ORDER_C,
                          MPI_FLOAT, &$subarray\_type$);

MPI_Type_commit(&$subarray\_type$);

// Compute file displacement

$displ = \left(\left(Z \cdot \frac{NX \cdot NY \cdot NZ}{PZ} + Y \cdot \frac{NX \cdot NY}{PY} + X \cdot \frac{NX}{PX}\right) \cdot NC\right) \cdot \texttt{sizeof(float)}$
MPI_File_set_view(..., $displ$, ...)
// Use displacement calculated above for offset
MPI_File_read_at_all(...,0, $local\_data$, ...)
// Collectively read into local_data. Use offset=0, since file view is set.
```

To perform efficient parallel I/O on a 4D dataset (of dimensions `[NZ, NY, NX, NC]`) distributed across different process grid, we used `MPI_Type_create_subarray` **Algorithm 1** to define a custom subarray datatype corresponding to each process's local subdomain. This abstraction allows MPI to internally handle non-contiguous memory layouts and gives much better performance.

1.5 Face Exchange Using `MPI_Type_vector`

Pseudocode:

Algorithm 2 Communication between neighbours using Type_vector

```
Create subarray MPI datatypes{
    $vector\_type1$ : (stride = $local\_nt$, block size = $local\_ny * local\_nz$)
    $vector\_type2$ : (stride = $local\_nx * local\_nt$, block size = $local\_nz$)
}

Exchange faces along X-direction
if($rank \% PX > 0$)
    MPI_Isend (left face, vector_type1 to rank-1)
    MPI_Irecv (left face,local_ny*local_nz*local_nt floats from rank-1)

if(rank % $PX < PX - 1$)
    MPI_Isend (right face, vector_type1 to rank+1)
    MPI_Irecv (right face,local_ny*local_nz*local_nt floats from rank+1)

Exchange faces along Y-direction
if($(rank/PX)\%PY > 0$)
    MPI_Isend (front face, vector_type2 to rank-PX)
    MPI_Irecv (front face,local_nx*local_nz*local_ntfrom rank-PX)

if($(rank/PX)\%PY < PY - 1$)
    MPI_Isend (back face, vector_type2 to rank+PX)
    MPI_Irecv (back face, local_nx*local_nz*local_nt from rank+PX)

Exchange faces along Z-direction
if($rank/(PX.PY) > 0$)
    MPI_Isend(top face, first local_nx·local_ny·local_nt to rank-PX·PY)
    MPI_Irecv(top face, last local_nx·local_ny·local_nt from rank-PX·PY)

if($rank/(PX.PY) < PZ - 1$)
    MPI_Isend(bot face, first local_nx·local_ny·local_nt to rank+PX·PY)
    MPI_Irecv(bot face, first local_nx·local_ny·local_ntfrom rank+PX·PY)
```

For checking if the points at the boundary of subdomain are local minimas/-maximas, each MPI process exchanges its six faces (along X, Y, and Z axes) with its neighboring processes. Faces are non-contiguous in memory. To address this, we use `MPI_Type_vector` to describe strided memory layouts for face slices:

- `vector_type1` captures planes orthogonal to X, used for left and right face exchanges.
- `vector_type2` captures planes orthogonal to Y, used for front and back face exchanges.
- since Z-directional faces are contiguous in memory, no derived type is needed for top and bottom exchanges.

This approach reduces complexity, avoids manual packing/unpacking, and allows efficient non-blocking point-to-point communication. Neighbor checks are based on rank positioning in the 3D process grid, and all six directions are handled conditionally depending on process location.

1.6 Local and Global Min/Max Computation

Explanation: Local and Global Min/Max Computation

After the halo exchange, each MPI process possesses all the required data to compute local minima and maxima for its subdomain. This includes not just the local grid data but also the boundary face data received from neighboring processes.
Each grid point (x, y, z) is compared against its six immediate neighbors: left-/right (in x), front/back (in y), and bottom/top (in z), across each timestep t. The algorithm proceeds as follows:

- If a neighbor lies within the local subdomain, its value is accessed directly from the `local_data` array.
- If a neighbor lies outside the local subdomain, but within the global domain, the value is fetched from the halo buffer corresponding to the appropriate direction (e.g., `x_left`, `y_back`, `z_top`, etc.).
- For each grid point and timestep, we find the minimum and maximum among its neighbors (denoted `min_temp` and `max_temp`).
- If the current value is less than all its neighbors, it is considered a **local minimum**, and the local minimum count and value for that timestep are updated.
- Similarly, if the current value is greater than all its neighbors, it is considered a **local maximum**, and the local maximum count and value for that timestep are updated.

Algorithm 3 Local Extrema Detection and Global Reduction

```
For $z = 0$ to $local\_nz - 1$
    For $y = 0$ to $local\_ny - 1$
        For $x = 0$ to $local\_nx - 1$
            For $t = 0$ to $local\_nt - 1$
                min_temp = DBL_MAX, max_temp = -DBL_MAX

                **Check 6 neighbors using local or halo data**
                if($x > 0$)
                    Compare with left neighbor in local data
                else if($rank \% PX > 0$)
                    Compare with x_left halo
                if($x < local\_nx - 1$)
                    Compare with right neighbor in local data
                elseif($rank \% PX > PX - 1$)
                    Compare with x_right halo

                if($y > 0$)
                    Compare with front neighbor in local data
                elseif($(rank/PX) \% PY > 0$)
                    Compare with y_front halo
                if($y < local\_ny - 1$)
                    Compare with back neighbor in local data
                elseif($(rank/PX) \% PY < PY - 1$)
                    Compare with y_back halo

                if($z > 0$)
                    Compare with bottom neighbor in local data
                elseif($rank/(PX * PY) > 0$)
                    Compare with z_bottom halo
                if($z < local\_nz - 1$)
                    Compare with top neighbor in local data
                elseif($rank/(PX * PY) < PZ - 1$)
                    Compare with z_top halo

                **Check if current value is local minimum/maximum**
                if($current\ value < min\_temp$)
                    Increment local_min_cnt[t]
                    Update local_min_val[t]
                if($current\ value > max\_temp$)
                    Increment local_max_cnt[t]
                    Update local_max_val[t]
MPI_Reduce(local_max_cnt → global_max_count, SUM, 0)
MPI_Reduce(local_min_cnt → global_min_count, SUM, 0)
MPI_Reduce(local_min_val → global_minimum, MIN, 0)
MPI_Reduce(local_max_val → global_maximum, MAX, 0)
MPI_Reduce all timing variables to compute max time across processes
```

Global Reduction

Once local extrema have been determined, MPI collective reduction operations are used to compute the global statistics:

- `MPI_Reduce` with `MPI_SUM` is used to aggregate the local minimum and local maximum counts into `global_min_count` and `global_max_count` arrays (one value per timestep).
- `MPI_Reduce` with `MPI_MIN` and `MPI_MAX` is used to compute the global minimum and maximum values per timestep.
- Timing for various phases of the program is measured using `MPI_Wtime()`, and `MPI_Reduce` with `MPI_MAX` is used to gather the maximum time taken for reading, computing, and total execution across all processes.

This ensures that rank 0 holds the final global results and performance metrics after synchronized parallel execution.

2 Code Compilation and Execution Instructions

- **Required Files**:
  - Source code: `src.c` (main source code file)
  - Job scripts (like `job_x_y_z.sh`) where
    - x stands for the data number-
      - x=1 → data_64_64_64_3.bin.txt
      - x=2 → data_64_64_96_7.bin.txt
    - y represents total number of processes, $y = \{8, 16, 32, 64\}$
    - z represents run number, $z = \{1, 2\}$
  - Automated sbatch script: `run.sh` - submits each job script using sbatch one by one with an interval of 30 seconds between each submission.
  - Input files (data_64_64_64_3.bin.txt and data_64_64_96_7.bin.txt)
- **Compilation**:

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

- **Execution**:

```
# Slurm submission for one job
sbatch <job_script.sh>

# Automated Slurm submission of all the jobs one by one
./run.sh
```

  Note: Each execution of `run.sh` generates outputs for 2 runs of each configuration. We renamed the output files and ran `run.sh` again to generate output of total 4 runs.

- **Output File**:
  - Output file names have same format as mentioned in the assignment, with run{k} appended to each, denoting the k-th run.
  - Output files contain 3 lines as specified below.
  - First line as (local min,local max) for each time step.
  - Second line as (global min,global max) for each time step.
  - In last line timing is printed as follow: Read Time, Main code Time, Total Time.

Job Script Example (for 8 processes on data_64_64_96_7.bin.txt)

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --ntasks-per-node=8
#SBATCH --error=job.%J.err
#SBATCH --output=job.%J.out
#SBATCH --time=00:10:00          ## wall-clock time limit
#SBATCH --partition=standard     ## can be "standard" or "cpu
    "

echo `date`
mpirun -n 8 ./src data_64_64_96_7.bin.txt 2 2 2 64 64 96 7
    output_64_64_96_7_8_run1.txt
echo `date`
```

3 Code Optimizations

- **Optimization 1 :** Reading from text files has to be done one float at a time. This involves a large number of IO read requests and hence takes more time. Instead, we used the binary files to read chunk of data (multiple values) at a time so that less IO read requests are needed.
- **Optimization 2 :** If a single process reads the data and distributes it, rest of the processes remain idle, thereby reducing efficiency. This can be seen as a load imbalance since only one process is reading. We noticed that the reading time is the bottleneck. Hence we used parallel IO so that each process directly reads its own data.
- **Optimization 3 :** Since each process' data is spread across the file, independent parallel IO would result in overlapping read requests. To optimize this further, we used collective parallel IO.
- **Optimization 4 :** We utilized the Type_create_subarray and Type_vector functions in MPI library to define the data to be read by each process instead of changing the offset each time. This allows us to exploit internal optimizations of the MPI library.
- **Optimization 5 :** To ensure that the data fits in the process' memory, we read the data one time step at a time. But this would need larger number of IO read requests. This is necessary when file size is large since reading the whole file at once is not possible. However, for smaller files, this would reduce the efficiency. Hence, we checked if the data size to be read by each process is less than MAX_RAM_READ, each process can read its complete data in one shot, otherwise, it would read its data for one time step at a time.

4 Results

4.1 Parallel I/O results

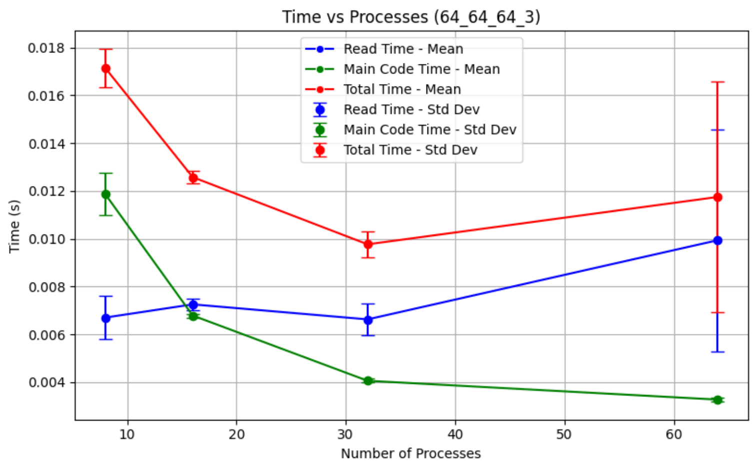

Figure 2: Parallel IO results for data_64_64_64_3.bin.txt

Table 1: Timings (in seconds) for data_64_64_64_3.bin.txt using Parallel IO

Process	Read Time	Main Code Time	Total Time
8	0.0067045	0.0118685	0.01714525
16	0.00725725	0.00679175	0.01257825
32	0.00662975	0.004062	0.009768
64	0.00993725	0.0032755	0.01174825

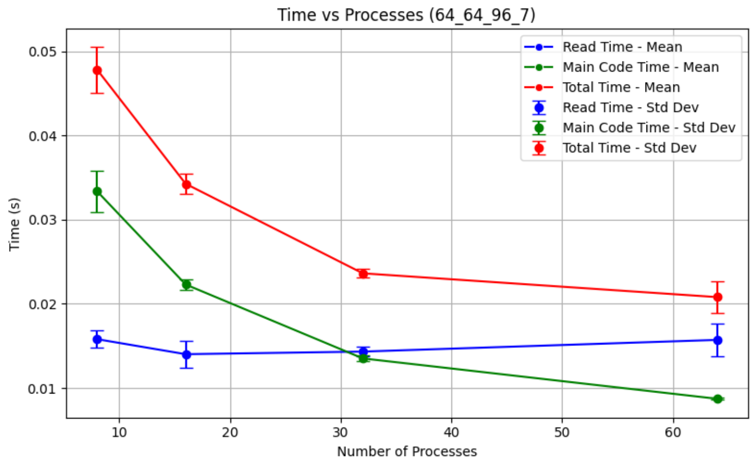

Figure 3: Parallel IO results for data_64_64_96_7.bin.txt

Table 2: Timings (in seconds) for data_64_64_96_7.bin.txt for Parallel IO

Process	Read Time	Main Code Time	Total Time
8	0.015805	0.033372	0.047821
16	0.014022	0.022299	0.034263
32	0.014330	0.013522	0.023626
64	0.015718	0.008726	0.020802

4.2 Inferences from Parallel I/O results

- For both datasets, increasing the number of processes reduces the **Main Code Time**, showing effective parallelisation of computation.
- In both cases, **Total Time** decreases steadily up to 32 processes.
- At 64 processes:
  - For `64×64×64×3`, Total Time increases slightly, suggesting performance saturation.
  - For `64×64×96×7`, Total Time continues to decrease, indicating that larger data benefits more from higher parallelism.
- **Read Time** remains inconsistent in both datasets due to I/O overhead; this suggests that parallel I/O may scale well only with larger dataset , smaller dataset have no improvement.

- In the smaller dataset (`64×64×64×3`), Read Time at 64 processes is worse than at 32, indicating I/O overhead because of over decomposition.
- In the larger dataset (`64×64×96×7`), Read Time fluctuates but has less impact on overall scalability unlike smaller dataset.
- The difference in Total Time between 8 and 64 processes is:
  - 31.4% reduction for the smaller dataset (0.0171 s to 0.0117 s)
  - 56.5% reduction for the larger dataset (0.0478 s to 0.0208 s)
- Larger datasets exhibit better scalability across more processes, making full use of parallel resources.
- **Parallel I/O** remains a limiting factor in both cases.
- For the smaller dataset, the optimal process count appears to be 32; for the larger one, it shifts to 64, suggesting good strong scaling.
- Standard Deviation is high for 64 processes, possibly due to variation in process placements during different runs in internode communication.

4.3 Sequential I/O (All data read) results

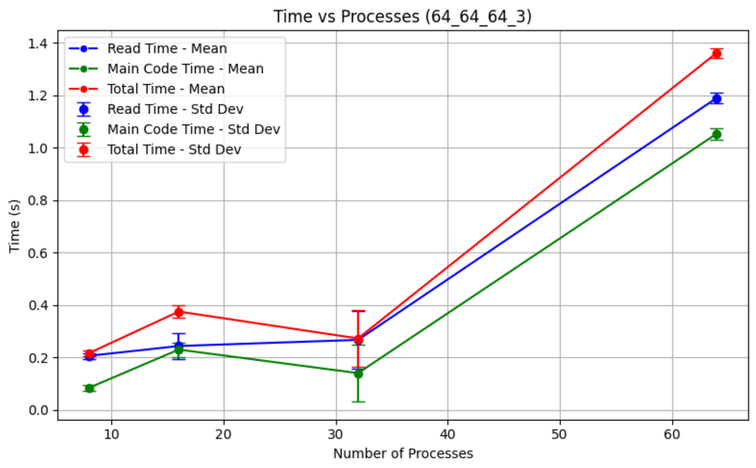

Figure 4: Sequential IO results for data_64_64_64_3.txt

Table 3: Timings (in seconds) for data_64_64_64_3.txt for Sequential IO

Process	Read Time	Main Code Time	Total Time
8	0.206265	0.084258	0.216657
16	0.243701	0.230083	0.375151
32	0.267150	0.140240	0.272594
64	1.190257	1.053918	1.362548

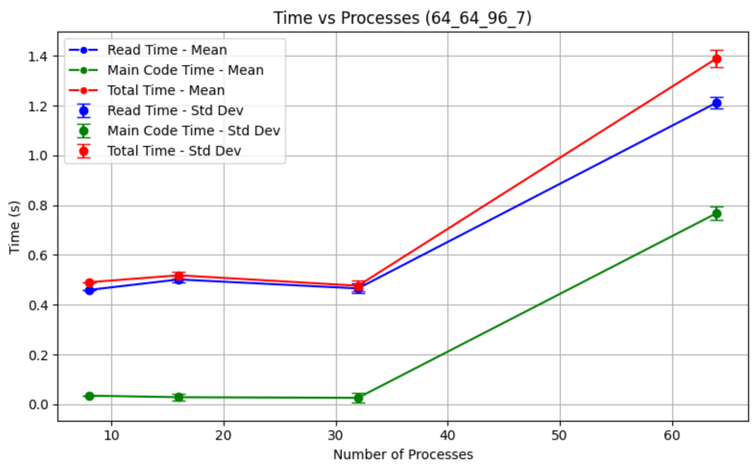

Figure 5: Sequential IO results for data_64_64_96_7.txt

Table 4: Timings (in seconds) for data_64_64_96_7.txt for Sequential IO

Process	Read Time	Main Code Time	Total Time
8	0.459599	0.034671	0.490316
16	0.501573	0.028551	0.518331
32	0.465673	0.026099	0.476295
64	1.212047	0.767842	1.389935

4.4 Inference from Sequential I/O Results

- For both datasets, **Read Time increases** significantly with more processes, especially at 64 processes.
- At 64 processes:
  - `64×64×64×3` shows a **5.7× increase** in Read Time compared to 8 processes.
  - `64×64×96×7` shows a **2.6× increase** in Read Time compared to 8 processes.
- **Main Code Time is irregular** across process counts, possibly due to variable synchronisation delays after data distribution.
- Total Time increases notably at 64 processes, driven primarily by large Read and Main Code times.

- Sequential I/O becomes a performance bottleneck at high process counts, as all processes wait for rank 0 to read and distribute data.
- For both datasets, the scaling trend is poor — adding more processes increases total execution time.

4.5 Sequential vs Parallel I/O: Comparative Analysis

- **Parallel I/O shows better scalability** with increasing processes for both datasets (up to 32 or 64), unlike Sequential I/O where performance degrades.
- For the `64×64×64×3` dataset:
  - Total Time at 64 processes is **1.36 s (Sequential)** vs **0.0117 s (Parallel)**, showing a drastic performance gap.
- For the `64×64×96×7` dataset:
  - Total Time at 64 processes is **1.39 s (Sequential)** vs **0.0208 s (Parallel)**, again confirming significant gains.
- **Sequential I/O does not benefit from increased process counts**; it suffers from centralised data reading and increased communication cost.
- **Parallel I/O shows strong scaling**, especially for larger datasets, although read time remains somewhat inconsistent due to I/O overhead.
- Parallel I/O, despite slight fluctuations, reduces both Read and Total Time drastically, making it a preferable approach for large-scale data handling.
- For high-performance parallel applications, **Parallel I/O is essential for scalability**, particularly when dataset size or process count grows.

5 Conclusions

Work done by each team member:

- **Pragati Agrawal (220779):**
  - Written and debugged code for halo exchange between neighbours in Method 2 (sequential IO code).
  - Contributed in codes for halo exchange using `Type_vector` and computation of extrema in code for Method 1 (sequential IO code).
  - Contributed in implementing optimizations using `MPI_File_read_at_all` in Method 3 and also, debugged and tested the correctness of both if-else parts in it (parallel I/O code).
  - Timed the parallel and sequential codes for finding the read time, main code time and total time.
  - Took various runs on Param Rudra for both parallel and sequential I/O codes.
  - Contributed in writing pseudocodes for Algorithm 1 and reviewed the report.
- **Dhruv Gupta (220361):**
  - Tested and debugged computation code for counts and values of extrema in Method 2 (sequential IO code).
  - Contributed in codes for reading and distributing data using `MPI_Type_create_subarray`, communication and computation in code for Method 1 (sequential IO code).
  - Contributed in implementing parallel IO using `MPI_File_set_view` and `MPI_File_read_at_all` in Method 3 (parallel I/O code).
  - Contributed in writing job scripts and `run.sh` script.
  - Contributed in plotting script `plot.py` and generated plots for both parallel I/O and sequential I/O.
  - Contributed in writing introduction and code optimizations in the report. Rechecked and corrected few errors in the report.
- **Kundan Kumar (220568):**
  - Developed the initial complete code , implemented core logic and halo exchange using `MPI_Send`/`MPI_Recv` to verify correctness and serve as a baseline for optimisation.
  - Initiated parallelisation by introducing `MPI_Scatter` replacing intial MPI_Send and MPI_Recv, which was later replaced with `MPI_Subarray` for improved efficiency.

  - Implemented optimised parallel I/O using `MPI_File_read_at_all` with dynamic offsets to minimise I/O request; collected data and compared it with `MPI_Subarray` to decide which performs better.
  - Contributed to optimisation and debugging across all stages of development to ensure correctness and performance.
  - Analysed performance data from multiple runs and documented data along with inferences in the report.
  - Contributed in code flow section of report to explain program logic.

- **Anshu Yadav (220171):**
  - Tested data reading code and order in Method 2(sequential IO code).
  - Contributed in executing job scripts output for Method 1 and 3.
  - Contributed in pseudo code for face exchange,code flow,definition of data structures, code compilation and execution in report.

- **Param Soni (220752):**
  - Tested code for Communication between Neighbours in Method 1 (sequential IO code).
  - Tested code for correct send and receives in correct order.
  - Contributed in writing Pseudo Code and explanation for Local Extrema Detection and Global Reduction in the report.
  - Contributed in writing Definitions for variables used, and the data structures defined.