CS633 Assignment Group Number 07

Udhav Varma 211120 | Rounak Sharma 210876 | Tejus Khandelwal 211111 | Atharv Agarwal 210228

April 14, 2025

1 Code Description

This C code implements a parallel program to compute the count of local minima and maxima in a 3D dataset consisting of 4D points of the form (x, y, z, c) where x, y, z are spatial coordinates and c is a channel index.

- The total number of local minima and maxima within the dataset
- The global minimum and maximum values across all spatial points

The program operates on a three-dimensional Cartesian process grid defined by dimensions `px`, `py`, and `pz`, such that the total number of MPI processes is equal to `px` × `py` × `pz`.

Key steps in the program execution:

1. **Initialization:** The MPI environment is initialized, with each process identifying its rank and the total number of processes. Command-line arguments specifying the input/output file paths, process grid dimensions, and global data dimensions are parsed.

   We use only arrays on the heap, so for ease of access of a 3D/4D array stored as a 1D array, we use a macro to convert a 3D/4D index to a 1D index.

```c
#define GET_INDEX_3D(x, y, z, nx, ny, nz) (z + nz * (y + ny * x))
#define GET_INDEX_4D(x, y, z, c, nx, ny, nz, nc) (c + nc * (z + nz * (y +
    ny * x)))
```

Listing 1: Helper Macros

2.1 **Data Loading and Distribution (Process 0) (`src.c`):** In the Process 0 reads the entire dataset from the binary input file, assuming row-major order with interleaved channel values

$$x_0y_0z_0c_0,\ x_0y_0z_0c_1,\ \dots,\ x_0y_0z_0c_{nc},\quad x_1y_0z_0c_0,\ \dots$$

Now the process 0 will rearrange the data to scatter to all the processes. This rearrangement is dependent on how the data is to be decomposed. We have chosen a 3D decomposition, in a Z-fastest manner - that is, the process 0 takes the the cuboid of data from $(0, 0, 0)$ to $(nx/px, ny/py, nz/pz)$ and the second cuboid takes $(0, 0, nz/pz)$ to $(nx/px, ny/py, 2nz/pz)$ and so on. To facilitate this, we have to rearrange the data for `MPI_Scatter` to work - we do it in the following way:

Suppose we were to arrange the data in a 7D array - `data[px][py][pz][nx/px][ny/py][nz/pz][nc]` such that `data[i][j][k]` represents the cuboid of the process owns the cuboid of data from $(i*nx/px, j*ny/py, k*nz/pz)$ to $(i*nx/px+nx/px, j*ny/py+ny/py, k*nz/pz+nz/pz)$. Now if we were to scatter this data, since the data is in a row-major order, the process 0 will scatter the data in the z-fastest manner, i.e. `data[0][0][0]` goes to process 0, `data[0][0][1]` goes to process 1 and so on, which is exactly what we want.

However using a 7D array is not practical (stack size is limited), so we use a single buffer on the heap (called `rearranged_data`), to index it we use a linear index -

```c
int rearr_idx = c +
                    nc * (local_z +
                        lnz * (local_y +
                            lny * (local_x +
                                lnx * (pz_index +
                                    pz * (py_index +
                                        py * px_index)))));
```

Listing 2: Linear Index

2.2 **Parallel IO (`src_io.c`):** We employ a similar logic to read the data in parallel, first for each process to read its own data, each process creates a displacement array to store the starting indices of segments of data it needs to read. We read the data in $ny/py \times nz/pz$ blocks, and each block has $nx/px \times nc$ elements (since the elements are in x-fastest manner, with the channels adjacent to each other in the file). Therefore the blocklengths array has the same value in every index - $nx/px \times nc$.

We use `MPI_File_set_view` for every process to set its view of the file using the type created by the `MPI_Type_create_hindexed` function and `MPI_File_read_all` is used to read the data to its `local_data` array. However, we would like to have the data to be in a way so that `local_data[x][y][z][c]` is (x, y, z, c), that is we need `local_data` to be z-fastest manner, so we rearrange the data to make it so.

3. **Boundary Data Exchange:** Each process identifies the data points located on the six faces of its local 3D data block, referred to as boundary data. It then populates six corresponding buffers (left ($x = 0$), right ($x = nx/px - 1$), top ($y = ny/py - 1$), bottom ($y = 0$), front ($z = 0$), back ($z = nz/pz - 1$)) with 2D planes extracted from the respective boundary faces.

   Now each sell uses non blocking communication to send and receive the data to each of the 6 neighbours (depending on the coordinates of the process).

```c
if(crd.x > 0){
    MPI_Isend(left, ny/py*nz/pz*nc, MPI_FLOAT, get_pid((coord){crd.x-1, crd
        .y, crd.z}, px, py, pz), 0, MPI_COMM_WORLD, &send_requests[
        send_count++]);
    MPI_Irecv(recv_left, ny/py*nz/pz*nc, MPI_FLOAT, get_pid((coord){crd.x
        -1, crd.y, crd.z}, px, py, pz), 0, MPI_COMM_WORLD, &recv_requests[
        recv_count++]);
}
```

Listing 3: Example (`src.c`): Exchange with left neighbour

4. **Local Extrema Detection:** Each process iterates over its `local_data` array and compares the value at that point with the values of its spatial neighbours, that are either in the same process or received from the neighbours. A local minimum is assumed to be a point where all neighbours have strictly greater values than the point. A local maximum is assumed to be a point where all neighbours have strictly smaller values than the point.

5. **Global Reduction:** To compute global statistics, the local counts of minima and maxima identified by each process are aggregated using the `MPI_Reduce` with `MPI_SUM` as operator. This operation accumulates the total number of local minima and maxima across all processes and delivers the final result to process 0.
   The local minimum and maximum values are reduced using `MPI_Reduce` with `MPI_MIN` and `MPI_MAX` respectively, to find the true global minimum and maximum for each channel at process 0.

6. **Output (Process 0):** Process 0 outputs the values in the format described in the question.

2 Code Compilation and Execution Instructions

2.1 Compilation

```
mpicc -o main main.c
```

2.2 Execution

```
mpirun -np <P> ./main <input_file> <px> <py> <pz> <nx> <ny> <nz> <nc> <output_file>
```

For example, to run with 8 processes arranged in a $2 \times 2 \times 2$ grid on a dataset of size $100 \times 100 \times 100$ with 5 channels:

```
mpirun -np 8 ./main data.bin 2 2 2 100 100 100 5 results.txt
```

3 Code Optimizations

- **Non-blocking Communication:** The neighbouring cell exchange uses non-blocking sends and receives (`MPI_Isend` and `MPI_Irecv`). This allows communication to potentially overlap with computation or other communication preparations.
- **Data Distribution:** The Process 0 (in the case of sequential IO) rearranges the data so that the native optimised `MPI_Scatter` can be used to transfer the data to all the processes in one go.
- **Parallel IO:** We have also implemented parallel IO using `MPI_File_set_view` and `MPI_File_read_all` to read the data in parallel and rearrange it to be z-fastest manner. This technique shows significant speedup in IO time for large datasets (for $256 \times 256 \times 256 \times 2$ and $512 \times 512 \times 512 \times 2$). § 4.3 describes the speedups in detail.

4 Results

4.1 Sequential IO Timing Analysis

The table below shows the breakdown of the maximum time spent in different phases of the execution for various process counts (np).

- T_1: Read by process 0 & Distribute to each process = $(t_2 - t_1)$
- T_2: Main Code time time = $(t_3 - t_2)$
- T_3: Total $(t_4 - t_1)$
- Note: The time $t_4 - t_3$ is the time for the Reduce operations and output, which we are not reporting as it is very small compared to other timings.

Table 1: Maximum Phase Timings (seconds) for Test Case 1

np	T_1 **(Read time)**	T_2 **(Main Code Time)**	T_3 **(Total)**
8	0.0455	0.0153	0.0609
16	0.0453	0.0090	0.0544
32	0.0498	0.0053	0.0553
64	0.0710	0.0046	0.0756

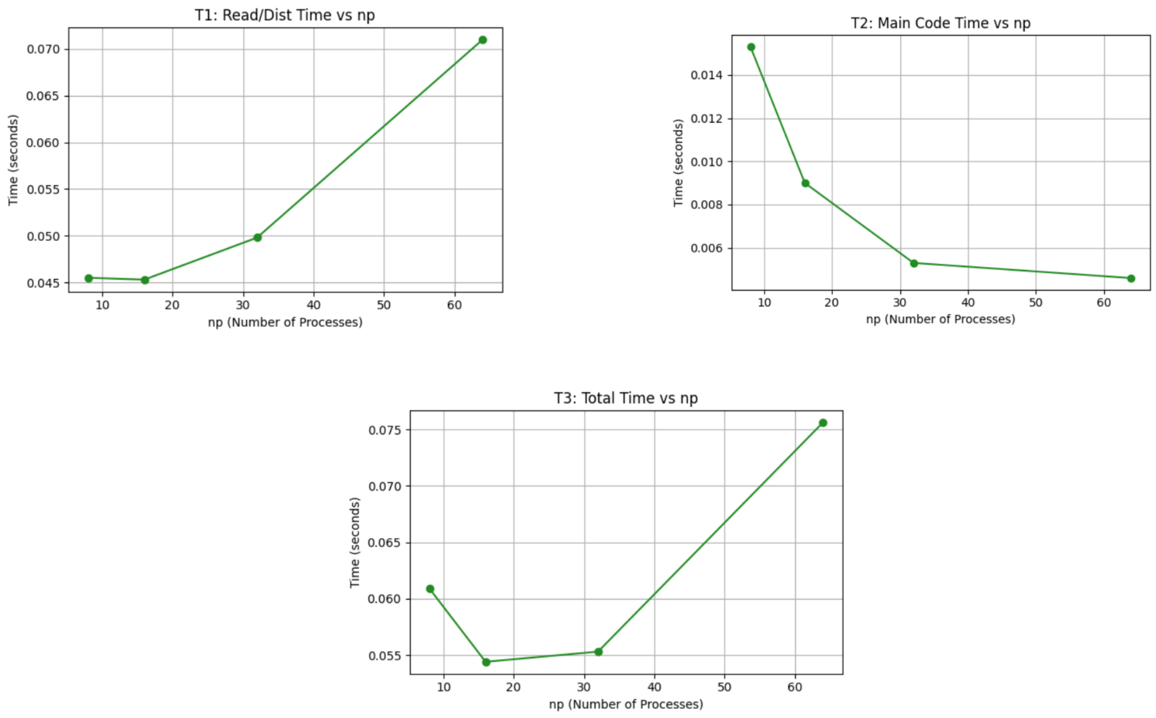


Figure 1: Timings for test case 1 - `data_64_64_64_3.bin`.
Top left: T_1 (Read Time)
Top right: T_2 (Main Computation Time)
Bottom: T_3 (Total Execution Time)

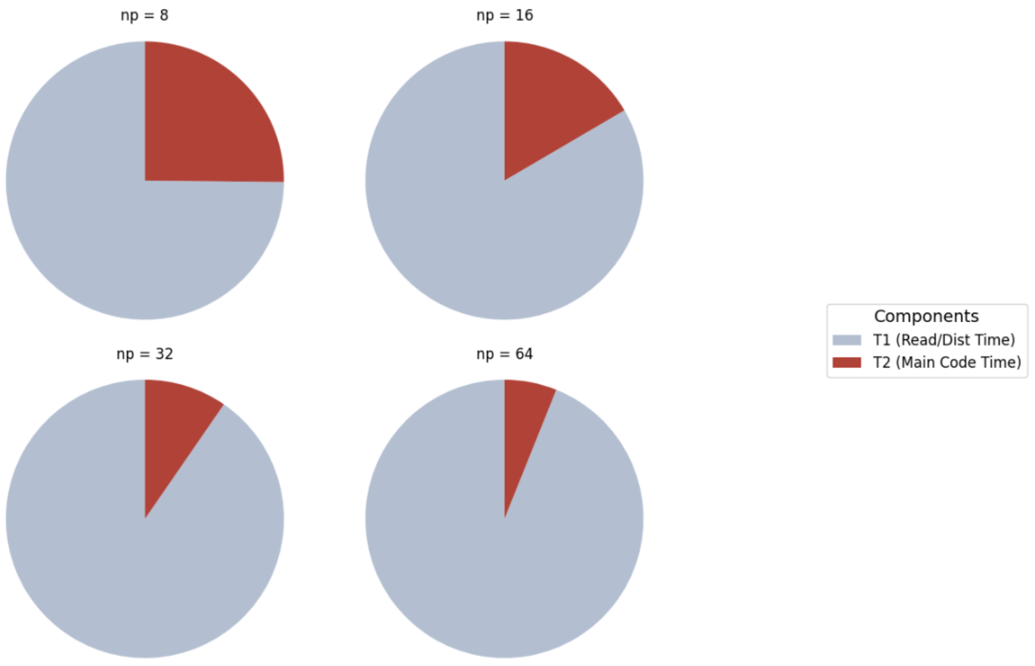
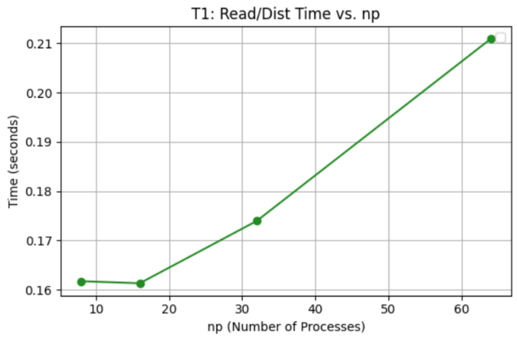
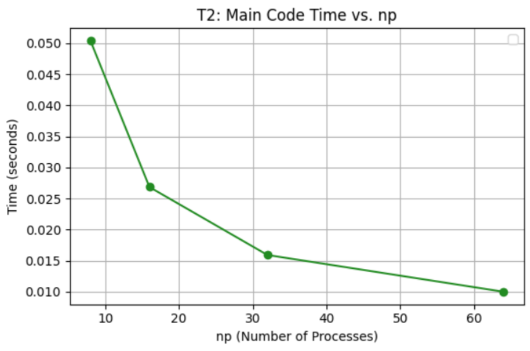


Figure 2: Time Distribution for each process

Table 2: Maximum Phase Timings (seconds) for Test Case 2

np	T_1 **(Read Time)**	T_2 **(Main Code Time)**	T_3 **(Total Time)**
8	0.1617	0.0504	0.2122
16	0.1612	0.0268	0.1881
32	0.1739	0.0159	0.1899
64	0.2109	0.0100	0.2210





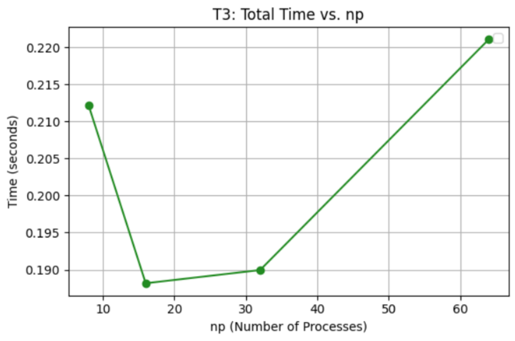


Figure 3: Timings for test case 2 - `data_64_64_96_7.bin`.
Top left: T1 (Read Time)
Top right: T2 (Main Computation Time)
Bottom: T3 (Total Execution Time)

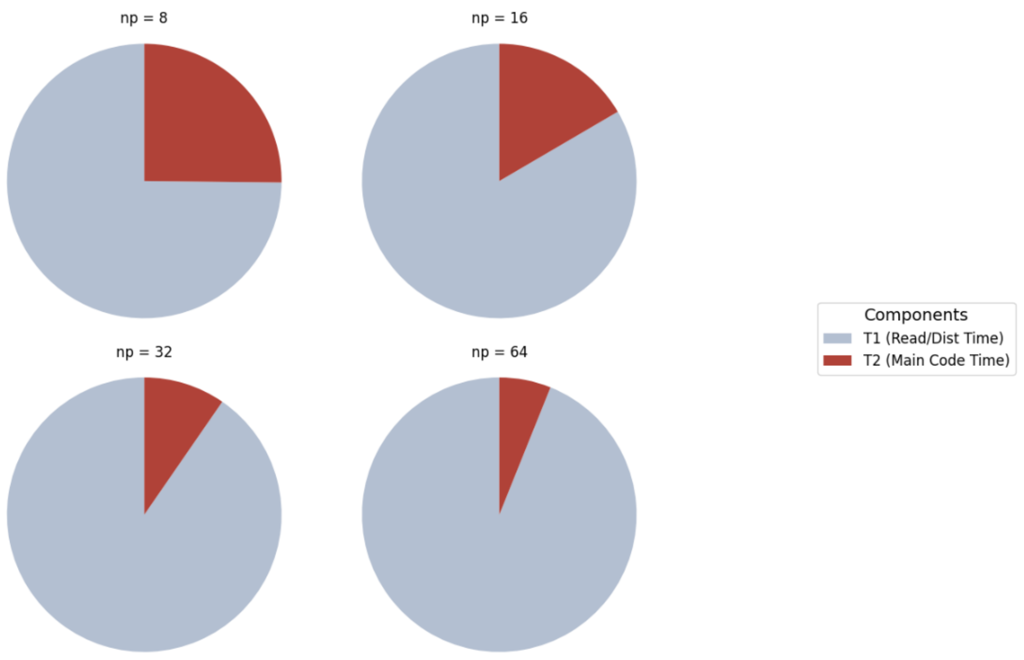


Figure 4: Time Distribution for each process

4.2 Analysis of Sequential IO Results

Here are some of the observation points derived from the tables and plots shown above

- The plots in Figures 1 and 3 show the time spent on reading and distributing data (T_1). Since the data is read by process 0 and then scattered to all other processes, the overall time increases with an increase in read time and the number of scatter calls. The notably higher time observed for 64 processes may be attributed to inter-node communication overhead during the scatter phase.
- Figures 1 and 3 also illustrate the time spent in the main computation phase (T_2), which includes neighbour data exchange and computation of local and global min/max values. The timing scales well with the number of processes; as more processes are used, the computational workload per process decreases, leading to a reduction in overall computation time.
- As shown in figures 1 and 3, the total time decreases when moving from 8 to 16 processes due to a significant reduction in main code time (T_2). However, with 32 and 64 processes, the total time increases. This is because the read time (T_1) becomes a dominant factor, as illustrated in figures 2 and 4. The gain from reduced main code time is outweighed by the increased read overhead, particularly in the 64-process case, leading to higher overall time (T_3).
- We observe that the trends of T_1, T_2, and T_3 with respect to the number of processes are similar between test case 1 and test case 2. However, in test case 2, the magnitude of each time component increases due to the larger input size.

4.3 Parallel IO analysis

In this section we present results of our code using Parallel IO and the rest of the code identical to the sequential IO case.

Table 3: Maximum Phase Timings (seconds) for Test Case 1 with parallel IO

np	**T1 (Read time)**	**T2 (Main Code Time)**	**T3 (Total)**
8	0.0148	0.0146	0.0295
16	0.0174	0.0076	0.0251
32	0.4780	0.0046	0.4826
64	1.9122	0.0051	1.9148

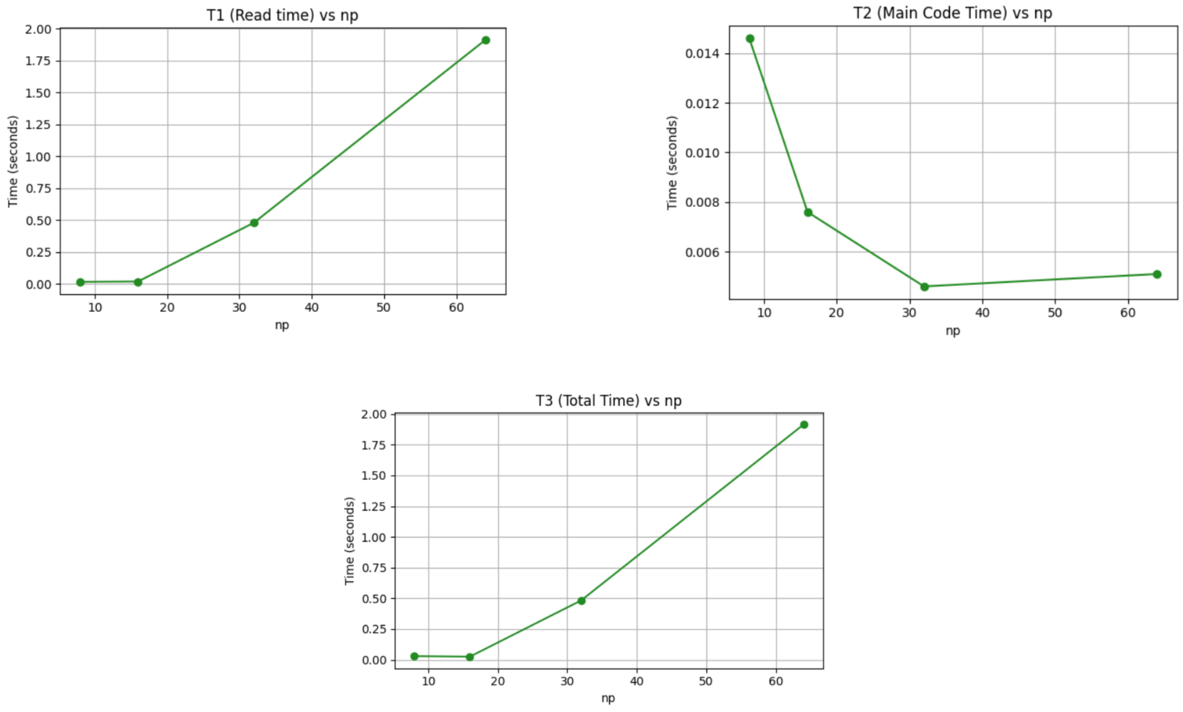


Figure 5: Timings for test case 1 with parallel IO.
Top left: T1 (Read/Dist Time)
Top right: T2 (Main Code Time)
Bottom: T3 (Total Execution Time)

Table 4: Maximum Phase Timings (seconds) for Test Case 2 with parallel IO

np	**T1 (Read time)**	**T2 (Main Code Time)**	**T3 (Total)**
8	0.0278	0.0502	0.0779
16	0.0258	0.0260	0.0517
32	0.4140	0.0147	0.4287
64	1.8659	0.0157	1.8804

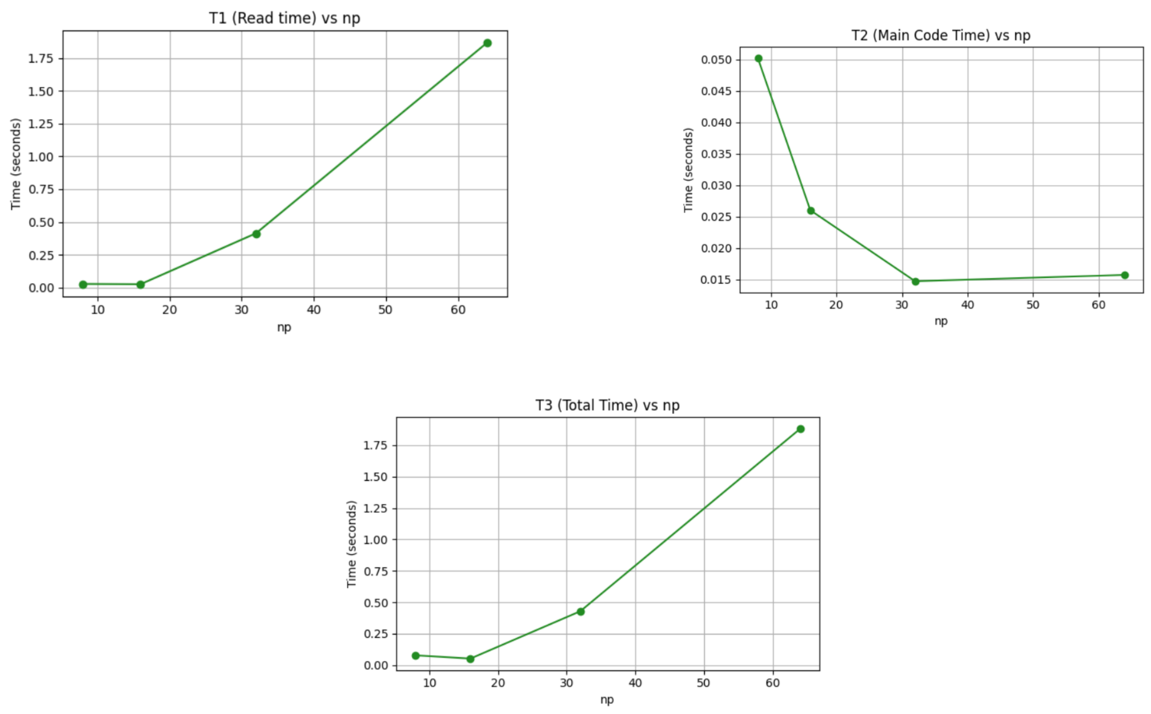


Figure 6: Timings for test case 2 with parallel IO.
Top left: T1 (Read/Dist Time)
Top right: T2 (Main Code Time)
Bottom: T3 (Total Execution Time)

Now we present results for two custom generated data points with dimensions as follows

- Test Case 3: $nx = 256$, $ny = 256$, $nz = 256$, $nc = 2$
- Test Case 4: $nx = 512$, $ny = 512$, $nz = 512$, $nc = 2$

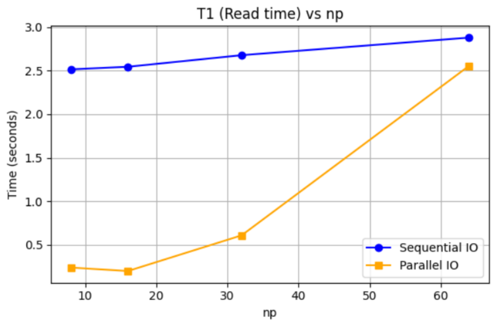
For each test case we first present the timings with sequential IO followed by parallel IO

Table 5: Maximum Phase Timings (seconds) for Test Case 3 with sequential IO

np	**T1 (Read time)**	**T2 (Main Code Time)**	**T3 (Total)**
8	2.5161	0.5809	3.0968
16	2.5453	0.2950	2.8403
32	2.6785	0.1572	2.8357
64	2.8805	0.0912	2.9715

Table 6: Maximum Phase Timings (seconds) for Test Case 3 with parallel IO

np	**T1 (Read time)**	**T2 (Main Code Time)**	**T3 (Total)**
8	0.237762	0.6005	0.815250
16	0.198002	0.3101	0.490362
32	0.607988	0.1644	0.764446
64	2.556088	0.0931	2.634354



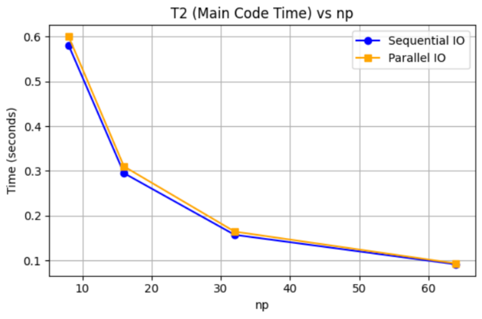


Figure 7: (Left) T1 (Read/Dist Time), (Right) T2 (Main Code Time)

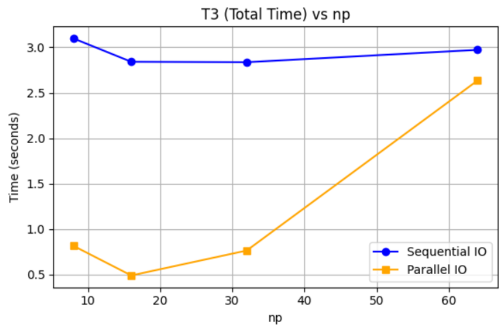


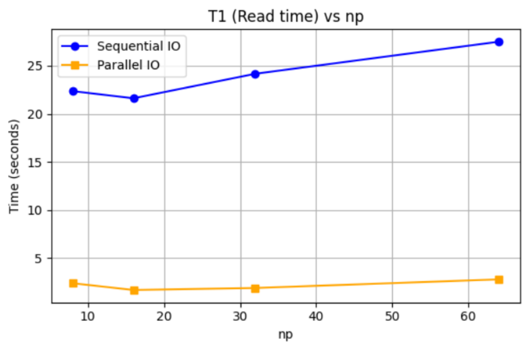
Figure 8: T3 (Total Time)

Table 7: Maximum Phase Timings (seconds) for Test Case 4 with sequential IO

np	**T1 (Read time)**	**T2 (Main Code Time)**	**T3 (Total)**
8	22.361758	4.5829	26.943862
16	21.617371	2.4095	23.942986
32	24.167078	1.2254	25.392287
64	27.502495	0.6980	28.199707

Table 8: Maximum Phase Timings (seconds) for Test Case 4 with parallel IO

np	**T1 (Read time)**	**T2 (Main Code Time)**	**T3 (Total)**
8	2.3764	4.5857	6.9484
16	1.6731	2.3916	3.9833
32	1.8829	1.2607	3.0899
64	2.7774	0.8212	3.4189



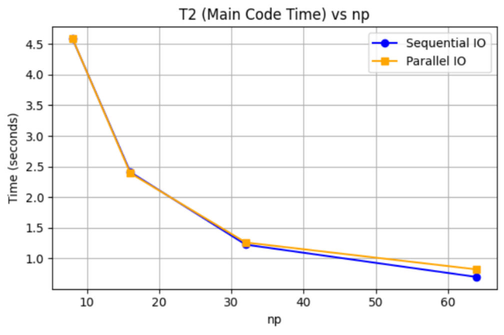


Figure 9: (Left) T1 (Read/Dist Time), (Right) T2 (Main Code Time)

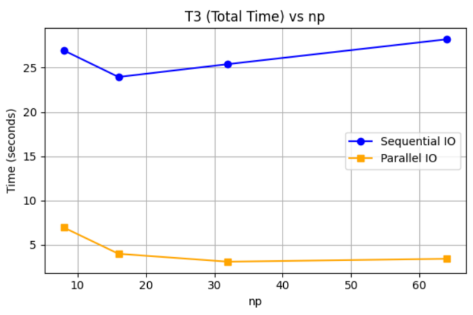


Figure 10: T3 (Total Time)

5 Conclusions

- **Udhav Varma (211120):** [Contribution description]
- **Rounak Sharma (210876):** [Contribution description]
- **Tejus Khandelwal (211111):** [Contribution description]
- **Atharv Agarwal (210228):** [Contribution description]