CS633 Assignment

Parallel Computing

Professor: Preeti Malakar

Group Number 36

Group Members:

Keshav Raj Gupta (210508)

Mehar Goenka (210605)

Mridul Pandey (210634)

Rudransh Goel (210880)

Suryansh Goel (221111)

April 13, 2025

Contents

1 Code Description

1.1 Overview

The objective of this assignment is to process time series data defined on a 3D spatial domain ($NX \times NY \times NZ$), where each 3D point contains of multiple time steps (or channels). The goal is to determine the following, in parallel, for each time step:

1. Count of local minima
2. Count of local maxima
3. Global minimum
4. Global maximum

To achieve high performance, we leverage domain decomposition in 3D — partitioning the volume along the X, Y, and Z axes into a grid of MPI processes defined by (PX, PY, PZ). Each process is responsible for a sub-block of the overall domain and performs computations independently on its block, while communicating only with neighboring processes as needed.

Approach

The overall workflow of the program is composed of several logical stages:

1. Data Reading and Distribution. The first step involves reading the binary time-series volume data and distributing it among all MPI processes. Two broad strategies were implemented and benchmarked:

- **Single Rank Input + Scatter:** In this method, rank 0 reads the entire dataset and scatters relevant portions to other ranks using custom packing and `MPI_Scatterv`. This method is simple and effective for small- to medium-sized datasets, though it places a significant I/O burden on rank 0.
- **Parallel I/O with Multiple Readers:** A more scalable approach where multiple ranks (leaders) independently read non-overlapping parts of the data and distribute them to other ranks in their group. This method reduces I/O contention and utilizes the bandwidth of multiple processes.

Once each rank receives its corresponding data chunk, the data is reconstructed into a local 4D array of dimensions $local\_nx \times local\_ny \times local\_nz \times NC$, where NC is the number of time channels. This structure allows easy indexing and supports neighborhood-based operations required for detecting local minima and maxima.

2. Halo Exchange. To accurately compute local extrema near subdomain boundaries, a 1-layer halo region is introduced around each rank's 4D block. Each rank exchanges its boundary layers with its six neighboring processes (left, right, top, bottom, front, back) using non-blocking communication primitives (`MPI_Isend`, `MPI_Irecv`) followed by `MPI_Waitall`. This ensures that every process has valid neighbor information before performing local computations.

3. Local Extrema Computation. Each rank computes local minima and maxima by comparing each voxel's value against its 6-connected neighbors in 3D space, for every time step. If a voxel is strictly lower (or higher) than all its valid neighbors, it is counted as a local minimum (or maximum) for that time step.

4. Global Reduction. Each rank keeps track of:

- The number of local minima and maxima found per time step.
- The smallest local minimum and the largest local maximum it has seen.

Using `MPI_Reduce`, all local results are combined at rank 0 to compute:

- The global minimum and maximum for each time step.
- The total count of local minima and maxima across all ranks.

This modular design separates communication, I/O, and computation stages, making the code extensible and easy to optimize.

1.2 Code Explanation

1.2.1 Data Input & Distribution

Single-Rank Input and Distribution The first strategy we implemented for data reading and distribution included a single process (rank 0) reading the entire dataset. The data was then distributed to the remaining ranks.
The distribution strategy is as follows. After process with rank 0 reads the entire 4D dataset (with dimensions $NX \times NY \times NZ \times NC$) into memory, rank 0 partitions the volume into smaller subarrays of dimentions $(NX/PX) \times (NY/PY) \times (NZ/PZ)$. These subarrays then have the be assigned to each of the processes. To manage the complexity of indexing and memory layout, MPI subarray datatypes are used to describe the shape and location of each block. Each subarray/subdomain is packed into a contiguous buffer using `MPI_Pack`, and then distributed to the corresponding rank using `MPI_Scatterv`. On the receiving side, each rank unpacks its data into a local buffer.

```
Pseudocode: scatter_data(data, local_data, ...)
-----------------------------------------------
1. At rank 0:
```

```
For each rank p:
   a. Compute starting indices (start_x, start_y, start_z): denoting
   the offset where their data starts in the global array
   b. Create MPI subarray datatype for p's block
   c. Compute packed size for p
   d. Pack p's data into a contiguous buffer
   e. Use MPI_Scatterv to send packed blocks to each rank

3. On all ranks:
   a. Receive packed block
   b. Unpack into local_data buffer
```

This approach is robust and uses MPI's advanced datatype handling to represent subdomains without reshaping arrays manually. However, as the entire I/O load and all packing operations are done by rank 0, this method is not suitable for large-scale execution or high-performance environments.

Parallel Input and Distribution using Subcommunicators To overcome the limitations of centralized I/O, we implemented a parallel data reading strategy using MPI subcommunicators. The core idea is to divide the global communicator into multiple subcommunicators along the Z-axis, with each subcommunicator handling a contiguous slice of the 3D domain. Within each subcommunicator, rank 0 is designated as the leader responsible for reading a corresponding slice of the data from the binary file. The leader then scatters the data to the other ranks in its subcommunicator.

This strategy significantly improves performance by parallelizing file I/O across multiple ranks while still maintaining a structured and localized communication model. It also reduces memory overhead by ensuring that only the required portions of data are read into memory.

Subcommunicator Construction. The communicator splitting is done in two levels:

- **Subcommunicators by Z-slice:**

  ```
  int pz_coord = rank / (PX * PY);
  MPI_Comm_split(MPI_COMM_WORLD, pz_coord, rank, &subcomm);
  ```

  Each process calculates its Z-index using `pz_coord = rank / (PX * PY)`. MPI processes with the same Z-index are grouped into the same subcommunicator using `MPI_Comm_split`. Each subcommunicator contains all the processes in a single horizontal slice ($PX \times PY$) of the 3D domain. This allows localized communication within each layer.

- **Leader Communicator:**

```
int leader_color = (subrank == 0) ? 0 : MPI_UNDEFINED;
MPI_Comm_split(MPI_COMM_WORLD, leader_color, rank, &leader_comm);
```

Among each subcommunicator, only the rank with subrank 0 becomes part of a special leader communicator, which allows coordination among leaders. This communicator is used to coordinate MPI I/O operations such that only one process per pz layer accesses the disk.

Data Reading by Subcomm Leaders. Each subcomm leader reads its portion of the global 4D volume corresponding to a Z-slab using `MPI_File_read`. The offset is calculated based on the position of the slab in the full volume. This approach reduces I/O contention by limiting access to one process per slice.

```
Pseudocode: read_data_from_file_subcomm(...)
-------------------------------------------
1. Compute number of elements to read based on:
   - NX, NY, local_nz, NC
   - Adjust local_nz for edge slabs (if NZ % PZ  0)

2. Calculate byte offset into the binary file:
   offset = pz_coord * local_nz * NX * NY * NC * sizeof(float)

3. Open the file with MPI_File_open
4. Set view using MPI_File_set_view (starting at offset)
5. Read the contiguous slab into buffer using MPI_File_read
6. Close the file
```

Scattering Data within Subcommunicators. After reading the slab, the subcomm leader scatters the data to its subcomm peers. This is done using `MPI_Pack`, subarray datatypes, and `MPI_Scatterv`.

```
Pseudocode: scatter_data_subcomm(...)
--------------------------------------------
1. On subrank 0(leader for each subcommunicator):
   a. For each rank p in subcomm:
      - Determine (start_x, start_y)
      - Adjust block size for edges
      - Create MPI subarray datatype
      - Compute pack size, and pack into sendbuf
      - send via scatterv to all other ranks
2. All ranks:
   - Receive packed data with MPI_Scatterv
   - Unpack into local_data
```

Once each rank receives its corresponding data chunk, the data is reconstructed into a local 4D array of dimensions $local\_nx \times local\_ny \times local\_nz \times NC$, where

NC is the number of time channels. This structure allows easy indexing and supports neighborhood-based operations required for detecting local minima and maxima.
This method ensures efficient data access and communication by limiting file I/O to a subset of processes and using collective operations within subcommunicators. It significantly outperformed the centralized approach in our experiments, particularly at scale.

1.2.2 Halo Exchange

After data distribution, each MPI process holds a subvolume of the global 4D dataset. However, to correctly compute local minima and maxima near the boundaries of each subdomain, a one-layer "halo" (or ghost layer) is maintained around the local data. This halo stores neighboring values from adjacent processes so that local minima and local maxima computations can access data across subdomain boundaries without requiring special logic for boundary handling.

The function `exchange_halo()` handles this communication. It sends and receives boundary data in all six directions — left, right, top, bottom, front, and back — to the respective neighboring processes using non-blocking MPI communication primitives (`MPI_Isend`, `MPI_Irecv`), followed by `MPI_Wait` to ensure data consistency. Each direction has its own buffer for sending and receiving, ensuring clean and isolated communication.

```
Pseudocode: exchange_halo(...)
-----------------------------------------
1. Determine neighbor ranks in 3D grid:
   - left, right, top, bottom, front, back

2. For each neighbor:
   a. Extract boundary layer into send buffer
   b. Initiate non-blocking send (MPI_Isend)
   c. Initiate non-blocking receive (MPI_Irecv)

3. Wait for all sends and receives to complete:
   - Use MPI_Wait (or MPI_Waitall)

4. Copy received data into halo layers of the 4D buffer
```

Each subdomain is allocated with a halo region of size +1 on each face, i.e., it spans from index `[0...local_nx+1]` in X, and similarly for Y and Z. Actual data values are stored in the range `[1...local_nx]` in each dimension, while the zero-th and last index are used for halo cells.

This method is critical for enabling correct detection of extrema at the domain boundaries, especially when using distributed memory parallelism. Moreover, using non-blocking communication allows for potential overlapping with

computation, though in our current implementation, halo exchange is performed before computation begins.

1.2.3 Local Minima and Maxima Computation

Once the halo exchange is complete, each MPI process possesses a fully padded local subdomain with one layer of ghost cells in all six directions. This ensures that all internal points, including those at the edges of the subdomain, have immediate access to neighboring values for comparison. Each process then iterates over its interior grid (excluding the halo), and for each time step, checks if a point is a local minimum or maximum by comparing it against its six spatial neighbors. If a point is strictly smaller (for minima) or strictly larger (for maxima) than all its valid neighbors, it is counted as a local extremum. These extremum values are stored in arrays. Additionally, each rank tracks the smallest local minimum and the largest local maximum across its own subvolume for each time step.

1.2.4 Global Reduction and Output

After computing local extrema, each process performs a reduction operation to find the global minimum and maximum for each channel across all subdomains. This is achieved using `MPI_Reduce` with appropriate operators (`MPI_MIN` and `MPI_MAX`) to gather global statistics at rank 0. Similarly, the total count of local minima and maxima across all ranks is computed using `MPI_Reduce` with the `MPI_SUM` operator. Once all global statistics are available at rank 0, the results are written to an output file. This file includes the global minimum and maximum for each channel, the total count of minima and maxima, and a breakdown of the time taken for data reading and total execution. This final step consolidates all distributed results and provides a summary of the computation's correctness and performance.

2 Code Compilation and Execution Instructions

2.1 Compilation

To compile the 'src.c' program with MPI support, use the following command with an MPI compiler wrapper such as `mpicc`:

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

Ensure that the MPI development libraries and headers are installed on your system.

2.2 Execution Using SLURM

To run the program on a cluster managed by SLURM, you can use the provided batch script `job2.sh`.

SLURM Script: `job2.sh`

```
#!/bin/bash
#SBATCH -N 2
#SBATCH --ntasks-per-node=32
#SBATCH --error=job.%J.err
#SBATCH --output=job.%J.out
#SBATCH --time=00:00:02          ## wall-clock time limit
#SBATCH --partition=standard     ## can be "standard" or "cpu"

echo ‘date‘
mpirun -n 64 ./src data_64_64_64_3.bin.txt 4 4 4 64 64 64 3 output_64_64_64_3_64.txt
echo ‘date‘
```

Explanation of Parameters

- `#SBATCH -N 2`: Requests 2 compute nodes.
- `#SBATCH --ntasks-per-node=32`: Launches 32 MPI tasks per node, for a total of 64.
- `mpirun -n 64`: Runs the program using 64 MPI processes.
- `src`: Executable name.
- `data_64_64_64_3.bin.txt`: Input binary file containing the 3D dataset.
- `4 4 4`: Specifies the process grid dimensions (PX, PY, PZ).
- `64 64 64`: Specifies the full data domain size (NX, NY, NZ).
- `3`: Number of data channels per grid point.
- `output_64_64_64_3_64.txt`: Output filename.

Submitting the Job

Submit the job script using:

```
sbatch job2.sh
```

SLURM will handle node allocation and process distribution automatically. Output and error logs will be written to files named `job.<jobid>.out` and `job.<jobid>.err`, respectively.

3 Code Optimizations

3.1 Sequential I/O with Centralized Scatter

Initially, we started with the single process data reading strategy (rank 0). In this method, rank 0 reads the entire binary file data and then distributes the relevant subarrays to other processes.

At first, we considered having rank 0 send data manually to all other ranks using multiple `MPI_Isend` calls (i.e., point-to-point communication). Each destination rank would use a corresponding `MPI_Irecv` to receive its respective chunk of the global array. Although this method works functionally, it results in a large number of small, serialized messages that make the communication phase expensive and non-scalable, while at the same time rank 0 becomes a bottleneck due to the increased load of communication to all other processes.

To address this, we replaced the manual sends with a more structured and scalable approach using `MPI_Scatterv`. This was implemented in a function called `scatter_data()`, which uses `MPI_Type_create_subarray` to define a 4D block for each process, followed by packing each block using `MPI_Pack` into a large buffer. The packed buffers were then scattered using `MPI_Scatterv`, and each process unpacked its local subarray using `MPI_Unpack`.

Below is a simplified breakdown of the core logic used in this method:

- Define a subarray MPI datatype representing each process's local block using `MPI_Type_create_subarray`.
- On rank 0:
  - Allocate a global buffer and read the full dataset.
  - For each rank, compute start indices and subarray size (handling edge cases).
  - Pack each subarray using `MPI_Pack` into a shared `sendbuf`.
  - Use `MPI_Scatterv` to distribute the packed data to all ranks.
- On all ranks:
  - Receive packed data.
  - Unpack using `MPI_Unpack` into `local_data`.

While this strategy utilizes optimized MPI collective operations and structured datatypes, it suffers from the primary bottleneck of centralized I/O on rank 0. For large-scale problems, this led to an imbalance in I/O workload and poor scalability.

3.2 Optimization Strategy: Toward Parallel I/O

To address these issues, we explored and implemented multiple parallel input strategies, including:

1. **Full Dataset Replication:** Each process independently opened the input file and read the entire dataset into memory. This approach eliminated centralized I/O but resulted in excessive memory usage and redundant I/O operations, therby not improving the overall efficiency of the code.

2. **Manual Offset-Based Reads:** Each process calculated its own byte offset in the binary file and performed an independent `MPI_File_read_at` to retrieve only its local chunk. While this avoided memory duplication, it led to inefficient access patterns and poor performance due to lack of coordinated I/O or aggregation.

3. **MPI-IO with Subarray Views:** This approach used `MPI_Type_create_subarray` to define each process's logical view of the file and employed collective reads through `MPI_File_read_all`. Although this strategy is elegant and portable, it showed little to no improvement in I/O performance over the centralized rank 0 approach in our benchmarking setup.

4. **Subcommunicator-Based Grouped I/O (Optimized):** Due to the above limitations, we implemented a final strategy that significantly improved scalability and I/O efficiency. We divided the global communicator into PZ subcommunicators, each representing a horizontal slab of the 3D domain (i.e., constant Z slice). In each subcommunicator, rank 0 was designated as the reader and responsible for reading only a portion of the data corresponding to that slab using `MPI_File_read`. It then used a customized `scatter_data_subcomm()` function to distribute data to the rest of the ranks in its subcommunicator.

   This optimized approach leveraged the benefits of localized communication and reduced file access overhead by:

   - Ensuring that each rank only participated in reading or receiving its relevant slice.
   - Reducing memory pressure by avoiding full-data reads on all ranks.
   - Allowing reads and scatters to occur concurrently across slabs.

   The core logic of the optimized I/O is shown below:

```
Function: read_data_from_file_subcomm(float **data...)

Purpose:
  Each subcommunicator's rank 0 reads a portion of the binary file
  corresponding to its Z-slab (PZ layer).

Steps:
```

```
1. If last PZ slice and NZ is not divisible by PZ:
     - Adjust local_nz_actual to read fewer slices.
2. Allocate a buffer for the full slab data (NX * NY * local_nz_actual * NC).
3. Open the binary file using MPI_File_open within the subcommunicator.
4. Calculate byte offset:
     offset = pz_coord * local_nz * NX * NY * NC * sizeof(float)
5. Set the file view to start from this offset using MPI_File_set_view.
6. Read the entire slab into data using MPI_File_read.
7. Close the file.
```

Pseudocode: `scatter_data_subcomm`

```
Function: scatter_data_subcomm(float *global_slice_data, float *local_data,
                               int local_nx, int local_ny, int local_nz,
                               int num_channels, MPI_Comm comm,
                               int subrank, int subsize)

Purpose:
  Within each subcommunicator, rank 0 scatters the Z-slice to its members
  using MPI_Pack and MPI_Scatterv.

Steps:
1. Define a subarray MPI datatype for a local 4D block (Z, Y, X, C).
2. Determine packed size of each local block.
3. If subrank == 0 (leader):
   a. For each process p in the subcomm:
      i.   Calculate block start indices (start_x, start_y).
      ii.  Adjust block sizes for edge processes.
      iii. Create a subarray MPI datatype for p's block.
      iv.  Pack p's data from global_slice_data into sendbuf.
4. Each process allocates recvbuf of pack_size.
5. Use MPI_Scatterv to distribute packed blocks from leader to all.
6. Each rank unpacks its buffer into local_data using MPI_Unpack.
7. Free all buffers and datatypes.
```

This fourth method significantly outperformed the previous strategies in I/O time and total execution time, particularly as the number of processes increased.

3.3 Other Optimizations

- **Efficient Data Packing & Communication:** The program leverages **MPI subarrays** for structured **3D domain decomposition**, reducing redundant memory operations. Instead of direct data copying, `MPI_Pack`

and `MPI_Unpack` are employed to facilitate efficient message passing. Furthermore, non-blocking MPI calls help minimize synchronization overhead.

- **Optimized Halo Exchange:** Only the necessary boundaries are exchanged to reduce memory and communication overhead. The halo buffers are allocated once and reused throughout the program's execution, improving memory efficiency.
- The use of non blocking communication primitives (`MPI_Isend`, `MPI_Irecv`) improved efficiency
- Using Scatterv instead of multiple send-receives helped decrease the burden on data reader ranks.

4 Results

4.1 Scalability Analysis

To evaluate the scalability of our implementation, we conducted experiments using different numbers of processes ($np = 8, 16, 32, 64$). The execution times were measured for each configuration, and the results were averaged over multiple runs(2 in this case). **We compared different strategies of data reading and distribution to quantify the performance improvements.**

Analysis has been carried out for 5 different strategies as mentioned below:

1. **Single-Rank Input (Centralized I/O):** In this baseline approach, rank 0 is responsible for reading the entire dataset from file. It then distributes relevant subarrays to all other ranks using `MPI_Pack` and `MPI_Scatterv`.
2. **Full Dataset Replication:** Each process independently opened the input file and read the entire dataset into its local memory. Then takes out it's portion of the dataset.
3. **Manual Offset-Based Reads:** To reduce memory usage, each rank computes its own byte offset in the binary file and performs a point-to-point read using `MPI_File_read_at`. This ensured that each rank only read its own chunk of data.
4. **MPI-I/O with Subarray :** In this method, we leveraged `MPI_Type_create_subarray` to describe the local view of the global dataset for each rank. A collective read was then performed using `MPI_File_read_all`, allowing the MPI library to optimize data access internally.
5. **Subcommunicator-Based Grouped I/O (Optimized):** A hybrid I/O strategy using MPI subcommunicators was implemented. The global communicator was split along the Z-axis into P_Z subcommunicators, each responsible for a horizontal slab of the 3D volume. Within each subcomm,

rank 0 acted as a local leader and read the entire slab corresponding to its group using `MPI_File_read`. The leader then distributed data to its group members using scatterv.

Each method was implemented, validated for correctness, and benchmarked to evaluate data read time, compute time and overall execution time, and scalability across different problem sizes and process counts. The fifth method was observed to perform best under our experimental conditions and was ultimately adopted for the final implementation(The codes for all the methods are present in a seperate folder named experimentation).

4.1.1 Execution Time Analysis with number of processes

Section 4.1.2 contains Table 1 and Table 2 presents the split timings for different numbers of processes using using Sub-communicator based group Input(Optimized Parallel Input). Section 4.1.3 contains Table 3 and Table 4 presents the split timings for different numbers of processes using Single Rank input. The total execution time is divided into three components:

- **Initial Phase** ($T_2 - T_1$): Data reading and distribution.
- **Main Computation** ($T_3 - T_2$): Core computation, including halo exchange and local extrema detection.
- **Total Execution Time** ($T_3 - T_1$): Overall runtime.

4.1.2 Subcommunicator-Based Grouped I/O (Optimized)

Table 1: For the data: data_64_64_64_3.bin.txt

Number of Processes (np)	**Initial Phase** ($T_2 - T_1$)	**Main Computation** ($T_3 - T_2$)	**Total Time** ($T_3 - T_1$)
8	0.00872550 s	0.0132755 s	0.022001 s
16	0.008922 s	0.006986 s	0.0159085 s
32	0.0096015 s	0.0052105 s	0.014812 s
64	0.010635 s	0.002421 s	0.0130555 s

Table 2: For the data: data_64_64_96_7.bin.txt

Number of Processes (np)	**Initial Phase** ($T_2 - T_1$)	**Main Computation** ($T_3 - T_2$)	**Total Time** ($T_3 - T_1$)
8	0.015048 s	0.0362015 s	0.0512495 s
16	0.015343 s	0.0184855 s	0.033829 s
32	0.0162885 s	0.0112265 s	0.0275155 s
64	0.020578 s	0.0063415 s	0.0269195 s

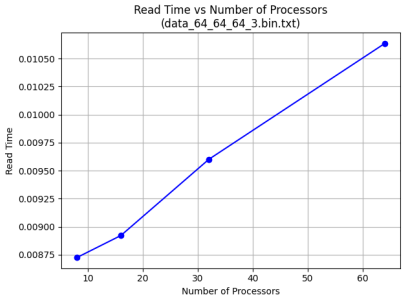

Figure 1: First image

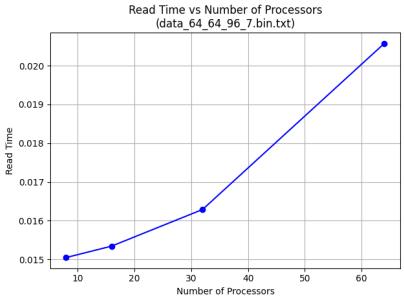

Figure 2: Second image

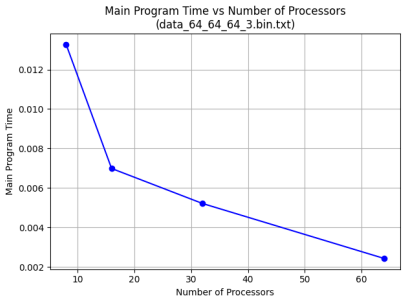

Figure 3: First image

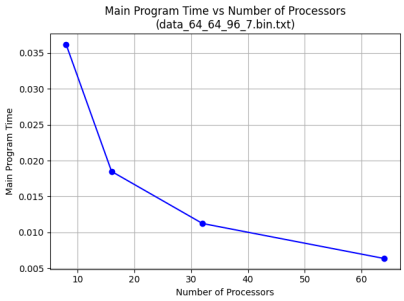

Figure 4: Second image

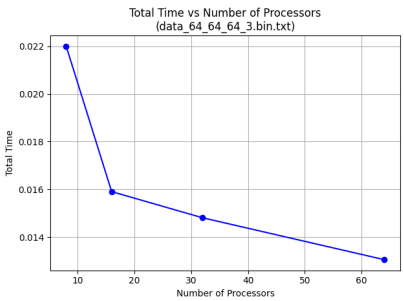

Figure 5: First image

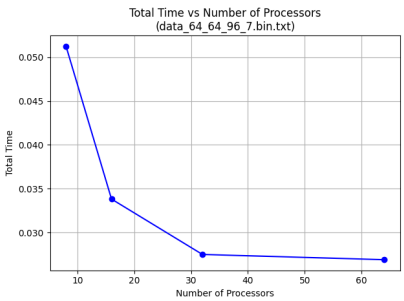

Figure 6: Second image

Scaling Observations on SLURM Cluster An important observation from our experiments relates to how read time scales with the number of processes. For process counts less than or equal to 50, the read time remains relatively low and stable. However, a noticeable jump in read time is observed when scaling up to 64 processes. This is primarily due to SLURM's constraint where a single compute node can host a maximum of 50 MPI processes. When the process

count exceeds this threshold, the job is distributed across multiple nodes, introducing inter-node communication. Since intra-node communication (shared memory) is significantly faster than inter-node communication (which involves network overhead), performance begins to degrade due to increased latency and communication cost across nodes.

4.1.3 Single-Rank Input (Centralized I/O)

Table 3: Execution time breakdown for different process counts.

Number of Processes (np)	**Initial Phase** $(T_2 - T_1)$	**Main Computation** $(T_3 - T_2)$	**Total Time** $(T_3 - T_1)$
8	0.010604 s	0.013574 s	0.024177 s
16	0.176951 s	0.007378 s	0.184331 s
32	0.518679 s	0.005515 s	0.524194 s
64	1.221621 s	0.089022 s	1.310643 s

Table 4: Execution time breakdown for different process counts.

Number of Processes (np)	**Initial Phase** $(T_2 - T_1)$	**Main Computation** $(T_3 - T_2)$	**Total Time** $(T_3 - T_1)$
8	0.018829 s	0.037613 s	0.056442 s
16	0.175557 s	0.018873 s	0.194429 s
32	0.560058 s	0.010368 s	0.570426 s
64	1.202659 s	0.125399 s	1.328057 s

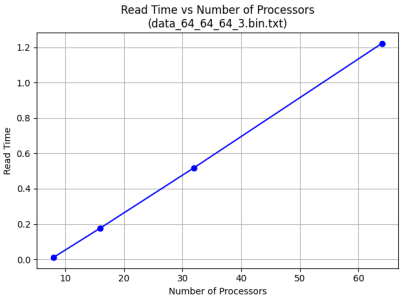

Figure 7: First image

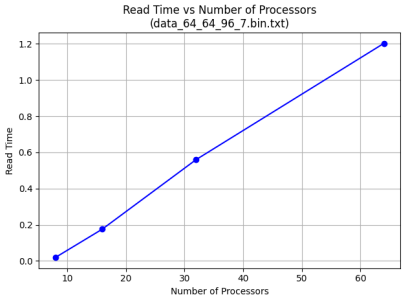

Figure 8: Second image

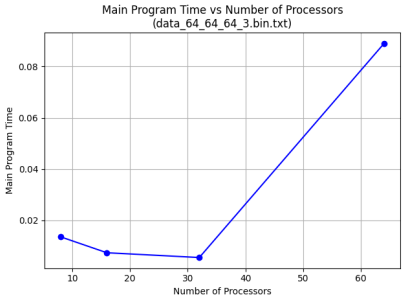

Figure 9: First image

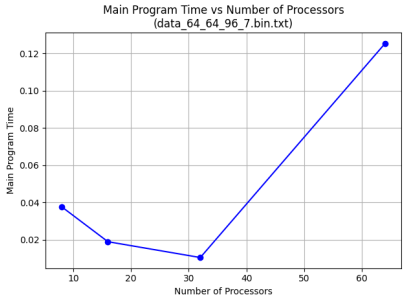

Figure 10: Second image

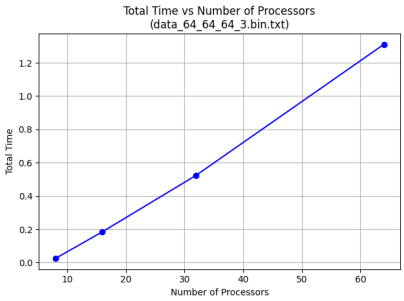

Figure 11: First image

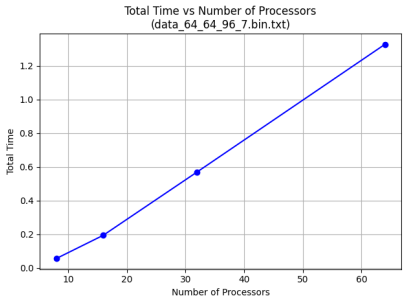

Figure 12: Second image

Scaling Observations on SLURM Cluster In our benchmarking experiments, we observed that the total execution time remained relatively low for process counts up to 32. However, there was a significant increase in execution time when scaling to 64 processes. This behavior is closely tied to the architecture of the SLURM-based HPC cluster, where were allowed a maximum of 50 processes per node. Beyond this limit, MPI processes were distributed across multiple nodes, triggering inter-node communication which is considerably more expensive than intra-node communication.

In the case of sequential input (centralized I/O), rank 0 reads the entire dataset and scatters the relevant portions to other ranks, which might need to inter-node communication in case of 64 nodes.

Similarly, during halo exchange, each rank communicates boundary layers with its six spatial neighbors. When all processes are within the same node, this halo communication uses shared memory and benefits from low-latency access. However, with 64 processes spread across multiple nodes, some halo exchanges involve inter-node communication, which relies on slower network links. Consequently, the communication overhead during halo exchange increases sharply, adding to the overall time even though the data read step remains centralized.

4.1.4 Impact of Parallel I/O Optimization

Figure 13, 14 and 15 compare the timings of the 5 different Input reading and distribution strategies.

Graph Legend

The following keys correspond to the I/O strategies presented in the performance graphs:

- **Sequential Read:** Single-Rank Input (Centralized I/O)
- **Parallel Input 1:** Full Dataset Replication
- **Parallel Input 2:** Manual Offset-Based Reads
- **Parallel Input 3:** MPI-I/O with Subarray
- **Parallel Input 4:** Subcommunicator-Based Grouped I/O (Optimized)

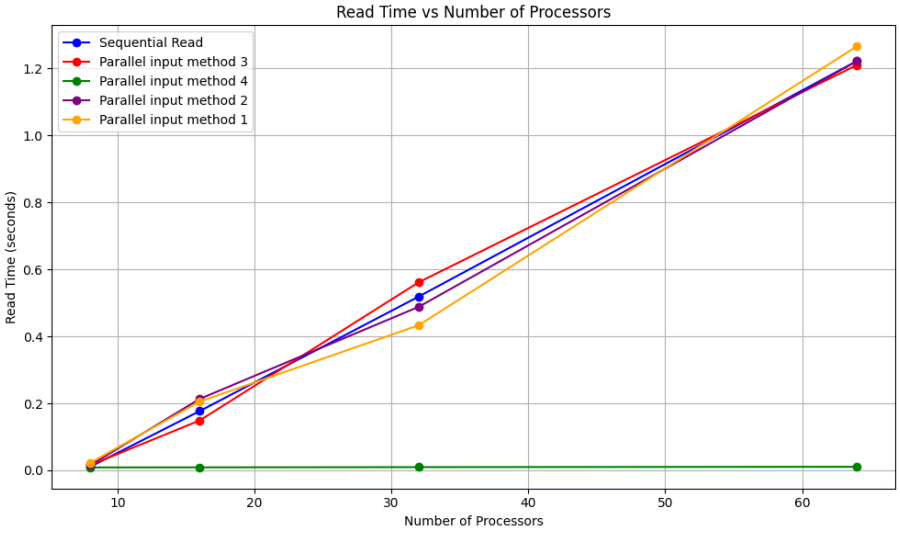

Figure 13

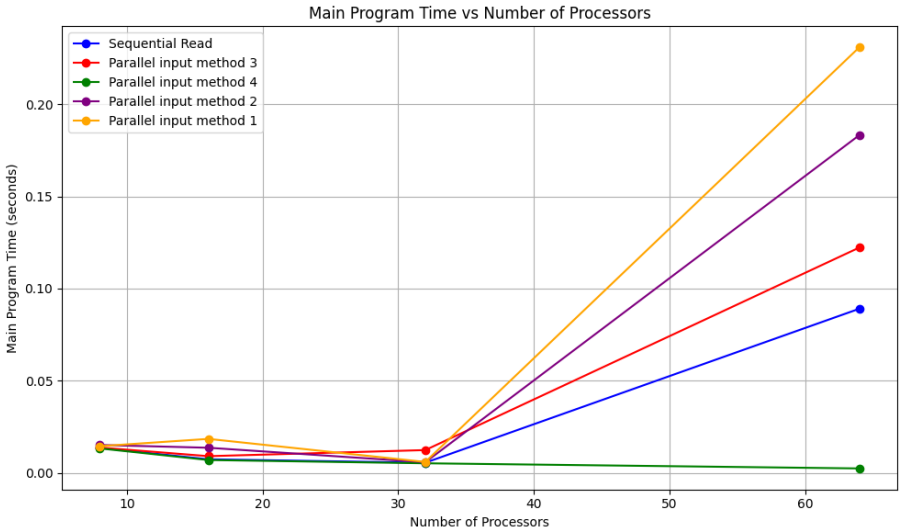

Figure 14

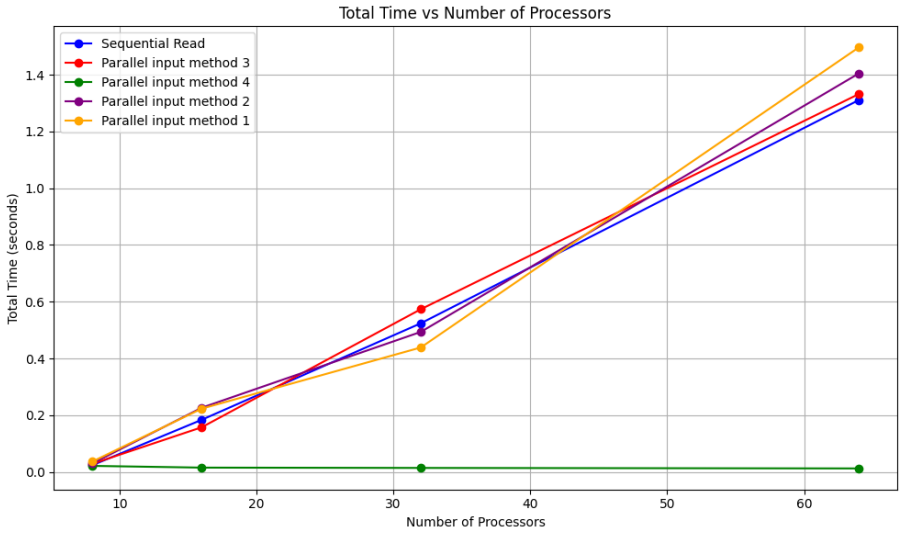

Figure 15

The optimized parallel input approach, which utilizes group leaders and sub-communicators, delivers a substantial performance improvement. It significantly reduces data reading and distribution time, leading to a marked decrease in the overall execution time. This methodology effectively exploits the structure of

the input data by forming sub-communicators that align with the locality of process-to-data mapping, thereby enhancing data access efficiency.

4.1.5 Speedup and Efficiency

To further assess scalability, we compute the **speedup** S_p and **parallel efficiency** E_p:

$$S_p = \frac{T_1}{T_p}, \quad E_p = \frac{S_p}{p} \times 100\% \tag{1}$$

where T_1 is the execution time for a single process, and T_p is the execution time for p processes. Table 5 summarizes these results.

Table 5: Speedup and parallel efficiency analysis for data_64_64_96_7.bin.txt using src.c (most optimal code)

Number of Processes (np)	**Speedup (S_p)**	**Efficiency (E_p)**
8	5.48	68.50%
16	8.21	51.30%
32	9.97	31.17%
64	14.43	22.55%

4.2 Performance Bottlenecks

To understand the potential inefficiencies in data distribution, we compare traditional approaches with our optimized strategy that uses subcommunicators and I/O leaders.

4.2.1 Comparison with Single Process Read and Global Scatter

Traditional Approach:

- **Method:** Rank 0 reads the entire file and distributes the data using either `MPI_Scatterv` or point-to-point messages.
- **Limitations:** This creates a significant I/O bottleneck and induces high network congestion due to long-distance communication.

Subcomm + Leader Approach Advantages:

- Distributes the I/O load across multiple leader processes.
- Localizes communication within subcommunicators, which reduces network overhead.
- Improves scalability and enhances file system performance.

4.2.2 Comparison with Each Process Reading Independently

Traditional Approach:

- **Method:** Each process independently opens the file and reads its designated data block.
- **Limitations:** This method leads to a high number of file opens, uncoordinated I/O, and increased file system contention.

Subcomm + Leader Approach Advantages:

- Reduces the number of file opens to one per I/O slice.
- Supports larger, more efficient MPI I/O operations.
- Utilizes collective I/O optimizations to achieve improved performance on parallel file systems.

4.3 Observations and Analysis

Key Observations:

- **Sequential Read (Centralized I/O):**
  - **Trend:** Read time increases significantly with more processes, due to the centralization of I/O at rank 0.
  - **Limitation:** Creates an imbalance in resource utilization and heavy network traffic for subsequent data scattering.
- **Parallel Input 1 (Full Dataset Replication):**
  - **Trend:** Offers improved read times at intermediate process counts; however, performance benefits taper off at larger scales.
  - **Limitation:** High memory overhead and file system contention from replicating data across all processes.
- **Parallel Input 2 (Manual Offset-Based Reads):**
  - **Trend:** Each process reading only its needed block provides better performance, though gains plateau at larger scales due to file system contention.
  - **Limitation:** Complexity in manually calculating and synchronizing file offsets.
- **Parallel Input 3 (MPI-I/O with Subarray):**
  - **Trend:** Collective I/O via subarray descriptors results in more efficient scaling and reduced redundant data access.
  - **Strength:** Simplifies data partitioning and leverages collective I/O optimizations.

- **Parallel Input 4 (Subcommunicator-Based Grouped I/O, Optimized):**
  - **Trend:** Exhibits the most favorable scalability by maintaining low read times even at high process counts.
  - **Strength:** By using subcommunicators and limiting direct file access to leaders, this method reduces network congestion and file system contention.

In-depth Analysis:

- **Bottleneck Reduction:** Transitioning from a centralized rank 0 I/O model to a distributed approach using grouped I/O mitigates major bottlenecks. Localized I/O within subcommunicators minimizes both the I/O load and network overhead.
- **Scalability and Efficiency:** The MPI-I/O with subarray method facilitates better scalability through collective I/O, yet the subcommunicator-based approach further enhances performance by reducing file opens and direct file system interactions.
- **Impact at High Process Counts:** At scales such as 32 or 64 processes, optimized grouped I/O effectively curtails the "thundering herd" effect. Although some performance degradation occurs due to necessary communication steps (e.g., halo exchanges), the overall trends remain significantly more favorable.
- **Implementation Complexity vs. Performance Gains:** Implementing the Subcomm + Leader approach requires additional coordination (e.g., subcommunicator creation), but the resulting performance improvements—especially for large-scale systems—justify the extra complexity.
- **Future Enhancements:** Continued enhancements could focus on overlapping communication with computation to further mitigate overhead in MPI operations. Moreover, developing adaptive I/O strategies that adjust subcommunicator sizes based on real-time file system load could push performance even further.

4.4 Summary

The scalability analysis demonstrates that the optimized version with parallel I/O significantly outperforms the Single-Rank Input implementation. The results suggest that the parallelization strategy which is aware of the process mapping is effective.

5 Conclusions

5.1 Final Remarks

This project aimed to efficiently compute local and global extrema over a 4D spatio-temporal dataset using parallel computing paradigms. Through a combination of domain decomposition, halo exchange, and optimized input strategies—including both centralized and distributed I/O—we achieved scalable solutions suitable for large data volumes. . Multiple I/O approaches were explored and benchmarked, leading to valuable insights on trade-offs between simplicity and performance.

5.2 Work Distribution

To ensure efficient progress and clear task ownership, responsibilities were divided among team members as follows:

- **Data Input and Distribution**
  - **Sequential I/O:** Mridul Pandey, Suryansh Goel, Keshav Raj Gupta, Rudransh Goel, Mehar Goenka
  - **Parallel I/O:** Mridul Pandey, Suryansh Goel
- **Halo Exchange:** Mridul Pandey, Mehar Goenka, Rudransh Goel
- **Local and Global Minima/Maxima Computation:** Keshav Raj Gupta, Mehar Goenka, Rudransh Goel, Mridul Pandey, Suryansh Goel
- **Job Scheduling and running:** Suryansh Goel, Keshav Raj Gupta
- **Report Writing:** Mridul Pandey, Rudransh Goel, Mehar Goenka, Suryansh Goel