CS633 Assignment Group Number 43

Akshit Shukrawal	220107
Ayush Meena	220268
Mohammed Anas	220654
Aman Yadav	220121

13-04-2025

1 Code Description

1.1 Introduction

The provided code is a parallel program utilizing the Message Passing Interface (MPI) for analyzing a three-dimensional dataset. The program processes data on a 3D grid partitioned among multiple processes and identifies local minima and maxima within the data.

1.2 Program Structure

1.2.1 Data Decomposition

The 3D dataset of dimensions $NX \times NY \times NZ$ with NC components per grid point is decomposed across $PX \times PY \times PZ$ processes, creating a 3D process grid. Each process is responsible for a subgrid of size $\frac{NX}{PX} \times \frac{NY}{PY} \times \frac{NZ}{PZ}$ with NC components.

1.2.2 Algorithm

The program follows these key steps:

1. Read input data from file using parallel I/O
2. Exchange boundary data with neighboring processes
3. Identify local minima and maxima in the dataset
4. Reduce local results to obtain global statistics
5. Write results to output file
6. Measure and report performance metrics

1.3 Key Components

1.4 Data Distribution

Each process reads its assigned portion of the 3D data directly from the input file using MPI file operations with proper offset calculations:

$$\text{offset} = z\_\text{rank} \cdot \text{sub\_nz} \cdot NY \cdot NX \cdot NC + y\_\text{rank} \cdot \text{sub\_ny} \cdot NX \cdot NC + x\_\text{rank} \cdot \text{sub\_nx} \cdot NC \quad (1)$$

1.5 Boundary Exchange

The program establishes a ghost region around each process's subgrid by exchanging boundary data with neighboring processes in all six directions (left, right, top, bottom, front, back) using non-blocking MPI communications:

- Custom MPI derived datatypes are created for efficient communication along x and y dimensions
- Non-blocking sends and receives (`MPI_Isend`, `MPI_Irecv`) are used to allow overlap of communication

1.6 Local Extrema Detection

A point is considered a local minimum or maximum if its value is strictly less than or greater than all its neighbors along the three coordinate axes. The algorithm:

- Examines each interior point and its six neighbors (± 1 in each dimension)
- Counts points that are local minima or maxima for each component
- Tracks the global minimum and maximum values for each component

1.6.1 Global Reduction

MPI collective operations reduce local results to global statistics:

- `MPI_Reduce` with `MPI_MIN` and `MPI_MAX` operations for minimum and maximum values
- `MPI_Reduce` with `MPI_SUM` operation for counting local extrema

1.7 Performance Measurement

The program measures three timing intervals:

- t_1: Time for reading input data
- t_2: Time for computation (boundary exchange and extrema detection)
- t_3: Total execution time (reading time + main code + printing result)

1.8 Output Format

The program writes to the output file:

- For each component: $(count\_min, count\_max)$ pairs representing the number of local minima and maxima
- For each component: (min_value, max_value) pairs representing the global minimum and maximum values
- Performance metrics: t_1, t_2, t_3

1.9 Technical Challenges

The implementation addresses several parallel computing challenges:

- Proper domain decomposition across processes
- Efficient parallel I/O for data loading
- Management of inter-process communication for boundary exchange
- Correct handling of global boundaries
- Efficient global reduction operations

2 Code Compilation and Execution Instructions

2.1 Code Compilation

The code can be compiled using the following command:

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

This will produce an executable named `src` in the current directory.

2.2 Code Execution

The compiled executable can be run using the following command:

```
mpirun -np <num_processes> -f hostile ./src data_NX_NY_NZ_NC.bin.txt PX PY PZ NX NY NZ NC
    output_NX_NY_NZ_NC.txt
```

- `<num_processes>`: Total number of MPI processes to launch. This must equal `PX × PY × PZ`.
- `-f hostile`: Specifies the hostfile containing the list of machines for distributed execution.
- `data_NX_NY_NZ_NC.bin.txt`: Input file containing the 3D volume data.
- `PX, PY, PZ`: Number of process partitions along the X, Y, and Z dimensions respectively.
- `NX, NY, NZ`: Global size of the 3D data grid in the X, Y, and Z directions.
- `NC`: Number of time steps (channels) in the data.
- `output_NX_NY_NZ_NC.txt`: Output file to store the final results.

3 Code Optimizations

To enhance both computational efficiency and I/O performance in our MPI-based 3D stencil application for local extrema detection, several optimizations have been implemented:

1. **Use of `MPI_Type_vector` for Efficient Ghost Layer Exchange:**
   Custom MPI datatypes, defined with `MPI_Type_vector`, allow for efficient exchange of ghost layers along non-contiguous memory regions. This eliminates the need for manual buffer packing/unpacking and reduces overhead in data transfers.

2. **Non-blocking Communication:**
   The usage of `MPI_Isend` and `MPI_Irecv` facilitates overlapping of computation with communication. This non-blocking approach hides latency and improves the overall scalability of the application.

3. **In-place Local Extrema Detection:**
   The detection algorithm considers only the 6 face neighbors for each voxel. This streamlined approach avoids unnecessary comparisons, significantly reducing computational overhead compared to testing all 26 surrounding voxels in a 3D grid.

4. **Structured Domain Decomposition:**
   The global volume is partitioned into subvolumes using a structured block-wise 3D grid across the MPI processes. This ensures balanced computational load and minimizes the surface-to-volume ratio, thereby reducing the overall communication requirements.

5. **Contiguous Memory Allocation:**
   Allocating subvolumes and ghost cells in contiguous memory regions enhances cache utilization and simplifies array indexing during stencil computation.

6. **Parallel I/O:**

   - **Parallel File Access:** The application uses MPI parallel I/O routines (`MPI_File_open` and `MPI_File_read_at`) to read subvolume data directly into the process's memory space. This allows simultaneous I/O operations across processes, reducing the file reading time.
   - **I/O Pattern Matching Domain Decomposition:** The file reading operations leverage the domain decomposition to compute file offsets so that each process accesses only its required data subset. This minimizes the amount of unnecessary data being transferred and processed.
   - **Reduction of I/O Bottlenecks:** By combining parallel I/O with contiguous memory allocation, the overall I/O performance improves, contributing to better scalability for large-scale datasets.

4 Results

4.1 Local Extrema Count

The following results show the number of local minima and maxima detected in each time step of the datasets. Each pair represents `(local minima count, local maxima count)`.

- `data_64_64_64_3.bin.txt`
  - `(37988, 37991)`, `(37826, 38005)`, `(37788, 37978)`
- `data_64_64_96_7.bin.txt`
  - `(56875, 56848)`, `(56703, 56965)`, `(56680, 56847)`, `(56601, 56980)`, `(56769, 56937)`, `(56657, 56950)`, `(56862, 56996)`

4.2 Global Extrema Values

This section reports the global minimum and maximum values found at each time step. Each pair represents `(global minimum value, global maximum value)`.

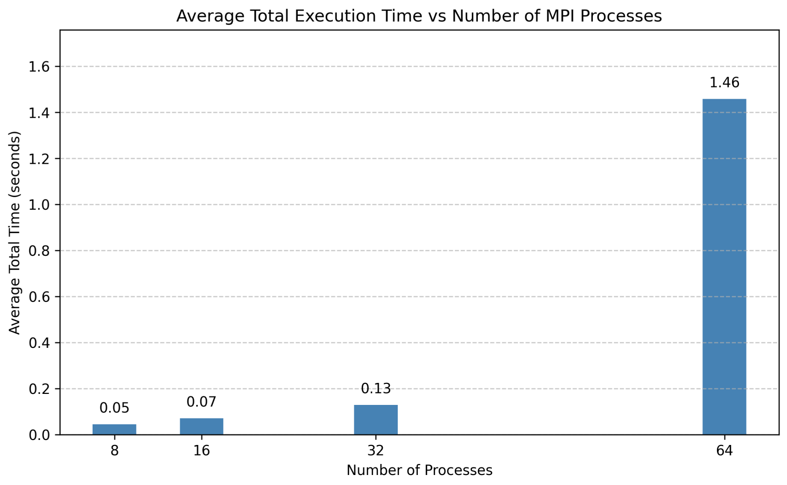
- `data_64_64_64_3.bin.txt`
  - `(-48.25, 33.63)`, `(-51.45, 33.35)`, `(-48.55, 33.35)`
- `data_64_64_96_7.bin.txt`
  - `(-48.25, 33.63)`, `(-51.45, 33.35)`, `(-48.55, 33.35)`, `(-43.13, 32.00)`, `(-53.55, 34.06)`, `(-49.68, 34.18)`, `(-53.55, 34.34)`

4.3 Scalability Results

- For `data_64_64_64_3.bin.txt`

Table 1: Execution Times for Different MPI Process Counts (in seconds)

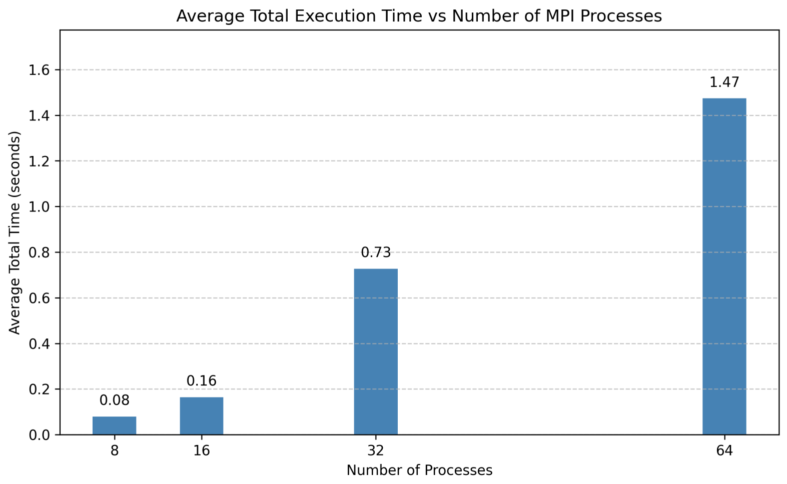
Processes	**Run**	**Read Time**	**Main Code Time**	**Total Time**
8	1	0.033741	0.012038	0.045625
	2	0.033981	0.010567	0.045047
16	1	0.062488	0.006251	0.068729
	2	0.068787	0.006727	0.075134
32	1	0.096927	0.006162	0.114722
	2	0.139569	0.008312	0.143332
64	1	1.452069	0.004537	1.453706
	2	1.454486	0.011444	1.463017
Average for 8		0.033861	0.011303	0.045336
Average for 16		0.065638	0.006489	0.071932
Average for 32		0.118248	0.007237	0.129027
Average for 64		1.453278	0.007991	1.458362



- For `data_64_64_96_7.bin.txt`

Table 2: Execution Times for Different MPI Process Counts (in seconds)

Processes	**Run**	**Read Time**	**Main Code Time**	**Total Time**
8	1	0.048191	0.033960	0.081031
	2	0.045537	0.033008	0.077444
16	1	0.092092	0.017413	0.108587
	2	0.100912	0.017532	0.219659
32	1	0.662728	0.011450	0.673577
	2	0.663676	0.011726	0.780815
64	1	1.389576	0.014664	1.394253
	2	1.547761	0.006082	1.552861
Average for 8		0.046864	0.033484	0.079238
Average for 16		0.096502	0.017473	0.164123
Average for 32		0.663202	0.011588	0.727196
Average for 64		1.468669	0.010373	1.473557



Result Analysis

Observations:

- **Computation Time Efficiency:** As process count increases, compute time decreases (from 0.011s at 8 processes to 0.007s at 64 processes for 1st data and from 0.033s at 8 processes to 0.010s at 64 for 2nd data), showing good parallel scalability.
- **Read Time Bottleneck:** Average read time increases significantly with more processes (e.g., 1.47s at 64 processes), indicating parallel I/O contention.

- **Optimizations Helped:**
  - `MPI_Type_vector` improved ghost layer exchange efficiency.
  - Parallel I/O allowed distributed subvolume reading.

Reason for Read Time Increase:

- Each compute node supports a maximum of 32 processes. At 64 processes, the job spans 2 nodes.
- This results in increased I/O overhead due to inter-node communication and simultaneous access to the file system, which causes contention and latency.

Limitations:

- Read time scalability poor at high process counts due to inter-node communication.
- Poor I/O scaling beyond 32 processes due to node-bound process limits.

5 Conclusions

1. **MPI Communication and Subdomain Exchange (30%)** (Akshit Shukrawal)
   Implemented MPI logic for communication between 3D subdomains using `MPI_Isend`, `MPI_Irecv`, and derived datatypes. This included handling face communication and ensuring proper synchronization across all neighboring processes.

2. **Domain Decomposition and Parallel I/O (30%)** (Ayush Meena)
   Designed the 3D domain decomposition strategy and developed parallel file reading and writing routines using `MPI_File` operations. This involved correctly partitioning the grid and mapping global indices to local ones for efficient parallel processing.

3. **Boundary Handling, Ghost Cells, and Validation (30%)** (Mohammed Anas)
   Implemented logic for managing ghost cells and enforcing periodic boundary conditions. Responsible for validation of the communication mechanism across different processor topologies and conducting thorough correctness testing.

4. **Utility Functions and Code Integration (10%)** (Aman Yadav)
   Contributed to helper functions for index mapping and assisted in integrating code modules. Also performed minor bug fixes and ensured compilation and runtime stability.