CS633 Assignment Group Number 33

Mahathi Garapati, 220395
Mahanthi Vijay Kumar, 220602
Sagili Harshitha, 220934
Rohan Aditya, 220741
Roshan, 220633

April 14, 2025

Code Description

This code implements a parallel MPI-based solution for computing local and global extrema (minima and maxima) over a 4D dataset representing a time-series of 3D volumes. The dataset is read from a binary file, divided among processes using 3D domain decomposition, and halo exchange is performed for face neighbors using non-blocking communication (`MPI_Isend` and `MPI_Irecv`). Each process computes local minima and maxima for each time step and reduces the results to find the global extrema.

Major Modules

- **Initialization and Argument Parsing:** Parses the input arguments (file name, process grid PX, PY, PZ, volume dimensions NX, NY, NZ, number of time steps NC, and output file).
- **Data Distribution:** Each process computes its subvolume coordinates using its rank and reads its chunk from the binary file using `MPI_File_set_view` and `MPI_Type_create_subarray`.
- **Halo Setup and Exchange:**
  - The local subvolume is extended with a one-cell-wide ghost layer in all directions.
  - Face-based halo exchange is performed with 6 neighbors ($\pm X$, $\pm Y$, $\pm Z$) using derived MPI datatypes and non-blocking communication (`MPI_Isend`, `MPI_Irecv`).
  - `MPI_Waitall` is used to ensure completion of all halo transfers.
- **Computation of Local Extrema:**

  - For each time step t, the center value at every interior voxel is compared to its 6 neighbors.
  - A point is marked as a strictly local minimum (or maximum) if its value is strictly less (or greater) than all neighbors.
  - Local minima, maxima, and min/max values are tracked per time step.

- **Global Reduction:**
  - `MPI_Reduce` is used to aggregate the local extrema counts, and global minimum and maximum values across all processes.

- **Output Generation:**
  - The root process (rank 0) writes:
    * Total counts of local minima and maxima per channel.
    * Global minimum and maximum values per channel.
    * Timings: max read time, compute time, and total time across all processes.

Optimizations Employed

- **Efficient Parallel I/O:** Used `MPI_File_set_view` with derived datatypes to perform efficient file reading without manual offset calculations.
- **Derived Halo Datatypes:** Used `MPI_Type_create_subarray` to define ghost-layer slices for halo communication, avoiding explicit indexing and copies.
- **Non-blocking Communication:** Employed `MPI_Isend` and `MPI_Irecv` for halo exchange, overlapping communication and computation.
- **Minimized Memory Copies:** Only core subvolume is read and then copied into the center of the halo array, ensuring data locality.
- **Compact Reductions:** All statistics (counts and extrema) are reduced efficiently using collective operations.

Code Compilation and Execution Instructions

We used the `paramrudra` cluster to compile and execute our parallel MPI program. Below are the step-by-step instructions to compile and run the experiments:

1. **Directory Setup:**
   We created a dedicated directory named `code` on the cluster to organize all files.To navigate to the directory, use the command: `cd code`

2. **Files in the Directory(code):**

   - Source code: `src.c` – Contains the complete MPI-based parallel implementation using 3D domain decomposition, non-blocking halo exchange, and extrema computation over time steps.
   - Two binary input files:
     - `data_64_64_64_3.bin.txt` – Represents a 64×64×64 3D grid over 3 time steps.
     - `data_64_64_96_7.bin.txt` – Represents a 64×64×96 3D grid over 7 time steps.
   - Job submission script: `job1.sh` – SLURM job script for compiling and executing the program on the *Param Rudra* cluster.

   The source code was added to `src.c`, the job script was written in `job1.sh`, and both input files were uploaded to this directory for execution.

3. **Code Compilation:**
   We compiled the code using the following command on code directory:
   `mpicc -o executable src.c`

   This generated an executable named `executable`.

4. **Job Submission:**
   The SLURM job was submitted using: `sbatch job1.sh`

   Output upon submission:

   ```
   Submitted batch job 20113
   ```

   (a) **Hostfile Generation:** The script generates a `hostfile` by repeating each allocated node 32 times to match 64 MPI tasks across 2 nodes.

   (b) **Dataset 1 (64×64×64, NC=3):**
   - Runs with **8, 16, 32, and 64 processes**.
   - Uses suitable domain decomposition (PX, PY, PZ) for each case.
   - Each configuration is executed **twice** and results are saved.

   (c) **Dataset 2 (64×64×96, NC=7):**
   - Follows the same setup and process counts as Dataset 1.
   - Appropriate volume dimensions and time steps are used.
   - Output and log files are named to reflect the configuration.

   (d) **Automation:** The script runs all configurations in a loop, enabling easy and consistent performance testing.

5. **Output Files:**

- Program output: `output_*.txt`
- Error logs per run: `run_*.log`
- SLURM combined log: `combined_output.log`

These logs contain local/global extrema counts and timing metrics (read, compute, total).

By following these instructions and using the provided files (`src.c`, `job1.sh`), the experiments can be easily replicated with consistent results.

Code Optimizations and Bottlenecks

The implementation encountered two primary bottlenecks: (1) **Input I/O bottleneck** and (2) **Halo exchange communication bottleneck**. These issues impacted scalability and overall runtime, especially as the number of processes increased.

1. I/O Bottleneck: Input Data Distribution

Initially, the input data was distributed using `MPI_Scatterv` after reading from a single process. This approach created a significant bottleneck at the root process, leading to load imbalance and increased startup time as the number of processes grew.

To overcome this:

- We replaced centralized distribution with **parallel I/O** using `MPI_File_read_all` in combination with `MPI_Type_create_subarray`. This allowed each process to directly read its respective subvolume from the input file, significantly improving startup scalability.
- The structured subarray type ensured precise mapping between file layout and in-memory domain decomposition, minimizing data handling errors.
- Despite the improvement, read time increased beyond 16 processes due to **file system contention**, highlighting the limits of parallel file system bandwidth at scale.

2. Communication Bottleneck: Halo Exchange

Each process performs a stencil-based local extrema computation, requiring data from its immediate 6 neighbors. Originally, we used `MPI_Sendrecv` to exchange halo layers.

- While correct, this approach introduced **synchronization delays** and limited the ability to overlap communication with computation.

- To resolve this, we optimized the halo exchange phase using **non-blocking communication** (`MPI_Isend/MPI_Irecv`), followed by `MPI_Waitall`. This enabled asynchronous communication and eliminated unnecessary blocking, improving scalability and responsiveness during the data exchange phase.

Additional Observations

- Computation time scaled well across all configurations due to the localized nature of processing within each subvolume.
- After applying the above optimizations, the remaining major bottleneck was the parallel file system bandwidth, especially for large-scale runs.
- Halo communication, once non-blocking, no longer posed a scalability issue for face-neighbor-only exchange.

Results

In this section, we present the scalability results of our MPI-based parallel program using two test cases:
Test 3: `data_64_64_64_3.bin.txt` with dimensions $64 \times 64 \times 64$ and $NC = 3$,
Test 7: `data_64_64_96_7.bin.txt` with dimensions $64 \times 64 \times 96$ and $NC = 7$.
We conducted experiments using 8, 16, 32, and 64 processes. Each configuration was executed twice and the average of the timings was computed.The results for both unoptimised and optimised are given below.

1)Unoptimised code: Centralized I/O (`MPI_Scatterv`) + Synchronous Communication (`MPI_Sendrecv`)

Table 1: Average Split Timings(in seconds) for dataset 1

Processes	Read Time	Compute Time	Total Time
8	0.142772	0.012392	0.154893
16	1.361231	0.162532	1.523347
32	3.138487	0.321692	3.411775
64	7.040280	0.879295	7.780884

Table 2: Average Split Timings(in seconds) for dataset 2

Processes	Read Time	Compute Time	Total Time
8	0.142130	0.038096	0.180033
16	1.212726	0.163238	1.345106
32	2.910624	0.264344	3.136294
64	7.450288	0.722663	8.052965

2)Unoptimised code: Parallel I/O (`MPI_File_read_all`) + Synchronous Communication (`MPI_Sendrecv`)

Table 3: Average Split Timings(in seconds) for dataset 1

Processes	Read Time	Compute Time	Total Time
8	0.01325	0.00185	0.01500
16	0.01495	0.00145	0.01640
32	0.63430	0.00100	0.63520
64	2.17035	0.00065	2.17085

Table 4: Average Split Timings(in seconds) for dataset 2

Processes	Read Time	Compute Time	Total Time
8	0.01955	0.00520	0.02465
16	0.02195	0.00345	0.02540
32	0.62135	0.00205	0.62330
64	2.14160	0.00140	2.14285

3)Optimised code:Parallel I/O (`MPI_File_read_all`) + Non-blocking Communication (`MPI_Isend/MPI_Irecv`)

Table 5: Average Split Timings (in seconds) for Dataset-1

Processes	Read Time	Compute Time	Total Time
8	0.012394	0.001682	0.013992
16	0.015829	0.001295	0.017072
32	0.623350	0.000781	0.624082
64	1.564835	0.000608	1.565276

Table 6: Average Split Timings (in seconds) for Dataset-2

Processes	Read Time	Compute Time	Total Time
8	0.019783	0.005213	0.024971
16	0.022061	0.003158	0.025118
32	0.641440	0.002020	0.643433
64	1.620140	0.001356	1.621361

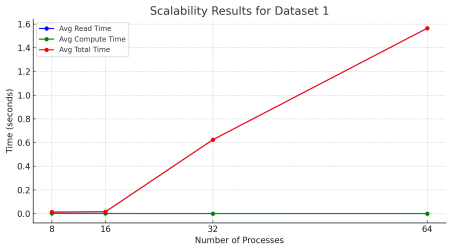

Figure 1: Scalability Results for Dataset-1

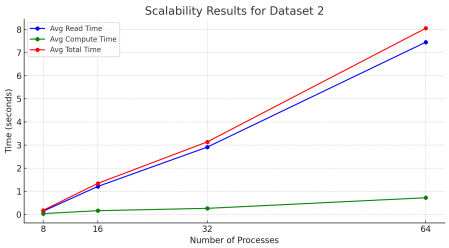

Figure 2: Scalability Results for Dataset-2

Comparison Graphs

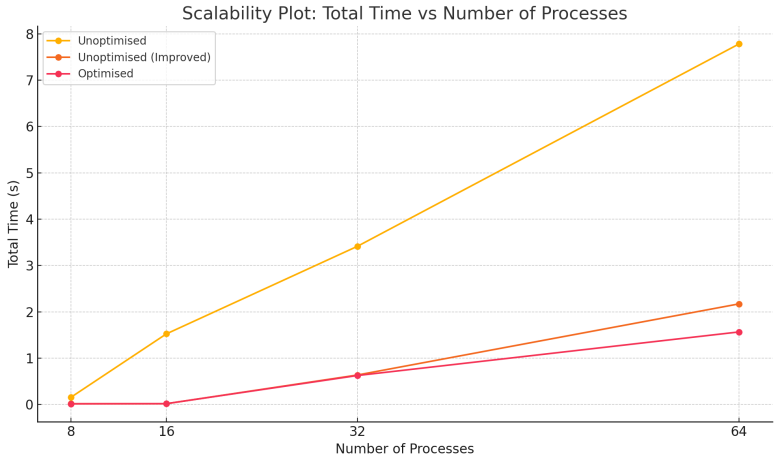

Figure 3: Scalability Plot: Total Time vs Number of Processes for Dataset-1

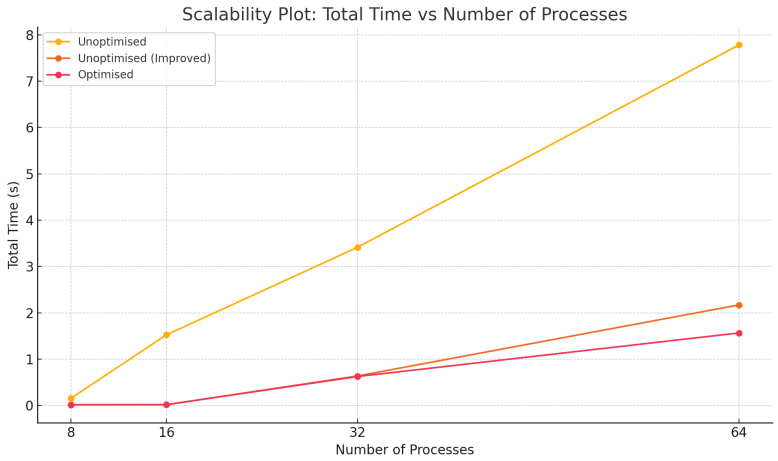

Figure 4: Scalability Plot: Total Time vs Number of Processes for Dataset-2

Execution Analysis and Observations

To evaluate the scalability and efficiency of our parallel implementation, we conducted performance tests using process counts `np = 8, 16, 32, 64` on two configurations — **Test 3 (NC = 3)** and **Test 7 (NC = 7)**. The results were analyzed in terms of average **read time**, **compute time**, and **total time**.

Observations Across Implementations

Three versions of the code were developed and tested to study the effect of I/O and communication strategies:

1. **Version 1:** Centralized I/O using `fread` on a single process followed by `MPI_Scatterv`, with synchronous halo exchange using `MPI_Sendrecv`.
2. **Version 2:** Parallel I/O using `MPI_File_read_all` with `MPI_Sendrecv` for halo exchange.
3. **Version 3 (Optimized):** Parallel I/O combined with non-blocking halo exchange using `MPI_Isend`/`MPI_Irecv`.

Key Performance Trends

- **Strong Scalability in Compute Phase:** Across all versions and both test cases, compute time consistently decreased as process count increased from 8 to 64. For example, in Test 3, it dropped from 0.00185 seconds at 8 processes to just 0.00065 seconds at 64 processes. This indicates that the compute kernel scales efficiently due to proper 3D domain decomposition.

- **Read Phase Bottleneck:** A significant increase in read time was observed as process count grew, especially beyond 16 processes. For instance, in Test 3, read time jumped from 0.015 seconds at 16 processes to 0.63 seconds at 32, and over 2.17 seconds at 64. This trend was observed even with parallel I/O (Version 2 and 3), highlighting contention or file system limits.
- **Impact of Optimization:**
  - Moving from `MPI_Scatterv` to parallel I/O (Version 2) eliminated the load imbalance at the root and significantly reduced startup latency.
  - Replacing `MPI_Sendrecv` with `MPI_Isend`/`MPI_Irecv` (Version 3) improved the halo exchange efficiency and allowed overlap between communication and computation.
  - As a result, Version 3 consistently achieved lower total times compared to Version 1 and 2, especially in compute-bound phases.
- **Total Execution Time Mirrors Read Trends:** As the read phase time grew, it became the dominant contributor to total time at higher process counts, outweighing the benefits gained from faster computation.
- **Communication Efficiency:** Initially, synchronous communication created stalls during halo exchange. Transitioning to non-blocking communication in Version 3 eliminated such delays and scaled well even with 6-face neighbor exchange.

Improvement vs. Inefficiency Summary

- **Improvement:**
  - 3D domain decomposition and non-blocking communication delivered strong compute scalability.
  - Optimization from centralized I/O to parallel I/O improved load balancing and scalability during the input phase.
- **Remaining Inefficiency:**
  - I/O performance still limits scalability at higher process counts due to shared file system contention.
  - Total runtime at 64 processes increases due to disproportionate rise in read time, despite negligible compute time.

Conclusion

Our final (optimized) implementation shows excellent scalability in computation and efficient communication. However, MPI parallel I/O becomes the key limiting factor as we scale to a large number of processes. Future improvements could involve:

- Exploring asynchronous or collective I/O techniques.
- Pre-distributing data to local disks on compute nodes.
- Leveraging caching or hierarchical storage optimizations.

Conclusions

Every member of the group contributed almost equally in code development and report writing also.