CS633 Assignment Group Number 05

Aryan Agarwal (241110012)
Harsh Baid (241110026)
Tanuj Agarwal (241110076)
Vinayak Devvrat (241110079)
Khushboo Agrawal (242110605)

April 14, 2025

1 Code Description

We aim to solve the problem of finding in parallel maxima and minima of the time series data of 3D Volume. We have implemented two versions of code- Sequential Read and other is Parallel Read. We have explained working of both approaches, by building on learnings from Sequential Code and finally optimising our code with Parallel I/O and doing analysis on obtained results. We have used certain terminology to refer different times reported and datasets used, which is explained in further sections.
Our code is highly modularised as we have delegated the different phases of the implementation to different functions. We describe our algorithm in detail in the following subsections along with key working of the functions as well.

1.1 Code for Sequential Read

In this approach only 1 process (process with Rank 0) reads the entire data and distributes it to respective processes.

1.1.1 Pseudocode

Algorithm 1 represents the driver routine for sequential read of data.

1.1.2 Key User Defined Functions

The key user defined functions helps in performing different tasks including array allocation, domain decomposition, halo exchange for boundary data points across process, writing function etc. These options help make the code modular and easy to read and use in overall code. Some of key functions are listed below:

1. `compute_decomposition()`
   In our Algorithm 1, each process calls the compute_decomposition() subroutine on Line 4. This subroutine first computes the subdomain size (`sub_nx`, `sub_ny` and `sub_nz`). It then computes the coordinates of the start point(`start_x`, `start_y` and `start_z`) by computing the offset depending on the rank of the invoking process. It also computes the rank numbers of neighboring process for each process and stores it in `neighbors[ ]`.

2. `read_data()`
   The process with rank 0 first allocates a 4D array on Line6. It then reads the file on Line 8 in Algorithm 1. The data in input file is stored it in a array initialised on Line 6.

3. `create_sendbuf()`
   For each timestep, process with rank 0 calls the subroutine compute_sendbuf on Line 14. This subroutine packs the data of 3D subdomain corresponding to each process and returns sendbuf[].

4. `distribute_data()` On Line 17, for each timestep each process calls distribute_data subroutine. This subroutine invokes MPI_scatter(sendbuf,0) function to scatter data to all process with root rank 0. Each process receives the data in its recvbuf.

Algorithm 1 Sequential Code

```
1: function DRIVER_FUNCTION(inputs)
2:     MPI_Init();
3:     input.txt, px, py, pz, Nx, Ny, Nz, Nc ← Input();
4:     start_id, neighbor[ ]←compute_decomposition();
5:     if rank == 0 then
6:         Data ←[ ][ ][ ][ ];
7:         file.open('input.txt');
8:         Data = read_file('input.txt');
9:         file.close();
10:    end if
11:    for each timestep from t = 0 to NC − 1 do
12:        if rank == 0 then
13:            for each process P from p = 0 to px ∗ py ∗ pz − 1 do
14:                send_buff[p][t] = create_sendbuf(Data);
15:            end for
16:        end if
17:        recvbuf = distribute_data(send_buff[t], 0);
18:        bound_data[p][t] = exchange_bound_data(neighbor);
19:        local_comp[t] = compute_extrema(recvbuf[p][t],bound_data[p][t]);
20:        MPI_Reduce(local_comp[t],0)
21:    end for
22:    if rank == 0 then
23:        output.txt = file.write(output)
24:    end if
25:    Finalize MPI
26: end function
```

5. `exchange_bound_data()` In the second step each process invokes subroutine exchange_boundary_data() on Line 18 for exchanging the data corresponding to grid points on the border of its subdomain. Each process has six neighboring process located along positive and negative x,y and z-directions.This subroutine invokes MPI_Sendrecv() function calls. Each process exchanges data with neighbors in XYZ order. This subroutine returns bound_data[] for each process.

Each process allocates a 3D array padded (hence the `+2`) to accomodate ghost cell. It then loads recvbuf[] and bound_data[] data onto 3D array. This step ensures that each process has data corresponding to grid points in its subdomain as well as bordering grid points in its neighbor's subdomain required for computation of local and global extrema.
In our Algorithm each process then invokes find_minima_maxima function() on Line 19 for each time step. Each process iterates over all the grid point in its subdomain to calculate 1. count of local minima and maxima and 2. value of global maxima and minima in its subdomain.
All the process in our algorithm firstly invokes MPI_Reduce(result,0) on Line 20 with root rank 0. The process with rank 0 receives computed data from all the process. The process with rank 0 then invokes write_output()subroutine on Line 23. It then writes the counts of local minima and local maxima onto the file first and the values of global minima and global maxima are written in the next line for all time step. The format is shown below:

1. Line 1: `(local_minima_count, local_maxima_count)`
2. Line 2: `(global_minima, global_maxima)`

1.2 Code with Parallel Read (Optimization)

1.2.1 Pseudocode

Algorithm 2 Parallel I/O Implementation

```
MPI_Init();
input.txt, px, py, pz, Nx, Ny, Nz, Nc ← Input();
start_id, neighbor[ ]← compute_decomposition();
for each process P from p = 0 to px * py * pz − 1 do
    Open file using MPI_File_open();
    Define contiguous floatNC type (MPI_Type_contiguous())
    Define subarray filetype (MPI_Type_create_subarray())
    Set file view (MPI_File_set_view())
    Allocate local_buffer
    MPI_File_read_all() to read local data block
    Free MPI datatypes, close file
end for
Allocate arrays: minima_counts, maxima_counts, global_mins, global_maxs
for each timestep t = 0 to NC − 1 do
    Call allocate_3d_array()                          ▷ Allocate local_data with ghost cells
    Fill local_data from local_buffer (slice at t)
    recvbuf_n[] = exchange_boundary_data()            ▷ Update ghost layers
    local_res[] =find_minima_maxima()                 ▷ Compute local extrema
    MPI_Reduce() to gather global extrema
    if rank == 0 then
        Store global minima/maxima counts and values
    end if
    Free local_data
end for
Free local_buffer
Gather maximum timing information across ranks
if rank == 0 then
    Call write_output()                               ▷ Write results to file
end if
Free allocated arrays
Finalize MPI
```

1.2.2 Key Changes Details

In this algorithm each process performs domain decomposition as discussed in Section1.1.2. In this, the algorithm defines subarray to represent each process portion of the global array. It sets the file view so that each process can read the data corresponding to grid points in its subdomain. A local buffer of size of subdomain is allocated and data is read collectively using MPI_ File_read_all. As each process has read data for grid points in its subdomain, there is no communication for distribution of data. The algorithm for halo exchange and computation of maxima and minima is same as explained before.

2 Code Compilation and Execution Instructions

A single src.c file with complete code is submitted per the instructions. Steps to run the code:

1. Unzip the submission package:
   `unzip <filename>.zip`
   `cd <unzipped-folder>`

2. Compile the source code using MPICC:
   `mpicc src.c`

3. Run the main job file:
   `sbatch job.sh`

This script will compile and execute the code with all configurations as given in the assignment for both datasets. The output will be generated under the Latest_Outputs/ directory.

(Optional) If someone wants to run any individual test case separately:

`sbatch <job file of the corresponding test case>`

Input files, output files and even job files have been added in separate folders.

3 Code Optimizations

In this section we discuss the result obtained and optimizations performed for improving on the efficiency of code. Before discussing optimisations, few terminology used in further sections and figures are explained below. (Please note that we have followed the same logic as explained in the assignment PDF for reporting times. The three times that we report in the output files, are here on refered to as Time 1 , Time 2 and Time 3 respectively.)

1. Time 1: This means time taken by processes to read data from input file and distribute data such that each process has data corresponding to grid points in its subdomain. This is the first time that is asked to be reported in output file.

2. Time 2: This includes rest of code before output. This is the second time that is asked to be reported in output file.

3. Time 3: This includes the total time taken from reading of the data till write of computed result. This is the third time that is asked to be reported in output file.

4. Seq: This means our Sequential Code technique/approach. We refer to it as Seq to compare across different approaches with different optimisations.

5. Para: This technique means Parallel Read approach is implemented as per Algorithm 2.

6. leader: This technique means Parallel Read approach is implemented along with Group Leader only reading the file for given offsets for its group's process, and later distributes this data to its group members.

7. dataset 3: This refers to the data_64_64_64_3.bin.txt dataset where 3 indicated NC or number of timesteps

8. dataset 7: This refers to the data_64_64_96_7.bin.txt dataset where 3 indicated NC or number of timesteps .

9. dataset 128: This refers to a new dataset we made from dataset 7 to test scaling of our code on larger dataset to see improvements in parallel read approach. It was made by making Dataset-7 8 times bigger and hence each dimension got scaled to data_128_128_192_7.bin.txt

10. configurations like 8 process with 2 2 2 split is referred to as config 2 2 2

The optimisations are below-:

3.1 Optimisation by Parallel Read over Sequential Read and Analysis

Implemetation is explained in section Code with Parallel Read (Optimization). With this, we saw the massive improvements in results explained below. We observe in Figure 1 that there is a significant improvement in Time 2. This value is taken as average of all 5 trial runs done for each of the configurations (referred in legends). **Eg 0.12 seconds in sequential execution reduced to 0.005 seconds for 8 process configuration. PLEASE NOTE SCALE OF EACH PLOT IS DIFFERENT ON Y AXIS!**

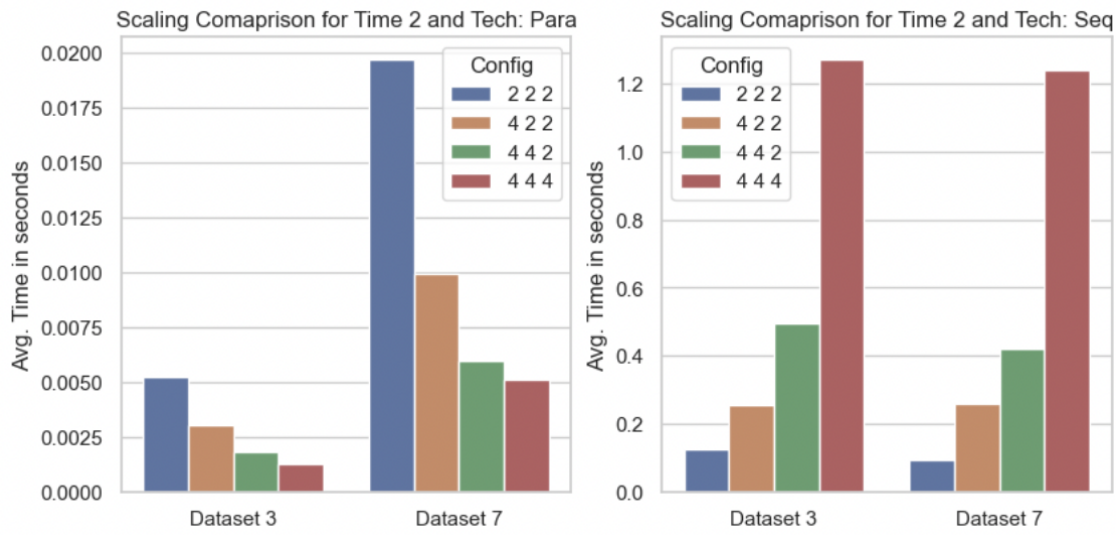

Figure 1: Time 2 Improvement in Parallel Read Code (left) vs Sequential Read Code (right)

3.2 Other Optimisations Attempts and Analysis

- We have implemented (not submitted) parallel read using group leaders such that each leader reads chunk of data, and distributes data to its respective group members. We did not observe much significant improvement, as performance was still same as our previous parallel read approach. This is shown in Figure 2. We expect primary reason for this to be that the dataset is considerably small and therefore the distribution overhead is overriding the time saved by reducing the number of read calls.
- We also implemented MPI_Isend and MPI_Irecv for our Halo Exchange function, but we did not observe any significant improvements, as we did not have any computation code in-between Isend and Irecieve calls, and hence it was ultimately followed by a MPI_Waitall call. We did not incorporate this function call in our final code.

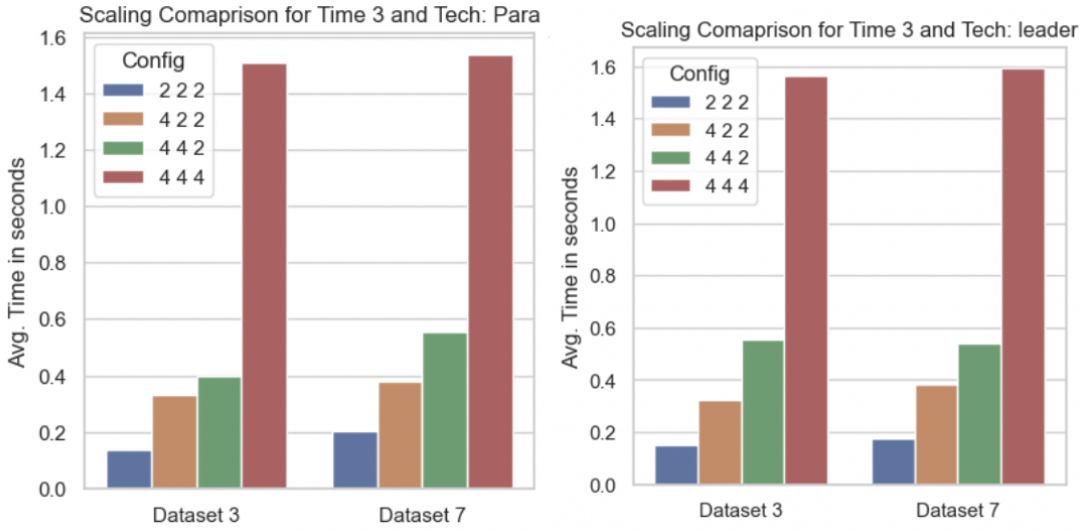

Figure 2: Parallel Read with Group Leaders Approach Performance Comparison

-

4 Results

Tables 1 and 2 show the results obtained for time series data of 3D Volume with three and seven timesteps respectively.

Timestep	Local Minima Count	Local Maxima Count	Global Minima	Global Maxima
1	37988	37991	-48.2500	33.6300
2	37826	38005	-51.4500	33.3500
3	37788	37978	-48.5500	33.3500

Table 1: Local maxima and minima counts and values of extrema in each timestep

Timestep	Local Minima Count	Local Maxima Count	Global Minima	Global Maxima
1	56875	56848	-48.2500	33.6300
2	56703	56965	-51.4500	33.3500
3	56680	56847	-48.5500	33.3500
4	56601	56980	-43.1300	32.0000
5	56769	56937	-53.5500	34.0600
6	56657	56950	-49.6800	34.1800
7	56862	56996	-53.5500	34.3400

Table 2: Local maxima and minima counts in each timestep

4.1 Reported Time Results

Table 3 refers to the time values reported in our final output files from our trials, across all configurations for our Parallel Read code. Figure 4 and Figure 5 helps us visualize it better. Analysis on this is explained in Section4.2

Dataset	Configuration	Time 1	Time 2	Time 3
Dataset 3 (`data_64_64_64_3.bin.txt`)	2 2 2 (8 Processes)	0.165973	0.005128	0.171098
	4 2 2 (16 Processes)	0.363770	0.002891	0.366647
	4 4 2 (32 Processes)	0.513718	0.001844	0.515552
	4 4 4 (64 Processes)	1.397759	0.001641	1.398982
Dataset 7 (`data_64_64_96_7.bin.txt`)	2 2 2 (8 Processes)	0.185056	0.019356	0.204404
	4 2 2 (16 Processes)	0.374701	0.010531	0.385226
	4 4 2 (32 Processes)	0.537125	0.006320	0.543436
	4 4 4 (64 Processes)	1.567809	0.011751	1.571303

Table 3: Reported times (in seconds) in datasets 3 and 7 across all configurations

We have also did 5 executions of each configuration, and plotted boxplots to see performance variations. It was observed in few cases, but mostly were in close range variation as seen by width of respective boxplots in figure shown below.

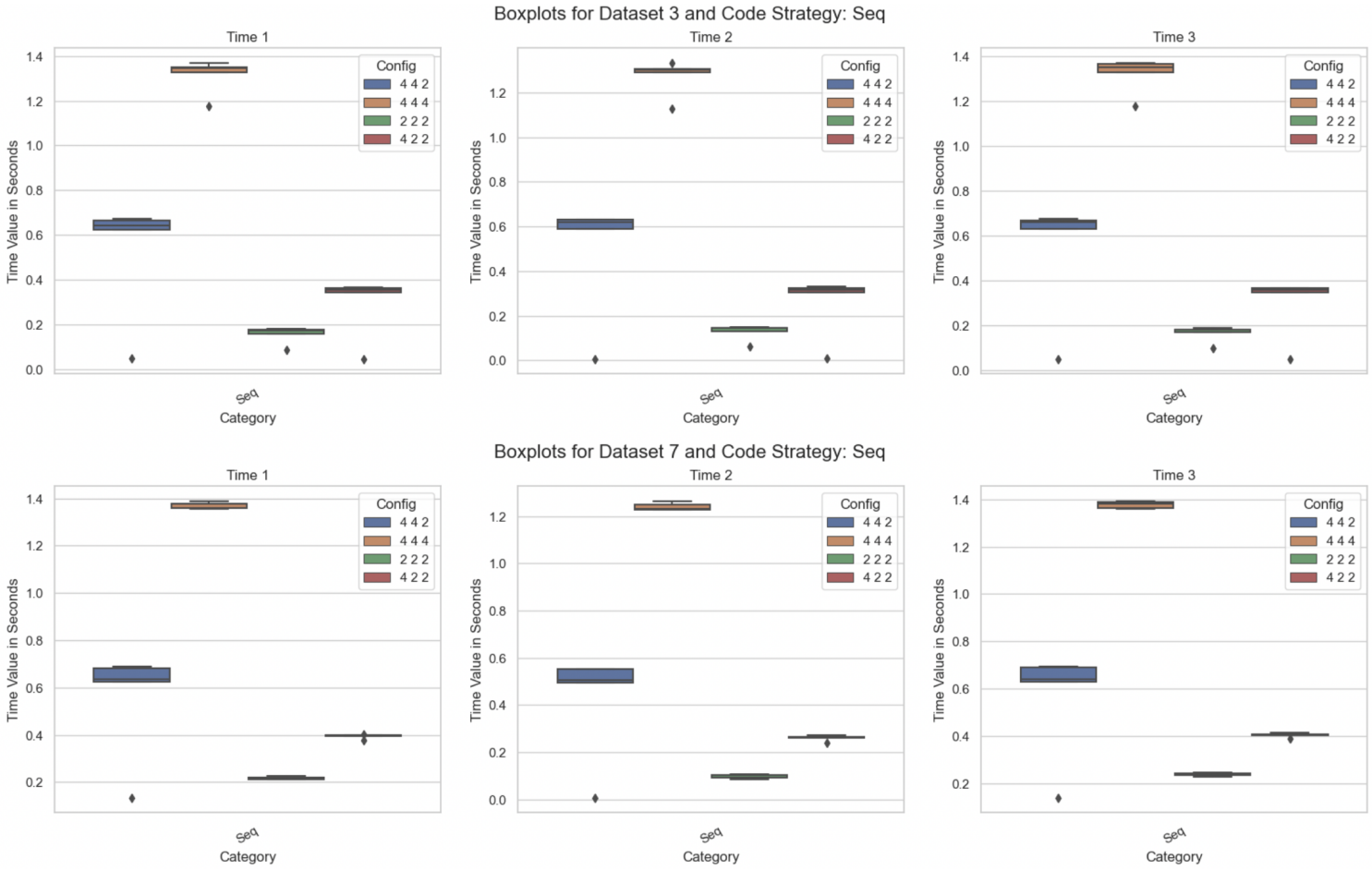

Figure 3: Boxplots of different configurations across datasets 3 and 7 for all three times reported

4.2 Scalability Analysis

Compute time reduces with increasing number of processes. As is evident in Figure4, in both datasets compute time reduces as the number of processes increases.

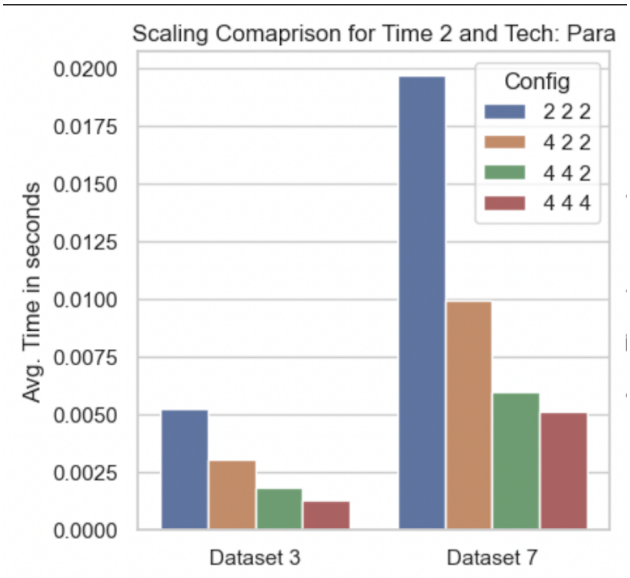

Figure 4: Scaling improvement observed in Average Time 2 as we increase number of processes

There is an over-decomposition for reading processes. The figure 5 clearly shows that as the number of processes increase the time for reading also increases. We speculate this to over decomposition which introduces overheads as a large number of processes are trying a relatively small dataset.

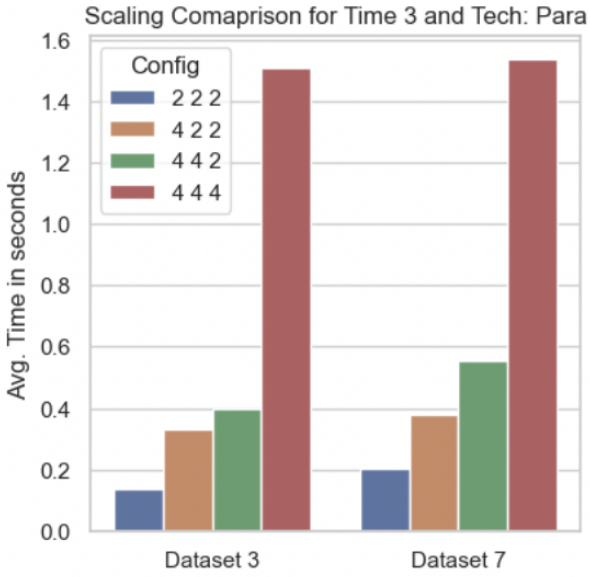

Figure 5: Average Read time increases due to over-decomposition considering small dataset and a large number of processes

4.3 Performance Impact due to Parallel Read

As seen in the figure 1, we attained improved performance over sequential read for all number of processes. We also tested our implementation on a dataset 8 times larger than the dataset 7, and its scaling results are shown figure 6.

It is also evident from this figure that when the given datasets were small, we were not able to see the impact of our parallel read code (nearly identical time values for both datasets). However, on a larger dataset the performance efficiency is more apparent. Observe the reduction between the LHS of the figure as compared to RHS for Dataset 128 in read time (Average over 5 trials of same configuration).

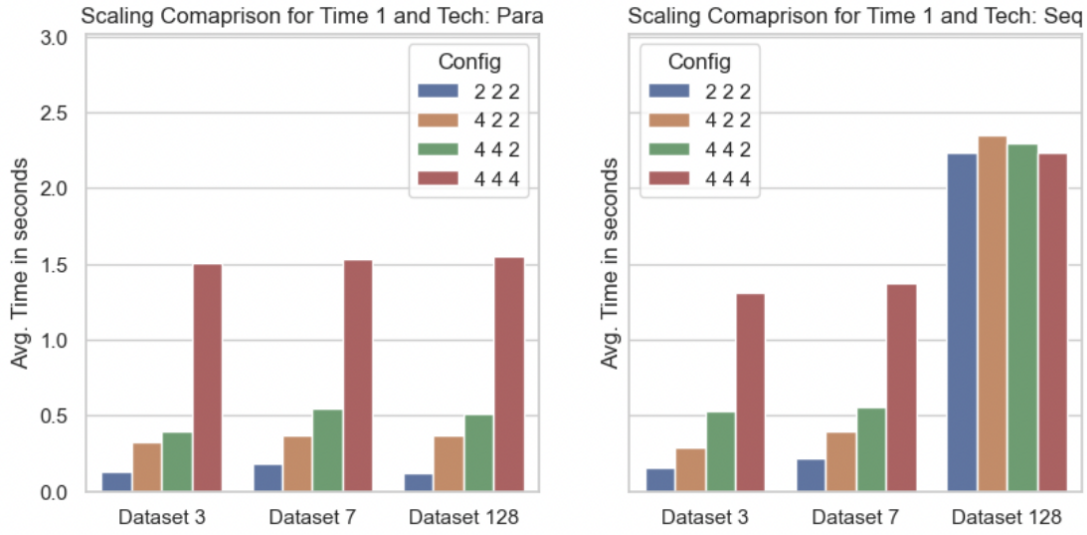

Figure 6: Parallel Read Code (Left) vs. Sequential Read Code (Right) (Shared Y -Axis)

This even more evident in total execution time as shown in figure 7. The reduction in read time has reduced the overall execution time.

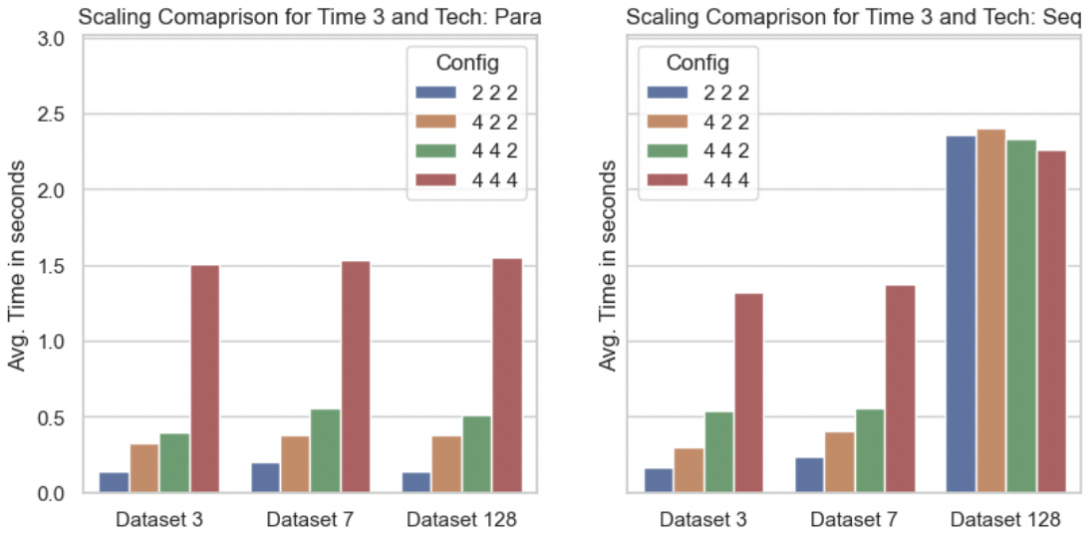

Figure 7: Enhanced Performance than Sequential Read Code (Right) by Parallel Read Code(left) in Total Average Execution Time (Shared Y -Axis)

5 Conclusions

We make the following conclusions from our analysis:

- Our parallel implementation is outperforming sequential code in execution time.
- For larger datasets, our code is performing better. Hence it is considerably scalable.
- We tried multiple optimisations, and based on our learnings and observations from all of them, we have compiled our final code.
- We saw issues of overdecomposition for smaller dataset, and bottleneck of sequential read that was improved with parallel read.

Member contributions

All members worked in synchronisation on different implementations/approaches of the problem statement, all segments of code, debugging and results analysis. The specific contributions are listed below.

Member Name	**Contribution**
Aryan Agarwal	Preparing Sequential and Parallel read code logic, domain decomposition function and results analysis
Harsh Baid	Preparing Sequential and Parallel read code logic, domain decomposition and halo exchange function
Tanuj Agarwal	Preparing Sequential and Parallel read logic, halo exchange function and results analysis
Vinayak Devvrat	Preparing Sequential and Parallel code read logic, computation function and results analysis.
Khushboo Agarwal	Preparing Sequential read logic, computation function and results analysis

Table 4: Team Members and Their Contributions