CS633 Assignment Group Number 38

Arindom Bora (210183)
Brid Ojas Chandrashekhar (210275)
Saurav Kumar (210950)
Umang Sinha (211124)

1 Code Description

The implemented MPI C program is designed to read and analyze time-series data from a 3D spatial volume distributed over multiple time steps. The primary goals of the code are to: (i) efficiently read and distribute the dataset across MPI processes, (ii) compute global minimum and maximum values at each time step, and (iii) count the number of local minima and maxima per time step in the 3D domain.

Domain Decomposition and Reading Strategy

The input data is structured in an `XYZ` order, where each spatial coordinate point (x, y, z) is associated with a time series of `nc` float values. The total data size is therefore $nx \times ny \times nz \times nc$ floats. This volume is decomposed into a 3D process grid with px, py, and pz number of processes in the X, Y and Z directions, respectively. The division is regular and assumes that the global dimensions (nx, ny, nz) are evenly divisible by (px, py, pz). Each process works on a sub-volume of size $(x_{wid}, y_{wid}, z_{wid})$, where:

$$x_{wid} = nx/px, \quad y_{wid} = ny/py, \quad z_{wid} = nz/pz$$

Each process identifies its local subdomain using its ranks in each of the 3 directions and calculates offsets according to the following formulae :

$$x_{rank} = rank\%px, \quad y_{rank} = (rank/px)\%py, \quad z_{rank} = rank/(px * py)$$

$$z_{offset} = z_{rank} * z_{wid} \quad y_{offset} = y_{rank} * y_{wid} \quad x_{offset} = x_{rank} * x_{wid}$$

MPI subarrays are used to read the appropriate chunk of data for each process. First, each MPI process allocates a buffer large enough to store its local subdomain, determined by $x_{wid} \times y_{wid} \times z_{wid} \times nc$. Then, a derived MPI datatype is created using `MPI_Type_create_subarray`, which defines the exact slice of the global dataset that this process should read. This datatype helps MPI skip over irrelevant parts of the file and focus only on the required subarray. The file is opened collectively with `MPI_File_open`, and a file view is set using `MPI_File_set_view`, associating the view with the custom datatype. Finally, `MPI_File_read_all` is used to read the data into the local buffer in parallel across all processes, ensuring each process gets the correct portion of the global dataset efficiently.

Data Reading and Domain Decomposition Code Snippet

```
int gsize[4] = {nz, ny, nx, nc};
int xwid = nx / px, ywid = ny / py, zwid = nz / pz;
int lsize[4] = {zwid, ywid, xwid, nc};
int x_rank = rank % px, y_rank = (rank / px) % py, z_rank = rank / (px * py);
int starts[4] = {z_rank * zwid, y_rank * ywid, x_rank * xwid, 0}; //z, y, x offsets
float *buffer = (float *)malloc(xwid * ywid * zwid * nc * sizeof(float));
MPI_Datatype filetype;
MPI_Type_create_subarray(4, gsize, lsize, starts, MPI_ORDER_C, MPI_FLOAT,&filetype);
```

```
MPI_Type_commit(&filetype);
MPI_File fh;
MPI_File_open(MPI_COMM_WORLD, file_name, MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh, 0, MPI_SEEK_SET);
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native", MPI_INFO_NULL);
MPI_File_read_all(fh, buffer, xwid*ywid*zwid*nc, MPI_FLOAT, MPI_STATUS_IGNORE);
MPI_File_close(&fh);
```

Neighbor Communication

For detecting local extrema, data from neighboring subdomains is required. Each process determines its six neighbors (left, right, up, down, front, back) with the formulae given below. To efficiently communicate 2D faces of the 3D grid, 2 custom MPI datatypes (x_face and y_face) are created using MPI_Type_vector to represent non-contiguous slices in memory for XZ and YZ planes. For the XY planes, the data is contiguous, hence, no vector datatype is created for it. A 2D array, containing 6 rows (one for each direction) is created to store data from neighboring processes. For each valid neighbor, the process sends its boundary face using non-blocking MPI_Isend and receives the corresponding face from the neighbor using MPI_Recv, with proper tags ensuring correct message matching.

Code Snippet to find rank of neighbour processes

```
int *neighbour_ranks = (int *)malloc(6 * sizeof(int));
neighbour_ranks[0] = (x_rank > 0) ? rank - 1 : MPI_PROC_NULL;            // left
neighbour_ranks[1] = (x_rank < px - 1) ? rank + 1 : MPI_PROC_NULL;       // right
neighbour_ranks[2] = (y_rank > 0) ? rank - px : MPI_PROC_NULL;           // up
neighbour_ranks[3] = (y_rank < py - 1) ? rank + px : MPI_PROC_NULL;      // down
neighbour_ranks[4] = (z_rank > 0) ? rank - px * py : MPI_PROC_NULL;      // front
neighbour_ranks[5] = (z_rank < pz - 1) ? rank + px * py : MPI_PROC_NULL; // back

MPI_Datatype x_face, y_face;
MPI_Type_vector(zwid, xwid * nc, xwid * ywid * nc, MPI_FLOAT, &x_face); // xz face
MPI_Type_commit(&x_face);
MPI_Type_vector(zwid * ywid, nc, xwid * nc, MPI_FLOAT, &y_face); // yz-face
MPI_Type_commit(&y_face);
```

Local and Global Computation

Each process analyzes its local data in the following way:

- For each time step, iterate over all spatial points in the subdomain.
- Check if the current point is the global minima or maxima in the subdomain.
- Determine whether each point is a local extrema by comparing it with its six neighbors. For boundary points, neighbor values are retrieved from the communicated faces (stored in a 2D array). A helper function `is_local_extrema()` performs this comparison, which is explained later.
- After local analysis, MPI reductions are used to gather global results at rank 0. `MPI_Reduce` with operation `MPI_MIN` and `MPI_MAX` is used to compute the global minimum and maximum values across all processes for each time step.
- `MPI_Reduce` with operation `MPI_SUM` is used to collect the total count of local minima and maxima in the entire domain.

The function `is_local_extrema()` checks whether a specific grid point in a 3D volume is a local minimum or maximum at a given time step by comparing its value with its six immediate neighbors (along ±X, ±Y, ±Z). It first retrieves the value at the current point and then iterates through each neighbor. If the neighbor is within the local domain, its value is directly accessed from the main data array. If the neighbor lies outside the domain boundary, the function checks if a corresponding neighboring process exists (via n_ranks) and accesses the appropriate halo layer data from n_values. If any neighbor's value violates the condition for local extrema (i.e., being strictly smaller or larger than the current value depending on find_max), the function returns false. If no such violation is found, it confirms the point as a local minimum or maximum.

Output and Timing

The rank 0 process writes the results to an output file in the following format:

1. A list of (`local_minima`, `local_maxima`) pairs for each time step.
2. A list of (`global_minima`, `global_maxima`) pairs.
3. Timings for reading, computation, and total execution.

Three key timing metrics are captured using `MPI_Wtime` to help analyze performance:

- Time taken to read the data (parallel input).
- Time taken for main computation (includes sending and receiving data from neighbour processes and checking for global and local extrema).
- Total execution time.

The maximum value of the times across all processes is reported in the output.

2 Code Compilation and Execution Instructions

To compile the code file (src.c), use the following command :

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

This creates an executable file soln.exe which can be executed using a job script as follows :

Job Script (job.sh)

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --ntasks-per-node=8
#SBATCH --error=job.%J.err
#SBATCH --output=job.%J.out
#SBATCH --time=00:10:00  ## wall-clock time limit
#SBATCH --partition=standard  ## can be "standard" or "cpu"

echo ‘date‘
for i in {1..5}
do
    output_file="output_64_64_64_3_8_run_${i}.txt"
    mpirun -np 8 ./src data_64_64_64_3.bin.txt 2 2 2 64 64 64 3 "$output_file"
done
echo ‘date‘
```

The above jobscript runs the test case 5 times and saves a different output file each time.
*While running the test cases, we ensured that N * ntasks-per-node = np. To run a test case more than 1 time,

3 Code Optimizations

It is observed in the results that the computation time is much less compared to the read time in all scenarios. Hence, optimizing the code has been focused on strategies to more efficiently read and decompose data. We have tried out 3 strategies for this task:

1. The initial approach is to read data through only one process and distribute it using MPI_ISend and MPI_Recv across all processes in a loop.
2. The second approach is to read data through only one process and using MPI_Type_create_subarray to distribute packets of data across different processes
3. The third and final approach is parallely open the data file in all processes and set view using MPI_Type_create_subarray to read the desired subdomain

. The results are discussed in the next section.

Domain Decomposition and Reading Strategy (Approach 1)

```
float *buffer = (float *)malloc(zwid * ywid * xwid * nc * sizeof(float));
if (rank == 0){
    MPI_Request req[size];
    float *global_data = (float *)malloc(nx * ny * nz * nc * sizeof(float));
    // Read full data in process 0
    FILE *f = fopen(file_name, "rb");
    fread(global_data, sizeof(float), nx * ny * nz * nc, f);
    fclose(f);
    for (int r = 0; r < size; r++){ // Iterate over all process ranks
        int x_r = r % px, y_r = (r / px) % py, z_r = r / (px * py);
        int x0 = x_r * xwid, y0 = y_r * ywid, z_r * zwid; // Starting indices
        float *temp = (float *)malloc(xwid * ywid * zwid * nc * sizeof(float));
        for (int z = 0; z < zwid; z++){
            for (int y = 0; y < ywid; y++){
                for (int x = 0; x < xwid; x++){
                    for (int c = 0; c < nc; c++){
                        int gx = x0 + x, gy = y0 + y, gz = z0 + z;
                        temp[IDX(x, y, z, xwid, ywid) * nc + c] =
                            global_data[IDX(gx, gy, gz, nx, ny) * nc + c];
                    }
                }
            }
        }
        if (r == 0){
            memcpy(buffer, temp, xwid * ywid * zwid * nc * sizeof(float));
        }
        else{
        MPI_Isend(temp, xwid*ywid*zwid*nc, MPI_FLOAT, r, r, MPI_COMM_WORLD, &req[r]);
        }
        free(tempbuf);
    }
    free(global_data);
}
else{
    MPI_Recv(buffer, xwid*ywid*zwid*nc, MPI_FLOAT, 0, rank, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
}
```

Domain Decomposition and Reading Strategy (Approach 2)

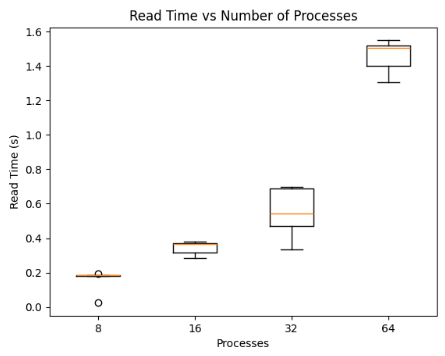
```
float *buffer = (float *)malloc(zwid * ywid * xwid * nc * sizeof(float));
float *all_data = NULL;
if (rank == 0){
    long total_elems = (long)nz * ny * nx * nc;
    all_data = (float *)malloc(total_elems * sizeof(float));
    FILE *f = fopen(file_name, "rb");
    fread(all_data, sizeof(float), total_elems, f);
    fclose(f);
}

int global_sizes[4] = {nz, ny, nx, nc};
int local_sizes[4] = {zwid, ywid, xwid, nc};
int elems_per_rank = zwid * ywid * xwid * nc;
MPI_Datatype global_type, local_type;
// Create a datatype representing the entire global array (needed for root process)
MPI_Type_create_subarray(4, global_sizes, global_sizes, (int[4]){0, 0, 0, 0},
    MPI_ORDER_C, MPI_FLOAT, &global_type);
MPI_Type_commit(&global_type);
// Create a datatype for the local portion each process will receive
MPI_Type_create_subarray(4, global_sizes, local_sizes, starts, MPI_ORDER_C,
    MPI_FLOAT, &local_type);
MPI_Type_commit(&local_type);
// Resized types for proper data distribution
MPI_Datatype resized_local_type;
MPI_Type_create_resized(local_type, 0, sizeof(float), &resized_local_type);
MPI_Type_commit(&resized_local_type);
// Scatter the data using derived datatypes
MPI_Scatter(all_data, 1, resized_local_type, buffer, elems_per_rank, MPI_FLOAT, 0,
    MPI_COMM_WORLD);
```

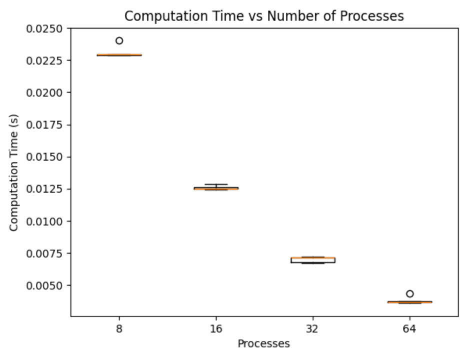
4 Results

4.1 Timing Results for 64_64_96_7 dataset

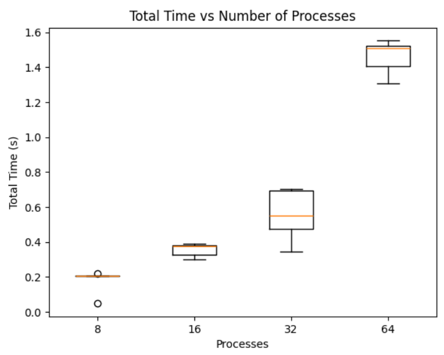
Processes	Run	ReadTime (s)	ComputationTime (s)	TotalTime (s)
8	1	0.024059	0.024005	0.048062
	2	0.185587	0.022895	0.208482
	3	0.182176	0.022917	0.205093
	4	0.195955	0.022906	0.218861
	5	0.184893	0.022896	0.207789
16	1	0.315859	0.012452	0.328311
	2	0.378927	0.012835	0.391762
	3	0.364038	0.012518	0.376556
	4	0.286068	0.012489	0.298557
	5	0.370589	0.012637	0.383226
32	1	0.336384	0.006730	0.343114
	2	0.697488	0.006789	0.704277
	3	0.544430	0.007175	0.551605
	4	0.468985	0.007157	0.475786
	5	0.686854	0.007195	0.694049
64	1	1.400546	0.004337	1.404883
	2	1.519469	0.003680	1.523149
	3	1.304731	0.003676	1.308407
	4	1.548557	0.003755	1.552312
	5	1.504362	0.003603	1.507965



(a) Read Time vs Processes



(b) Computation Time vs Processes

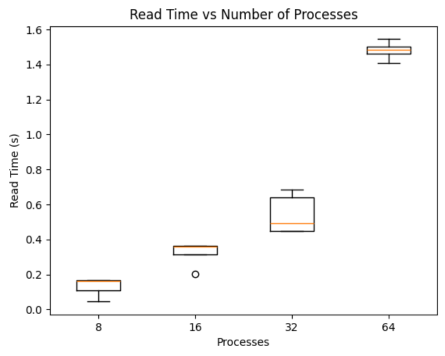


(c) Total Time vs Processes

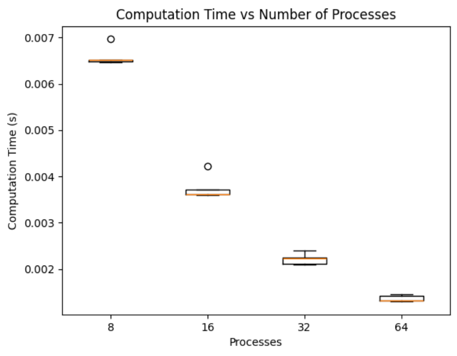
Figure 1: Box plots of times for different stages of MPI execution for data_64_64_96_7 dataset.

4.2 Timing Results for 64_64_64_3 dataset

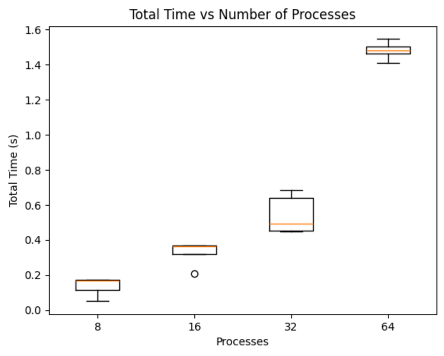
Processes	Run	ReadTime (s)	ComputationTime (s)	TotalTime (s)
8	1	0.109756	0.006966	0.116712
	2	0.044191	0.006468	0.050655
	3	0.162928	0.006525	0.169452
	4	0.166056	0.006481	0.172537
	5	0.164951	0.006523	0.171470
16	1	0.202934	0.004219	0.207134
	2	0.358937	0.003603	0.362531
	3	0.364957	0.003616	0.368566
	4	0.364329	0.003611	0.367925
	5	0.314542	0.003714	0.318245
32	1	0.683278	0.002096	0.685373
	2	0.493084	0.002229	0.495307
	3	0.448918	0.002244	0.451140
	4	0.638090	0.002114	0.640192
	5	0.448144	0.002399	0.450518
64	1	1.410070	0.001461	1.411363
	2	1.504192	0.001415	1.505575
	3	1.482704	0.001318	1.483973
	4	1.461268	0.001315	1.462538
	5	1.545580	0.001295	1.546817



(a) Read Time vs Processes



(b) Computation Time vs Processes



(c) Total Time vs Processes

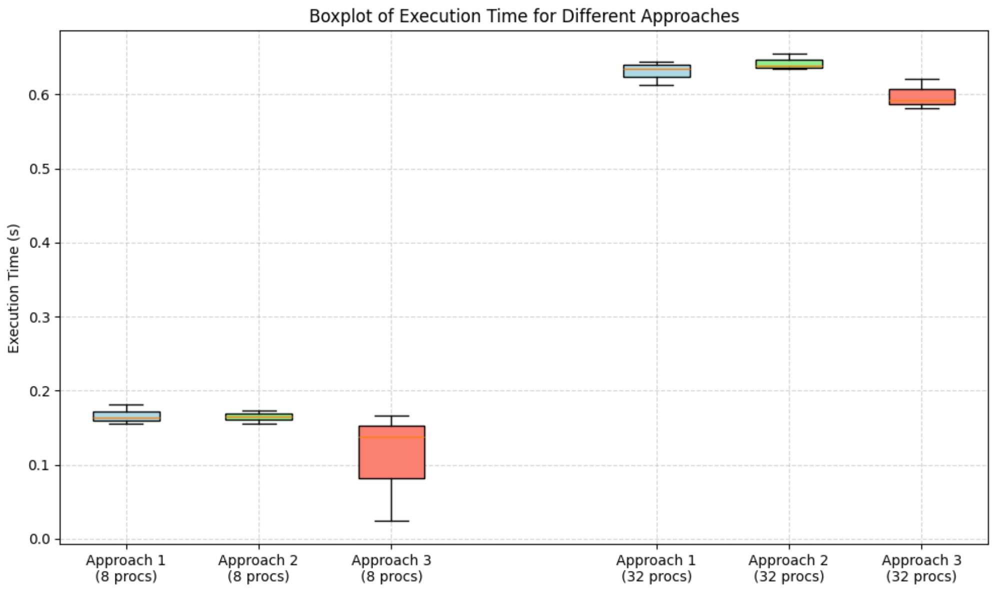
Figure 2: Box plots of times for different stages of MPI execution for data_64_64_64_3 dataset.

In both section 4.1 and 4.2, while executing the code, we ensured that N * nsteps per node is equal to number of processes, np. It is observed that the computation time decreases as the number of processes increase. This is because the domain size decreases. On the other hand, the read time increases because of collective overhead of the MPI_File_read_all function.

4.3 Code Optimization Results

The following table shows the average read time of three executions run on the data_64_64_96_7 dataset on following the approaches described in Section 3.

Processes	Approach 1 (s)	Approach 2 (s)	Approach 3 (s)
8	0.152609	0.164817	0.124012
32	0.630780	0.642756	0.598580



It is observed that the parallel input strategy is slightly faster than the other 2 strategies. For even bigger dataset, the difference in performance might be even higher.

5 Conclusions

Our project was divided into two primary segments: developing a logic for efficient neighbor communication and identifying an optimal strategy for data decomposition and reading. Three team members focused on exploring different approaches for the data reading strategy, while one member dedicated their efforts to optimizing neighbor communication. While our individual contributions were distinct, we collaborated closely while implementing the logic and debugging each other's code. It was a collective effort that ensured the completion of the project.