CS633 Assignment Group Number 24

Aaditi Anil Agrawal 220006
Ritesh Baviskar 220286
Aditya Jagdale 220470
Mahaarajan J 220600
Wattamwar Akanksha Balaji 221214

1 Code Description

Code Description

Our implementation to solve the given problem focuses on efficiently finding local minima, local maxima, global minimum, and global maximum in a 3D volume time-series dataset using MPI for parallel processing. The code employs a 3D domain decomposition strategy where the entire volume is partitioned equally among processes along the X, Y, and Z dimensions. Each process is assigned a unique 3D subdomain (containing data for all time steps) based on its rank.

Key Components

1. **Derived Datatypes**: We make extensive use of `MPI` derived datatypes to simplify data handling and communication. Two key custom datatypes are:

   - `time_block_type`: Created using `MPI_Type_contiguous`, this datatype represents a block of `NC` contiguous values across all time steps for a single grid point. This layout matches the input file structure, where values for all time steps of a point are stored together.
   - `subarray_type`: Defined using `MPI_Type_create_subarray`, this datatype allows each process to define a logical view of its subdomain within the global 3D volume, enabling efficient parallel I/O and communications.

2. **Leader-Based I/O**: Instead of relying on a single root process (rank 0) to perform file I/O and distribute data, we implemented a leader-based I/O strategy using `MPI_File` functions. In this approach, a subset of processes—specifically, one leader per plane along the Z-dimension (i.e., PZ leaders)—are designated to read distinct portions of the input file in parallel. Each leader then redistributes the data to the processes in its corresponding group.

   This method balances the benefits of both serial and fully parallel I/O:

   - It alleviates the bottleneck caused by centralized I/O by enabling concurrent reading from multiple leaders.
   - It avoids the file system contention and scalability issues that may arise when all processes access the file simultaneously in a fully parallel I/O model.

   Overall, leader-based I/O achieves more scalable and efficient data distribution, particularly on systems where full parallel file access is either limited or suboptimal.

3. **Ghost Cell Exchange**: To accurately detect local extrema at subdomain boundaries, each process requires information from its immediate neighbors in the six Cartesian directions: left, right, top, bottom, front, and back. For processes located at the global domain boundaries, we use `MPI_PROC_NULL` to safely disable communication in non-existent directions.

   The boundary data exchanged corresponds to 2D planes along each face of the 3D subdomain. To facilitate efficient communication, we utilize derived datatypes to represent these ghost planes, avoiding the overhead of manually packing and unpacking data. Non-blocking MPI operations (`MPI_Isend`, `MPI_Irecv`) are employed

to initiate halo exchanges in all directions concurrently, thereby overlapping communication with computation and reducing idle time.

Each process stores both its local domain data and the corresponding ghost planes in a structured format. The following structure is used to encapsulate the data layout for each process:

```
typedef struct {
    float *local_data;      // Pointer to the local subdomain data
    float *left_ghost;      // Ghost plane from neighbor in -X direction (YZ plane)
    float *right_ghost;     // Ghost plane from neighbor in +X direction (YZ plane)
    float *bottom_ghost;    // Ghost plane from neighbor in -Y direction (XZ plane)
    float *top_ghost;       // Ghost plane from neighbor in +Y direction (XZ plane)
    float *back_ghost;      // Ghost plane from neighbor in -Z direction (XY plane)
    float *front_ghost;     // Ghost plane from neighbor in +Z direction (XY plane)
    int local_nx, local_ny, local_nz, NC;  // Local domain dimensions and number of time steps
} DomainData;
```

This structured approach simplifies access to ghost cell data during computation and enables modular, clean handling of communication buffers.

4. **Extrema Detection**: Each process iterates over its subdomain to identify local minima and maxima. For local extrema detection, values are compared to the six direct neighbors using the local domain and the ghost planes. A point is classified as:

   - A *local minimum* if its value is strictly less than all its neighbors.
   - A *local maximum* if its value is strictly greater than all its neighbors.

   At domain boundaries, where neighboring data is absent, sentinel values such as `FLT_MAX` (for local minimum checks) and `-FLT_MAX` (for local maximum checks) are used to ensure correctness. For global maxima the process simply iterates over its subdomain and reports the least and greatest value.

5. **Global Reduction**: The code uses `MPI_Reduce` operations to:

   - Sum local extrema counts across all processes for each time step
   - Find global minimum and maximum values for each time step

6. **Performance Measurement**: The implementation times the execution of three phases to analyze performance:

   - File reading and data distribution.
   - Main computation (including ghost cell exchange and extrema detection).
   - Total execution time.

7. **Output Writing**: Since the final output is minimal (only global extrema and counts), we delegate writing to file solely to process 0. After performing reductions, process 0 formats and writes the results to the output file.

2 Code Compilation and Execution Instructions

To execute the final submission code `final_submission.c`, follow these steps:

1. Compile the C file using the command:

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

2. To run the test cases, use the following scripts:

```
sbatch job_8_64_64_64_3.sh
sbatch job_16_64_64_64_3.sh
sbatch job_32_64_64_64_3.sh
sbatch job_64_64_64_64_3.sh
```

This will run the script on the smaller dataset and produce the output files `output_64_64_64_3_8.txt`, `output_64_64_64_3_16.txt`, `output_64_64_64_3_32.txt` and `output_64_64_64_3_64.txt` respectively. Similarly to run the test cases for the larger dataset, use these scripts:

```
sbatch job_8_64_64_96_7.sh
sbatch job_16_64_64_96_7.sh
sbatch job_32_64_64_96_7.sh
sbatch job_64_64_64_96_7.sh
```

This will run the script on the smaller dataset and produce the output files `output_64_64_96_7_8.txt`, `output_64_64_96_7_16.txt`, `output_64_64_96_7_32.txt`, `output_64_64_96_7_64.txt` respectively.

3. To make the plots as reported, follow the following steps. For the dataset `data_64_64_64_3.bin.txt`,

```
cd 64_64_64_3
sbatch job_64_64_64_3.sh
python script.py
python plots.py
```

Similarly, for the dataset `data_64_64_96_7.bin.txt`,

```
cd 64_64_96_7
sbatch job_64_64_96_7.sh
python script.py
python plots.py
```

Make sure to have python installed along with the libraries matplotlib and pandas.

3 Code Optimizations

Throughout the development of our implementation, we identified and addressed several performance bottlenecks related to data communication, I/O, and memory usage.

Initial Implementation and Bottlenecks

Our initial implementation relied on a centralized I/O approach where only rank 0 read the entire dataset and distributed it to all other processes using point-to-point communication (i.e., `MPI_Send` and `MPI_Recv`). This design introduced two major bottlenecks:

- Rank 0 became a communication bottleneck, as it had to send large chunks of data to every other process.
- Reading the input file line-by-line, number-by-number in text format (instead of binary) resulted in significant overhead during parsing.

Text File I/O Optimization

To mitigate I/O inefficiency, we modified the file reading strategy. Instead of reading one number at a time, we optimized the parser to read an entire line of data at once, reducing the number of I/O calls and improving cache performance. While this did not fully resolve the I/O bottleneck, it yielded a noticeable speedup in the serial version.

Use of Derived Datatypes

To further improve performance, we leveraged MPI's support for derived datatypes:

- `MPI_Type_contiguous` was used to define a datatype representing values of a single grid point across all time steps, minimizing the need for intermediate buffers during communication.
- `MPI_Type_create_subarray` allowed each process to define its own logical view into the global 3D array, enabling clean and efficient data access during I/O and computation.

This removed the need for manual packing and unpacking of buffers and improved code readability while slightly reducing memory overhead and copying time.

Ghost Cell Communication Optimization

In the next phase, we applied derived datatypes to ghost cell exchanges. Instead of copying 2D boundary planes into temporary buffers before communication, we directly communicated using custom datatypes representing each boundary face giving a slight gain in performance

Binary File Format Optimization

One of the key changes we introduced was shifting from reading data in text format to reading from a binary file. Initially, our implementation parsed the dataset line by line as plain text, which introduced substantial overhead due to frequent string parsing, format conversion (e.g., from ASCII to float), and non-contiguous memory access.

By switching to a binary file format, we were able to read raw data directly into memory buffers without any intermediate parsing. This allowed us to create an efficient code which performed significantly better than the earlier versions even though we were using serial IO

Parallel I/O

Each process independently read its portion of the dataset from the input file using `MPI_File_set_view` and `MPI_File_read_all`. This eliminated the need for communication at rank 0 and allowed for concurrent file access. However the gain was pretty insignificant possibly due to the small datasize of the file or due to increased file contention as numerous processes were trying to read the file at the same time.

Leader-Based I/O

To balance the trade-offs between serial and fully parallel I/O, we introduced a leader-based I/O scheme as described in the earlier section, where one process per Z-plane reads a portion of the data and distributes it to nearby processes. This reduces contention while maintaining parallel efficiency.

Structure-Based Memory Optimization

In earlier versions, each process allocated an extended buffer that included both local data and ghost cells. This required explicitly copying local data into the center of the extended buffer before inserting ghost values. This approach was both memory- and computation-intensive.

To avoid this overhead, we introduced a structured layout where local data and each ghost plane are maintained as separate pointers in a single `DomainData` structure. This design avoids unnecessary memory copying, reduces overall memory usage, and simplifies the access pattern for neighbor comparisons. It also enhances cache performance and makes ghost handling more modular and readable.

Summary

Our optimizations followed an incremental refinement approach—starting with reducing I/O parsing overhead, then improving data communication via derived datatypes, and finally transitioning to a fully parallel I/O model. Each of these changes contributed to reducing execution time, improving memory efficiency, and enhancing scalability of the implementation.

4 Results

We experiment with the 2 datafiles provided on Helloiitk, namely `data_64_64_64_3.bin.txt` and `data_64_64_96_7.bin.txt`. Among the various methods, we choose three methods for comparison :

1. **Serial I/O**, where communication between processes is point-to-point and rank 0 reads the entire data at once and distributes it using simple send other processes receive their data using simple receive.
2. **Parallel I/O**, where each process reads its portion of data and communication in intermediate steps is using derived datatypes.
3. **Leader-based I/O**, with intermediate communications done using derived datatypes.

4.1 Number local optimas and global optima

Here are the final results we got for both datasets. `data_64_64_96_7.bin.txt`

Time Step	Local Minima / Maxima	Global Min / Max
0	(56875, 56848)	(-48.250000, 33.630001)
1	(56703, 56965)	(-51.450001, 33.349998)
2	(56680, 56847)	(-48.549999, 33.349998)
3	(56601, 56980)	(-43.130001, 32.000000)
4	(56769, 56937)	(-53.549999, 34.060001)
5	(56657, 56950)	(-49.680000, 34.180000)
6	(56862, 56996)	(-53.549999, 34.340000)

`data_64_64_64_3.bin.txt`

Time Step	Local Minima / Maxima	Global Min / Max
0	(37988, 37991)	(-48.250000, 33.630001)
1	(37826, 38005)	(-51.450001, 33.349998)
2	(37788, 37978)	(-48.549999, 33.349998)

4.2 Timing Results

We run each of these methods five times for each configuration and report the average values of timings in these runs. Time-1 represents the read time, Time-2 represents the Main code time and Time-3 represents the Total time.

Method	Number of processes	Time-1	Time-2	Time-3
Leader(final)	8	**0.0096564**	0.0081362	**0.0150514**
	16	0.0097030	0.0092248	0.0152626
	32	0.5069848	0.4980782	0.5092136
	64	1.2558448	1.3763858	1.3828812
Parallel	8	0.0126532	0.0039384	0.0165888
	16	0.0144918	0.0023358	0.0168252
	32	0.5376102	0.0016538	0.5392520
	64	1.5416764	**0.0010628**	1.5425326
Serial	8	0.0132060	0.0130336	0.0179364
	16	0.0148214	0.0134424	0.0175736
	32	0.5099982	0.4970824	0.5135150
	64	1.3010000	1.2001830	1.3850616

Table 1: For each configuration, five runs were done and the average of these five values are reported above. The values in bold represents the best time for each section(Read time, main code time and total time. This run was done on `data_64_64_64_3.txt`

Method	Number of processes	Time-1	Time-2	Time-3
Leader(final)	8	0.0184448	0.0127674	0.0297844
	16	0.0204416	0.0142028	0.0266372
	32	0.5122536	0.5130216	0.5236422
	64	1.3078070	1.5541770	1.5660092
Parallel	8	0.0197538	0.0134392	0.0334526
	16	0.0213184	0.0073346	0.0289054
	32	0.4947442	0.0045576	0.4995028
	64	1.4281368	**0.0044188**	1.4321778
Serial	8	**0.0127054**	0.0126726	0.0178470
	16	0.0141760	0.0127020	**0.0174244**
	32	0.5266730	0.5254130	0.5299070
	64	1.2950060	1.2740074	1.4599230

Table 2: For each configuration, five runs were done and the average of these five values are reported above. The values in bold represents the best time for each section(Read time, main code time and total time. This run was done of the `data_64_64_96_7.txt`

4.3 Scalability graphs

These are the visual representations of the results obtained.

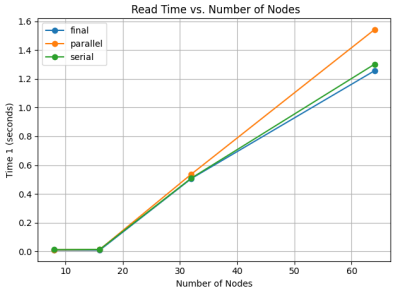

Figure 1: Plot of average times for reading(Time-1)

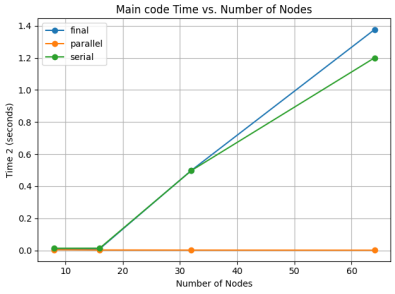

Figure 2: Plot of average times for main code(Time-2)

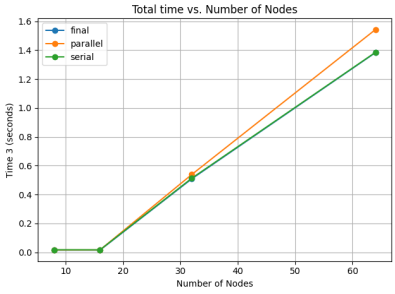

Figure 3: Plot of average times for total times(Time-3)

Figure 4: Results for `data_64_64_64_3.txt`

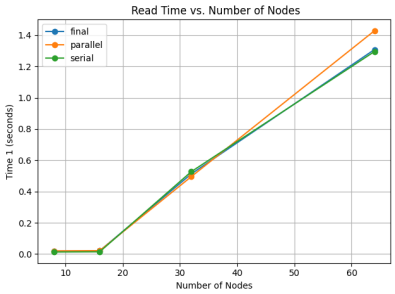

Figure 5: Plot of average times for reading(Time-1)

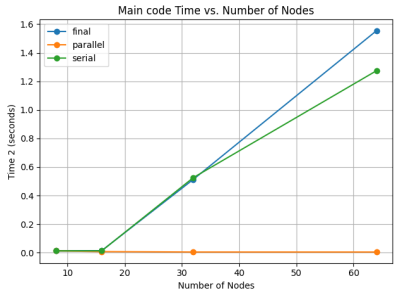

Figure 6: Plot of average times for main code(Time-2)

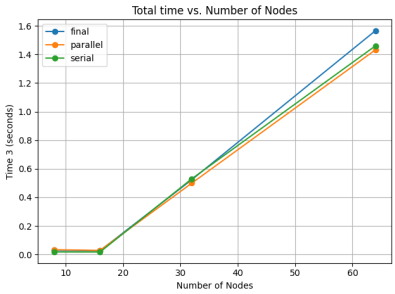

Figure 7: Plot of average times for total times(Time-3)

Figure 8: Results for `data_64_64_96_7.bin.txt`

4.4 Analysis of the Results

We analyzed the performance of three different implementations for reading and processing the data: a purely parallel version, a serial version, and a leader-based approach. The metrics considered include data scalability, process scalability, file read time, compute time, and overall execution time.

1. Data Scalability

We observed that the total execution time did not vary significantly between the smaller and larger datasets. This suggests that at the current dataset sizes, the execution time is dominated by fixed overheads such as MPI initialization, communication latency, and process synchronization. As a result, the actual data size does not significantly influence performance. We expect that this trend may change with larger datasets where the volume of data starts to dominate computation and communication.

2. Process Scalability

Contrary to the usual expectation, increasing the number of processes led to an increase in total execution time. This is primarily due to:

- Increased inter-process communication during ghost cell exchange.
- Higher contention and potential clashes during parallel file access.
- Overhead of data distribution from rank 0 (in the serial case) or leader processes.

The dataset is relatively small, and the cost of communication outweighs the computational savings achieved via domain decomposition. Therefore, scalability in terms of processes is not favorable in this case.

3. File Read Time

Among the three approaches, the leader-based I/O strategy resulted in the best read times. This approach balances the advantages of both parallel and serial file access:

- It avoids the bottleneck of a single process reading the entire file (as in the serial case).
- It reduces contention and overhead associated with all processes reading simultaneously (as in the fully parallel approach).
- Leaders (a subset of processes) read non-overlapping portions of the file in parallel and distribute data locally, resulting in efficient and scalable file access.

4. Compute Time

The compute time was surprisingly the lowest for the fully parallel implementation—significantly lower than both the serial and leader-based versions. While the exact reason is unclear, we hypothesize the following:

- The state of the MPI network after parallel file reads might favor subsequent communication patterns, reducing setup overheads for ghost cell exchange.
- There could be hidden synchronization effects where certain processes wait less in the parallel configuration.
- The overhead of intermediate data copying or buffer reuse might differ across implementations.

Further profiling and instrumentation would be needed to conclusively identify the reason behind this disparity.

5. Overall Execution Time

Overall execution times across the three approaches are relatively close, with the leader-based implementation slightly outperforming the others in some runs. However, due to differences in the read and compute strategies, their internal performance dynamics differ. A more comprehensive evaluation with varied dataset sizes and process counts would provide better insights into the most scalable and robust implementation.

In conclusion, while the leader-based approach provides a balanced and efficient method, each method has its trade-offs. Selecting the best method would depend on the specific use case, dataset size, and available computational resources.

5 Conclusions

This project gave us hands-on experience with parallel computation using MPI, focusing on domain decomposition, communication optimization, and performance-aware design. We compared multiple implementations, from serial to fully parallel and leader-based approaches, analyzing their performance across different data sizes and process counts.

We found that while parallel compute significantly reduces computation time, communication overhead and I/O strategy play a critical role in overall efficiency. The leader-based approach struck the best balance between performance and simplicity. Optimizations like contiguous memory layouts and derived datatypes had measurable impacts on runtime and memory traffic.

Overall, the assignment deepened our understanding of distributed memory programming and highlighted the value of thoughtful design in high-performance computing.

Group Members' Contribution

All five members contributed equally to the assignment. Code development was done collaboratively, either together or by dividing tasks and reviewing each other's work. Design discussions, debugging, testing, and performance analysis were shared responsibilities. The report was co-written, with everyone contributing to different sections and reviewing the final document together.