CS633 Group Project

Group 16

Team Members

Name	Roll No.	Email
Arshit	220209	arshitsk22@iitk.ac.in
Dobariya Jenil Bharatbhai	220385	dobariyajb22@iitk.ac.in
Harsh Agrawal	220425	harshag22@iitk.ac.in
Prem Kansagra	220816	premk22@iitk.ac.in
Priyanshu Singh	220830	spriyanshu22@iitk.ac.in

Indian Institute of Technology Kanpur

April 2025

1 Code Description

First We'll explain some basic definitions and functions and then move on to explaining the full code logic:

1.1 'Point' Struct Definition

```
typedef struct {
    int x, y, z;
} Point;
```

This 'Point' struct is used to represent 3D coordinates. It helps when dealing with both global and process grids.

1.2 Helper Functions

pos_to_coords

```
Point pos_to_coords(int i, int X, int Y, int Z) {
    //logic to convert
}
```

This function converts a 1D index `i` into 3D coordinates (x, y, z) in a grid of size $X \times Y \times Z$, assuming that 3D grid is stored in xyz format in the array.

coords_to_pos

```
int coords_to_pos(Point coords, int X, int Y, int Z) {
    // logic to convert
}
```

This function is the inverse of `pos_to_coords`, converting 3D coordinates back to a 1D index.

Note:

The above two functions are used in the following contexts:

1. To map actual 3D data coordinates to their linear representation.($X = N_x, Y = N_y, Z = N_z$)
2. To convert the rank of a process into its corresponding 3D coordinates, based on the decomposition of the 3D space into smaller cuboids. ($X = P_x, Y = P_y, Z = P_z$)

1.3 Program Flow

Each process retrieves its own rank and the total number of processes:

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

The program expects:

- `input_filename`
- `PX, PY, PZ` : dimensions of the process grid
- `NX, NY, NZ` : dimensions of the global grid

- `NC` : number of test cases
- `output_filename`

Elements per Rank

Each rank receives:

$$\texttt{elements\_per\_rank} = \frac{NX \times NY \times NZ}{PX \times PY \times PZ}$$

Reading Data

Without any optimization, only rank 0 reads the input data from the file using the helper function `read_input_file`. This function allocates a 2D array of dimensions `NC × (NX × NY × NZ)`, where each row represents one test case. The global data is stored such that each rank's data chunk is laid out contiguously in memory, which enables efficient scattering later.

The core idea is to iterate over all global indices, convert them to 3D coordinates, and determine which rank is responsible for storing each element. The responsible rank is computed as follows:

$$(x, y, z) = \texttt{pos\_to\_coords}(i, NX, NY, NZ)$$

The rank responsible for storing a particular coordinate is determined by first computing the corresponding rank's coordinates in the process grid:

$$\texttt{rank\_coords} = \left(\left\lfloor \frac{x}{NX/PX} \right\rfloor, \left\lfloor \frac{y}{NY/PY} \right\rfloor, \left\lfloor \frac{z}{NZ/PZ} \right\rfloor \right)$$

$$\texttt{responsible\_rank} = \texttt{coords\_to\_pos(rank\_coords, PX, PY, PZ)}$$

An auxiliary array `idx[]` of size of total number of processes $PX \times PY \times PZ$ is maintained, where `idx[r]` keeps track of how many elements assigned to rank r have been read so far. This is used to ensure that data meant for each rank is stored contiguously within the buffer.

As values are read from the file using `fscanf`, they are placed into the appropriate rank's memory segment.

Then, `idx[responsible_rank]` is incremented. This ensures the layout of `global_data[i]` is directly compatible with `MPI_Scatter`.

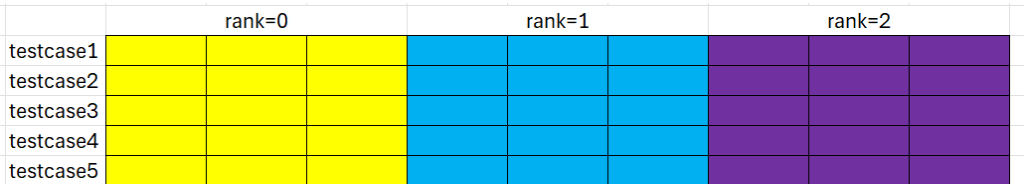


Figure 1: Layout of `global_data` after reading from file.

> Note:
>
> We have mentioned the ***optimized code*** in section 3 i.e. *Optimization using Parallel I/O*

Distributing Data

Once the data is read and laid out in rank-contiguous format, rank 0 distributes the relevant portion of each test case to all ranks using `MPI_Scatter`. Each rank receives a buffer of size:

$$\texttt{elements\_per\_rank} = \frac{NX \times NY \times NZ}{PX \times PY \times PZ}$$

The logic is as follows:

- Rank 0 calls `MPI_Scatter` on each test case array `global_data[i]`, sending each rank its contiguous slice.
- Other ranks call `MPI_Scatter` with `sendbuf = NULL`, as required by the MPI standard.
- Each rank stores its received data in `local_data[i]` for all test cases.

After scattering, rank 0 deallocates the global buffer to free memory.

local_data[0]	testcase1			
local_data[1]	testcase2			
local_data[2]	testcase3			
local_data[3]	testcase4			
local_data[4]	testcase5			

Figure 2: Layout of `local_data` after distributing(scatter) from file.

Halo Layer Allocation

Each process allocates six halo buffers for communication with its 6 direct neighbors (in $\pm X$, $\pm Y$, and $\pm Z$ directions).

```
//for sending
double** X_plus_ones_layer_send;
double** X_minus_ones_layer_send;
double** Y_plus_ones_layer_send ;
double** Y_minus_ones_layer_send;
double** Z_plus_ones_layer_send;
double** Z_minus_ones_layer_send;

//for receiving
double** X_plus_ones_layer_recv;
double** X_minus_ones_layer_recv;
double** Y_plus_ones_layer_recv;
double** Y_minus_ones_layer_recv;
double** Z_plus_ones_layer_recv;
double** Z_minus_ones_layer_recv;
```

Neighbor Rank Computation

A process computes its own coordinates in the process grid:

$$\texttt{rank\_coords} = \texttt{pos\_to\_coords(rank, PX, PY, PZ)}$$

Then, to get neighbor rank (e.g., in -X direction):

$$\texttt{neighbor\_coords} = \{x - 1, y, z\}$$

$$\texttt{neighbor\_rank} = \texttt{coords\_to\_pos(neighbor\_coords, PX, PY, PZ)}$$

Halo Exchange

Halo data is exchanged using:

```
MPI_Send(send_buffer, size, MPI_DOUBLE, neighbor_rank, tag, MPI_COMM_WORLD);
MPI_Recv(recv_buffer, size, MPI_DOUBLE, neighbor_rank, tag, MPI_COMM_WORLD, &status);
```

Max-Min Calculation

```
void isLocalMaxMin(bool *isMin, bool *isMax, float *local_data,int local_index,
    ↪ int NX, int NY, int NZ, int PX, int PY, int PZ,float *
    ↪ X_plus_ones_layer_recv,float *X_minus_ones_layer_recv,float *
    ↪ Y_plus_ones_layer_recv,float *Y_minus_ones_layer_recv,float *
    ↪ Z_plus_ones_layer_recv,float *Z_minus_ones_layer_recv,Point rankCoordinates
    ↪ )
```

For every sub-domain, there are two types of data points- boundary points and non-boundary points.Our function initially checks for boundary points and then for non-boundary points.

- If the point lies on a subdomain boundary, it checks the values of neighbor points that might reside in another process, using the corresponding `*_layer_recv` arrays.If a neighbor's value violates the maximum/minimum condition, the corresponding flag is set to false.
- After the boundary neighbor checks, the point is tested against its six direct neighbors (left, right, up, down, front, back) inside the local subdomain.If any neighbor value contradicts the condition for a maximum or minimum, the flags are adjusted accordingly.

Reduce and final writing to Output file

```
MPI_Reduce(local_minimaCount,global_minimaCount,NC,MPI_INT, MPI_SUM,0,MPI_COMM_WORLD);
MPI_Reduce(local_maximaCount,global_maximaCount,NC,MPI_INT, MPI_SUM,0,MPI_COMM_WORLD);
MPI_Reduce(local_maximum,global_maximum,NC,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
MPI_Reduce(local_minimum,global_minimum,NC,MPI_DOUBLE,MPI_MIN,0, MPI_COMM_WORLD);
```

Finally all processes reduce their local results(local minima count, local maxima count,max value, min value etc.) to rank 0 and rank 0 writes these results to output file.

2 Code Compilation and Execution Instructions

The submitted code is contained in `src.c`. To begin, access the *param rudra* server via SSH using the following command:

```
ssh -p4422 <username>@paramrudra.cdacdelhi.in
```

Replace `<username>` with your assigned account username. Upon connection, you will be prompted to enter a string CAPTCHA, followed by your account password.

Once logged in, create a new file named `src.c` and paste the contents of the solution file provided by Group 16 into it.

To compile the code using the MPI library (assuming it is installed as per the tutorial provided by ma'am and TAs), use the following command:

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

If this compiles successfully, it will create the required executable.

Next, create a job submission script named `job.sh` with the following structure:

```
#!/bin/bash
#SBATCH -N 2
#SBATCH --ntasks-per-node=32
#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`
mpirun -np 8 ./executable data_64_64_64_3.bin.txt 2 2 2 64 64 64 3
    ↪ output_64_64_64_3_a8.txt
echo `date`
```

The first set of `#SBATCH` directives define the computational resources requested from the supercomputer. These parameters (e.g., number of nodes, tasks per node, wall-clock time, etc.) can be modified based on specific needs.

The last three lines are crucial:

- The first and third `echo `date`` statements print the start and end times of the job execution.
- The middle line runs the MPI program using the format:

```
mpirun -np <total_processes> ./executable <input_filename> <PX> <PY> <PZ> <NX> <NY> <
    ↪ NZ> <NC> <output_filename>
```

This matches the usage constraint defined in the code as:

```
if (argc != 10) {
    if (rank == 0) {
        fprintf(stderr, "Usage: %s <input_filename> <PX> <PY> <PZ> <NX> <NY> <NZ> <NC>
            ↪ <output_filename>\n", argv[0]);
        fprintf(stderr, "Error: Incorrect number of arguments\n");
    }
    MPI_Finalize();
    return -1;
}
```

For example (if the executable is named `a8`) :

```
mpirun -np 64 ./a8 data_64_64_96_7.bin.txt 4 4 4 64 64 96 7 output_64_64_96_7_a8.txt
```

Ensure that the input file (e.g., `data_64_64_96_7.bin.txt`) is present in the working directory before scheduling the job.

Once `job.sh` is prepared, use the following command to submit the job to SLURM (the workload manager and scheduler):

```
sbatch job.sh
```

To monitor the status of your submitted job, use:

```
squeue --me
```

This command displays details such as process ID, job ID, and current status (e.g., Running, Queued). To cancel a running or queued job, use:

```
scancel <PID>
```

After the job completes, it will disappear from the `squeue --me` output. Additionally, SLURM creates two files in the same directory:

- `job.<PID>.err` – contains error messages (if any),
- `job.<PID>.out` – contains standard output from the execution.

These files are helpful for debugging and understanding any failures or timeout issues encountered during the run.

3 Optimization using Parallel I/O

3.1 Optimization attempt 1 [Parallel I/O with scatter]

In the unoptimized version of the code, Rank 0 is responsible for reading the entire input file and then distributing the data to all other processes using `MPI_Scatter`. This approach creates two major bottlenecks:-

- **I/O Bottleneck:** Only Rank 0 performs the file read operation, which means that all data must pass through a single process regardless of the total number of processes. This severely limits scalability, especially for large datasets.
- **Communication Bottleneck:** Once the data is read, Rank 0 must send portions of the data to every other process, creating heavy communication overhead centered on a single process.

In the optimized version, the data reading and distribution are parallelized. Specifically, P_z (the number of processes along the Z-direction) processes read the input file in parallel. Each of these processes is responsible for scattering its portion of the data to a group of Px $\times$ Py processes.

To enable this, a new sub-communicator is created for each group of processes aligned along the Z-direction. This is achieved using the following code:

```
int color = rank / (PX * PY);
MPI_Comm new_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, rank, &new_comm);
```

This splits the global communicator (`MPI_COMM_WORLD`) into P_z sub-communicators, where each sub-communicator contains all processes sharing the same Z-slice.Within each sub-communicator,new Rank 0 is responsible for scattering the data to the remaining ranks.

3.2 Optimization attempt 2 [I/O Per Process]

This optimized approach eliminates the need for data redistribution. Each MPI process calculates its local subarray in the 3D dataset and reads it directly from the binary file using MPI I/O. We use `MPI_Type_create_subarray` to define each process's view of the file, allowing for efficient, collective I/O using `MPI_File_read_all`.

Table 1: Comparison of Attempt 1 and Final Attempt Data Reading Strategies

Aspect	**Baseline Approach (Pz-Scatter)**	**Optimized Approach (MPI Subarray)**
I/O Participation	Only P_z processes read data	All processes read their subarrays in parallel
Data Distribution	Requires scattering from P_z processes to $P_x \times P_y$ group	No explicit communication needed; direct data access
Sub-communicator Usage	Requires communicator per Z-aligned group	No additional communicators needed

4 Results

Table 2: Sequential I/O Timings for data_64_64_64_3.txt

NP	Iteration 1			Iteration 2			Iteration 3		
	Read	**Main**	**Total**	**Read**	**Main**	**Total**	**Read**	**Main**	**Total**
8	0.147465	0.012917	0.268982	0.147749	0.013171	0.265800	0.148067	0.013012	0.161059
16	0.147830	0.007424	0.155297	0.147859	0.006926	0.262840	0.147942	0.007483	0.155452
32	0.631600	0.004410	0.635955	0.458962	0.004424	0.570129	0.551658	0.004738	0.556331
64	1.884142	0.003473	1.886673	1.897351	0.004333	1.900744	1.852665	0.003718	1.855354

Table 3: Parallel I/O Timings for data_64_64_64_3.txt

NP	Iteration 1			Iteration 2			Iteration 3		
	Read	**Main**	**Total**	**Read**	**Main**	**Total**	**Read**	**Main**	**Total**
8	0.013953	0.014467	0.027215	0.013174	0.014532	0.026768	0.015289	0.014475	0.028553
16	0.016755	0.008452	0.023911	0.016377	0.008108	0.022953	0.017497	0.008074	0.024082
32	0.016582	0.005042	0.020444	0.402439	0.005015	0.406305	0.416846	0.004985	0.420696
64	1.550894	0.008146	1.553199	1.431426	0.004159	1.433565	1.512874	0.004115	1.514938

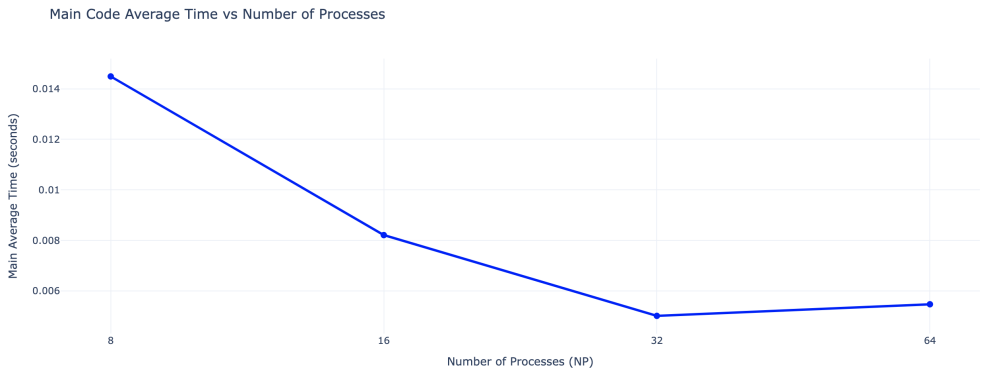
Table 4: Sequential I/O Timings for data_64_64_96_7.txt

NP	Iteration 1			Iteration 2			Iteration 3		
	Read	**Main**	**Total**	**Read**	**Main**	**Total**	**Read**	**Main**	**Total**
8	0.486561	0.04405	0.530479	0.484407	0.043781	0.630415	0.482486	0.044549	0.526854
16	0.483725	0.022833	0.506797	0.484145	0.022746	0.609109	0.48316	0.023528	0.50679
32	0.561838	0.013286	0.678821	0.534182	0.013137	0.547217	0.531983	0.013214	0.65109
64	1.865434	0.00862	1.872606	1.86509	0.008292	1.871766	1.828432	0.008327	1.835394

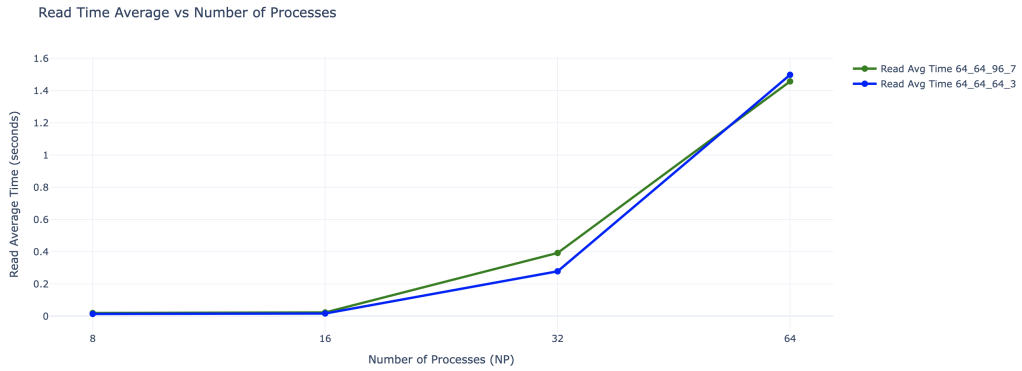
Table 5: Parallel I/O Timings for data_64_64_96_7.txt

NP	Iteration 1			Iteration 2			Iteration 3		
	Read	**Main**	**Total**	**Read**	**Main**	**Total**	**Read**	**Main**	**Total**
8	0.020532	0.04747	0.066693	0.017444	0.047549	0.063766	0.020976	0.047297	0.066991
16	0.02315	0.025971	0.046425	0.024222	0.026253	0.047965	0.022428	0.025697	0.045676
32	0.208765	0.012854	0.222135	0.485711	0.017969	0.498981	0.482943	0.01797	0.495987
64	1.447065	0.11423	1.454235	1.398947	0.012856	1.406684	1.526345	0.011474	1.533293

Analysis



- `Main Code Runtime Analysis -` The chart shows how the main code's average execution time varies with the number of parallel processes (NP). As NP increases from 8 to 32, there is a clear decrease in execution time, indicating effective parallelization where the workload is efficiently distributed among processes. However, at NP = 64, the average time slightly increases instead of decreasing further. This suggests the onset of parallel overheads such as increased inter-process communication, synchronization delays, and contention for shared resources. These factors can outweigh the benefits of adding more processes, leading to a point of diminishing returns in performance scaling. Thus, while parallelism improves performance up to a point, excessive process counts can reduce efficiency.



- `File Read Runtime Analysis -` The average read time increases significantly with the number of processes. At lower process counts (NP = 8 and 16), read times are low and manageable. However, as NP increases to 32 and then 64, there is a sharp rise in read time, growing from milliseconds to over a second. This trend indicates that reading data becomes a major bottleneck in higher parallel configurations. Unlike computation, which often benefits from parallelism, I/O operations may suffer when too many processes attempt to read simultaneously, leading to degraded performance.

- `Performance Analysis of Parallel and Sequential I/O` The analysis below corresponds to the experiment conducted for a computational setup with **NX = 64, NY = 64, NZ = 96, and NC = 7**. The performance comparison is made between Parallel and Sequential I/O in terms of *total average time* taken (in seconds) across different numbers of processors (NP = 8, 16, 32, and 64).

  At lower processor counts, particularly for NP = 8 and 16, the Parallel I/O demonstrates a significant performance advantage. For instance, at NP = 8, the parallel approach records

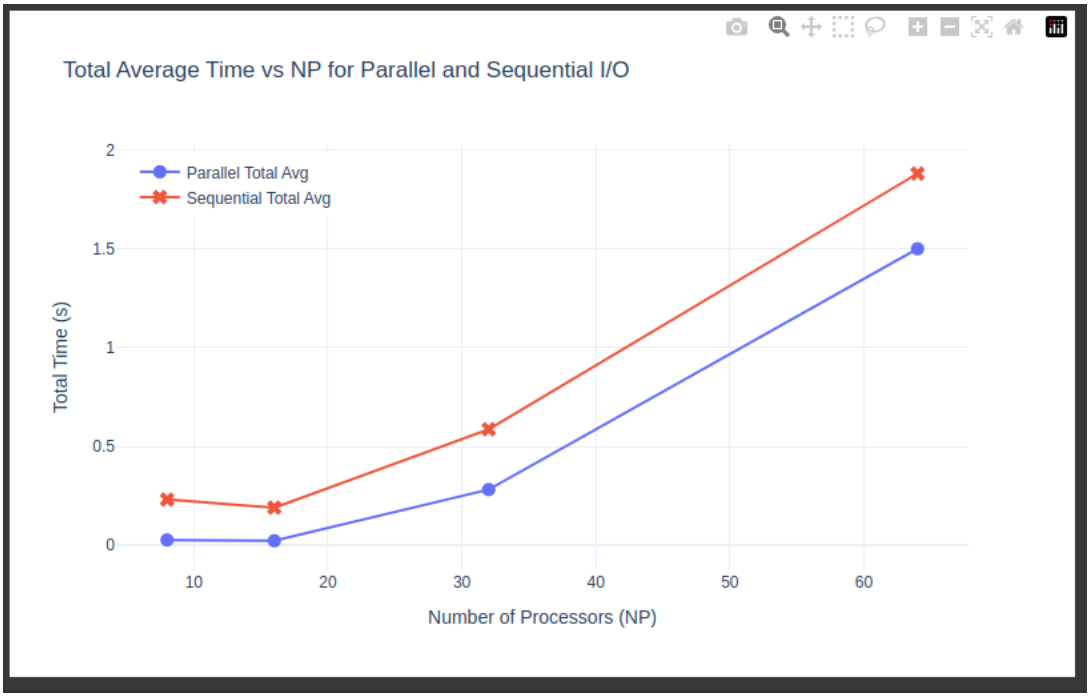


Figure 3: Total Average I/O Time vs NP for Parallel and Sequential I/O

a total average time of only 0.0275 seconds compared to 0.2319 seconds for sequential I/O. This showcases an almost **8.4x improvement** in performance. A similar trend is observed at NP = 16, with parallel averaging 0.0236 seconds versus 0.1912 seconds for sequential, indicating roughly an **8.1x improvement**.

However, at NP = 32, the parallel I/O time increases sharply to 0.2825 seconds, whereas the sequential time is 0.5875 seconds. While parallel I/O still performs better, the gain is not as significant, suggesting potential overheads due to I/O contention, or synchronization delays, at higher levels of parallelism.

When the processor count is increased to NP = 64, both parallel and sequential methods experience a rise in total average time. The parallel I/O records 1.5006 seconds, while the sequential method records 1.8809 seconds. Despite the increase, parallel I/O still outperforms sequential, although the performance gap has narrowed.

In conclusion, the parallel I/O technique provides substantial benefits for lower NP values. However, scalability becomes a concern at higher processor counts, where performance gains begin to plateau or even degrade.

5 Conclusion

The implemented MPI-based solution effectively handles large-scale 3D data by distributing the workload across a 3D process grid. It optimizes performance through efficient halo exchanges and leverages parallel I/O using MPI subarray views to significantly reduce data read times. The approach demonstrates scalability and robustness, making it suitable for high-performance scientific computing applications.