CS633 Assignment Group Number 28

Akshat Singh Tiwari (210094)
Hridhan Patel (210687)
Sanyam Jain (210928)
Satyam Gupta (218170942)
Tanmey Agarwal (211098)

14/04/2025

1 Code Description

The program operates on a 3D numerical dataset defined by global dimensions (`nx`, `ny`, `nz`) and containing multiple data columns or variables (`nc`). It utilizes a 3D Cartesian process grid (`px`, `py`, `pz`) to partition the dataset among the available MPI processes. The core logic involves reading data in parallel, exchanging boundary information (halos) between neighboring processes, performing local computations to find extrema, and finally reducing these local results to obtain global statistics.

1.1 Modular Breakdown

The code can be broken down into the following logical modules or stages:

1. **Initialization:**
   - Initializes the MPI environment (`MPI_Init`).
   - Retrieves the process rank and total number of processes (`MPI_Comm_rank`, `MPI_Comm_size`).
   - Parses command-line arguments to get dataset path, process grid dimensions (`px`, `py`, `pz`), global data dimensions (`nx`, `ny`, `nz`), number of columns (`nc`), and output file path.
   - Validates that the product of process grid dimensions matches the total number of MPI processes.
   - Creates a 3D Cartesian communicator (`MPI_Cart_create`) to manage the process grid topology and determine process coordinates (`MPI_Cart_coords`).
   - Calculates the dimensions of the local data block assigned to each process, excluding halos (`lx`, `ly`, `lz`), and including halos (`local_lx`, `local_ly`, `local_lz`).
   - Allocates memory for storing local data for all columns, including halo regions (`local_data_all_cols`, `full_data_all_cols`).
2. **Parallel Data Loading and Distribution:**

- Splits the Cartesian communicator to create specialized communicators (`io_comm`) for I/O, involving only processes with `px_id = 0` and `py_id = 0` (`MPI_Comm_split`).
- Reader processes (those in `io_comm`) collectively read horizontal slabs (layers) of the dataset from the input file using MPI-IO (`MPI_File_open`, `MPI_File_read_at_all`, `MPI_File_close`). Each reader process reads one `lz`-thick slab corresponding to its `pz_id`.
- Reader processes extract their own local data portion (including x and y halos) directly from the read slab.
- Reader processes pack and send the appropriate data portions (including x and y halos for all `nc` columns) to other non-reader processes within the same z-layer using non-blocking sends (`MPI_Isend`). Data for all columns is sent in a single message per destination process.
- Non-reader processes receive their data portion (`MPI_Recv`) and unpack it into their column-specific local data buffers (`local_data_all_cols`).
- Reader processes wait for all sends to complete (`MPI_Waitall`).

3. **Halo Exchange (Z-Dimension):**

   - Determines the ranks of neighboring processes in the z-direction (up and down) using `MPI_Cart_rank`.
   - Creates an MPI derived datatype (`MPI_Type_vector`) to represent an xy-plane of data, facilitating communication of non-contiguous halo regions.
   - Copies the core local data from `local_data_all_cols` to the `full_data_all_cols` buffer, which includes space for z-halos.
   - Exchanges halo data (xy-planes) in the z-direction with neighboring processes using `MPI_Sendrecv` for each data column. This ensures each process has the necessary boundary data from its top and bottom neighbors.
   - Synchronizes all processes (`MPI_Barrier`) after halo exchange to ensure data is ready for computation.

4. **Local Computation (Extrema Detection):**

   - Iterates through the *interior* points of the local subgrid for each data column (`c` from 0 to `nc-1`). Halo regions are used for comparisons but are not processed as central points.
   - For each interior point, compares its value (`full_data_all_cols[c][idx]`) with its six neighbors (left, right, bottom, top, front, back) residing within the `full_data_all_cols` buffer (which includes halos).
   - Identifies if the point is a local minimum (strictly less than all neighbors) or a local maximum (strictly greater than all neighbors).
   - Updates local counts for minima and maxima (`local_count`) and tracks the minimum and maximum values encountered (`local_minima`, `local_maxima`) within the process's subdomain for each column.

5. **Global Reduction:**

   - Uses `MPI_Reduce` to aggregate the local results across all processes for each column.

- Computes the global minimum (`MPI_MIN`) and global maximum (`MPI_MAX`) values.
- Calculates the total count of local minima and maxima (`MPI_SUM`) across the entire dataset.
- The root process (rank 0) receives the final global results.

6. **Cleanup:**
   - Frees all dynamically allocated memory (local data buffers, result arrays, MPI request arrays, temporary buffers).
   - Frees MPI derived datatypes and communicators (`MPI_Type_free`, `MPI_Comm_free`).
   - Finalizes the MPI environment (`MPI_Finalize`).

2 Code Compilation and Execution Instructions

This section provides instructions on how to compile and execute the parallel extrema detection code. An MPI environment (like OpenMPI or MPICH) must be properly installed and configured on the system.

2.1 Compilation

The C code, assumed to be in a file named `src.c`, can be compiled using an MPI C compiler wrapper, typically `mpicc`. Use the following command in the terminal:

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

This command compiles the source file `src.c` and creates an executable file named `src` in the current directory. The `-o src` flag specifies the name of the output executable.

2.2 Execution

The compiled program `src` is executed in parallel using the `mpirun` command. The program requires several command-line arguments specifying the input data, process grid configuration, global data dimensions, number of data columns, and the output file name.

The general format for execution is:

```
mpirun -n <np> ./src <input_data> <px> <py> <pz> <nx> <ny> <nz> <nc> <output_file>
```

Where:

- `-n <np>`: Specifies the total number of MPI processes (`np`) to launch. This number must equal the product of `px * py * pz`.
- `./src`: The path to the compiled executable.
- `<input_data>`: Path to the binary input data file (e.g., `./executable_data.bin`). Corresponds to `argv[1]` in the code[1].
- `<px>`: Number of processes in the x-dimension of the process grid. Corresponds to `argv[2]` [1].

- `<py>`: Number of processes in the y-dimension of the process grid. Corresponds to `argv[3]`[1].
- `<pz>`: Number of processes in the z-dimension of the process grid. Corresponds to `argv[4]`[1].
- `<nx>`: Global data dimension in the x-direction. Corresponds to `argv[5]`[1].
- `<ny>`: Global data dimension in the y-direction. Corresponds to `argv[6]`[1].
- `<nz>`: Global data dimension in the z-direction. Corresponds to `argv[7]`[1].
- `<nc>`: Number of data columns (variables) per grid point. Corresponds to `argv[8]`[1].
- `<output_file>`: Path to the file where results will be written (e.g., `output.txt`). Corresponds to `argv[9]`[1].

```
mpirun -n 16 ./src ./executable_data.bin 4 2 2 64 64 96 7 output.txt
```

2.3 Running Test Cases using Provided Scripts

For convenience, shell scripts are provided to run the specific test cases presented in the Results section. The scripts are named following the pattern `job_script_<np>_<tc>.sh`, where `<np>` is the number of processes (e.g., 8, 16, 32, 64) and `<tc>` indicates the test case (e.g., 1 for the 64x64x64x3 dataset, 2 for the 64x64x96x7 dataset).

3 Code Optimizations

To enhance performance and scalability, several key bottlenecks were identified and addressed through targeted optimizations:

1. **Parallel I/O for Faster Data Loading:**
   - **Bottleneck:** Reading large datasets serially using a single process was time-consuming.
   - **Optimization:** Implemented parallel I/O using MPI-IO (`MPI_File_read_at_all`). Given the row-major data layout, designated reader processes (one per z-layer, determined via `MPI_Comm_split`) read contiguous data slabs along the z-dimension, distributing the I/O workload.
2. **Efficient Data Distribution with Integrated Halos:**
   - **Bottleneck:** Distributing data and then performing separate halo exchanges in the x and y dimensions involved multiple communication steps.
   - **Optimization:** The initial data distribution phase was modified so that reader processes send data blocks that already include the necessary halo regions required by neighbours in the x and y dimensions. This eliminates the need for subsequent explicit x/y halo exchange steps within each z-layer.
3. **Reduced Communication Overhead During Distribution:**

- **Bottleneck:** Sending data for each of the `nc` columns separately during distribution would lead to a high number of small messages, increasing latency.
- **Optimization:** Data for all `nc` columns destined for a specific process is packed into a single buffer. This consolidated buffer is then sent using a single non-blocking message (`MPI_Isend`), significantly reducing the total message count and the impact of communication latency. Non-reader processes receive this combined data with a single `MPI_Recv` and unpack it locally.

4. **Optimized Z-Dimension Halo Exchange:**

   - **Refinement:** While not necessarily a primary bottleneck fix like I/O, the required halo exchange in the z-dimension was implemented efficiently.
   - **Technique:** Used `MPI_Sendrecv` for safe neighbour communication and employed an `MPI_Type_vector` derived datatype to represent the xy-plane, allowing direct transfer of halo layers without manual packing/unpacking for this specific step.

These optimizations collectively target I/O limitations and reduce inter-process communication overhead, improving the overall runtime and scalability of the analysis.

4 Results

This section presents the performance results of the parallel extrema detection code. The execution time was measured for different dataset sizes by varying the number of MPI processes (`np`). Timings were recorded using `MPI_Wtime()` and the maximum time across all processes for each phase was determined using `MPI_Reduce` with `MPI_MAX`[1].

The measured times correspond to:

- **I/O & Distribution (s)**: Parallel data reading, distribution (including x/y halos), and z-halo exchange (`time2-time1`)[1].
- **Computation (s)**: Local extrema detection and global reduction (`time3-time2`)[1].
- **Total (s)**: Sum of the above two phases (`time3-time1`)[1].

4.1 Test Case 1: 64x64x64 Grid, 3 Data Columns

Table 1: Timing Results (seconds) for 64x64x64 Grid, 3 Columns

Processes (np)	I/O & Distribution (s)	Computation (s)	Total (s)
8	0.0157	0.0064	0.0215
16	0.0179	0.0039	0.0211
32	0.5787	0.0032	0.5808
64	1.8631	0.0033	1.8651

Table 2: Timing Results (seconds) for 64x64x96 Grid, 7 Columns

Processes (np)	I/O & Distribution (s)	Computation (s)	Total (s)
8	0.0265	0.0204	0.0458
16	0.0279	0.0126	0.0379
32	0.5428	0.0094	0.5486
64	1.9419	0.0050	1.9455

4.2 Test Case 2: 64x64x96 Grid, 7 Data Columns

4.3 Test Case 3: Larger Input Size

Table 3: Timing Results (seconds) for Larger Dataset

Processes (np)	I/O & Distribution (s)	Computation (s)	Total (s)
8	5.1426	7.8324	12.2919
16	4.7381	4.6730	9.4111
32	5.4094	3.1695	8.5789
64	5.6933	1.7483	7.4416

4.4 Analysis of Results

The timing data (Tables 1-3) highlights the code's performance characteristics and scalability.

Computation Scaling: The core computation phase (`time3-time2`) consistently scales well, with its execution time decreasing as the number of processes (`np`) increases across all test cases. This indicates effective parallelization of the local extrema detection workload[1].

I/O and Distribution Bottleneck: The combined I/O and data distribution phase (`time2-time1`) presents a scaling challenge. For smaller datasets (TC1, TC2), its time increases sharply at higher process counts (32, 64), becoming the dominant factor. This suggests communication overhead (many messages from reader processes, potential network/file system contention) outweighs the benefits of parallel I/O in this regime[1]. For the larger dataset (TC3), this phase's time is more stable but remains significant.

Overall Strong Scaling: The total execution time reflects these competing factors.

- **Poor Scaling (Small Datasets):** For TC1 and TC2, the total time increases at higher `np` because the rising I/O & distribution costs negate the computation time savings. This occurs when the work per process becomes too small relative to communication overheads.
- **Good Scaling (Large Dataset):** For TC3, the total time decreases as `np` increases. Here, the problem size is large enough that the significant reduction in computation time outweighs

the relatively stable (though high) I/O & distribution costs, demonstrating effective strong scaling.

5 Conclusions

The following are the individual contributions by team members:

- **Akshat Singh Tiwari (210094):** Implemented parallel I/O and domain decomposition for data distribution. Debugged critical issues such as deadlocks, boundary conditions in halo exchanges, and memory allocation errors. Prepared a report documenting the project.
- **Hridhan Patel (210687):** Implemented the core logic for computing local and global extrema within subdomains. Also handled the validation of results and ensured correctness through test cases.
- **Satyam Gupta (21817042):** Responsible for trying alternating method of reading and parsing the input file at root process, distributing data across processes using mpi_type_create_subarray. Also helped in debugging and running the code on PARAM.
- **Tanmey Agarwal (211098):** Contributed to debugging and performance optimization of the code. Explored alternative strategies and compared their execution time with the final chosen approach. Generating testcases and checked the correctness of the code. Helped in preparing the report.
- **Sanyam Jain (210928):**