CS633 Assignment Group Number 15

Team Members:

Pankaj Nath, 221188
Saagar K V, 220927
Venkatesh, 220109
Sai Nikhil, 221095
Rushikesh Chary, 220336

Contents

1. Code Description

1.1. Overview of the Code

The implementation is designed to efficiently process 3D time-series data across multiple nodes using MPI. The code follows a structured workflow:

1. **Initialization and Setup**: Parse command-line arguments, initialize MPI, and establish process grid ($PX \times PY \times PZ$).
2. **Domain Decomposition**: Divide the 3D volume into subdomains with ghost zones for proper boundary processing.
3. **Data Reading**: Implement two strategies based on process count:
   - For fewer processes (≤ 24): Use parallel I/O with MPI derived datatypes
   - For larger process counts (> 24): Root process reads and distributes data using non-blocking sends
4. **Local Computation**: Each process analyzes its subdomain to:
   - Count local minima and maxima for each time step
   - Track global minimum and maximum values
5. **Result Aggregation**: Use MPI reduction operations to gather global statistics.
6. **Output Generation**: Write the consolidated results to the specified output file.

1.2. Our Approach

1.2.1. Domain Decomposition Strategy

In our implementation, we perform a full 3D decomposition of the 3D spatial domain. The global volume, defined over dimensions (X, Y, Z), is divided among all processes in a 3D Cartesian grid layout. If the total number of processes is $P = P_x \times P_y \times P_z$, then each process is assigned a subvolume of size $(X/P_x, Y/P_y, Z/P_z)$.

Along with the above sub-domain we also include ghost zones (halo regions) around each subdomain to accurately determine local extrema at sub-domain boundaries.

```
// Calculate boundaries including ghost zones
subdomain->tempStartX = (subdomain->startX > 0) ? subdomain->startX - 1 :
    subdomain->startX;
subdomain->tempStartY = (subdomain->startY > 0) ? subdomain->startY - 1 :
    subdomain->startY;
subdomain->tempStartZ = (subdomain->startZ > 0) ? subdomain->startZ - 1 :
    subdomain->startZ;

subdomain->tempEndX = (subdomain->endX < nX - 1) ? subdomain->endX + 1 :
    subdomain->endX;
subdomain->tempEndY = (subdomain->endY < nY - 1) ? subdomain->endY + 1 :
    subdomain->endY;
subdomain->tempEndZ = (subdomain->endZ < nZ - 1) ? subdomain->endZ + 1 :
    subdomain->endZ;
```

These ghost zones eliminate the need for communication during the computation phase, improving performance significantly.

1.2.2. I/O Strategies

We implemented two data loading approaches that are selected adaptively based on process count:

1. **Parallel I/O with Derived Datatypes**: For smaller process counts, each process independently reads its portion of data directly from the binary file using MPI's file view mechanism and custom derived datatypes.
2. **Root-based Distribution**: For larger process counts, having too many processes accessing the file system simultaneously could cause contention. Instead, the root process reads the entire dataset and uses non-blocking communication (`MPI_Isend`) to distribute data efficiently.

In both strategies, each process also reads the necessary `ghost cells`—boundary data from neighboring regions—to ensure accurate local extrema detection near subdomain edges. i.e., in the parallel I/O case, the ghost areas are read by corresponding processes and in sequential case, the root process (rank 0), is sending the data along with ghost areas to their corresponding processes. These ghost regions are essential for computations involving neighborhood information.

By dynamically selecting the I/O approach and incorporating ghost data handling, each process obtains all required information to proceed independently, reducing the need for inter-process communication during the computation phase.

1.2.3. Local Extrema Detection:

After loading the data and ghost layers, each process independently scans its subvolume (excluding the ghost regions) to detect local extrema. A point is considered a local minimum (or maximum) if its value is strictly less than (or greater than) all its 6 face neighbors in 3D space for a fixed time slice, and optionally across adjacent time slices if temporal locality is considered. This step is fully parallel and does not require inter-process communication since we have already read the required data (ghost data) from other processes.

1.2.4. Global Result Aggregation:

After each process computes the local minimum and maximum values within its assigned subvolume, we use the `MPI_Reduce` function to aggregate these results across all processes.

Each process contributes its local minimum and maximum to the reduction operations. The `MPI_MIN` and `MPI_MAX` operations are used to determine the overall minimum and maximum values, respectively, across all subvolumes. The final results are collected at the root process (typically rank 0), which then has access to the true global extrema of the entire 3D dataset.

2. Key Functions

This section provides detailed explanations of the most critical functions in our implementation. We present these functions in the order they appear in the program execution flow: starting with domain decomposition, proceeding through data reading and computation, and concluding with result aggregation and output.

2.1. Domain Decomposition

2.1.1. Calculating Subdomain Boundaries

The `calculateSubDomainBoundaries` function is foundational to our domain decomposition strategy. It computes the exact portion of the global 3D volume that each process will handle, including ghost zones for boundary calculations.

Key Implementation:

```
// Calculate process position in the 3D process grid
int posZ = rank / (pX * pY);
int posY = (rank % (pX * pY)) / pX;
int posX = rank % pX;

// Calculate core domain boundaries
subdomain->startX = posX * (nX / pX);
subdomain->startY = posY * (nY / pY);
subdomain->startZ = posZ * (nZ / pZ);

// Calculate boundaries including ghost zones
subdomain->tempStartX = (subdomain->startX > 0) ? subdomain->startX - 1 :
    subdomain->startX;
subdomain->tempStartY = (subdomain->startY > 0) ? subdomain->startY - 1 :
    subdomain->startY;
subdomain->tempStartZ = (subdomain->startZ > 0) ? subdomain->startZ - 1 :
    subdomain->startZ;
```

Usage in Main Program:

```
// In main():
SubDomain subdomain;
calculateSubDomainBoundaries(rank, pX, pY, pZ, nX, nY, nZ, &subdomain);
```

Parameters:

- `rank`: Unique identifier of the current process.
- `pX, pY, pZ`: Number of processes along X, Y, and Z directions.
- `nX, nY, nZ`: Total number of grid points in X, Y, and Z directions.
- `subdomain`: Pointer to a structure where the calculated boundary values are stored.

Logic:

1. Maps the MPI rank to a logical position in the 3D process grid (posX, posY, posZ).
2. Calculates the core subdomain boundaries by dividing the global domain evenly.
3. Handles remainder cells by assigning them to processes at the edge of the domain.
4. Extends boundaries to include ghost zones (one-cell halos) for neighbor access.
5. Computes the dimensions of both the core and extended subdomains.

2.2. Data I/O Strategies

Depending on the number of processes, we implement two distinct strategies for reading input data into the local memory of each process. These approaches ensure efficient I/O across different scales.

```
// In main() - Adaptive I/O strategy selection
if (size <= 24) {
    // Method 1: Parallel I/O with derived datatypes
    localData = readInputDataParallel_Level2(inputFile, &subdomain, nX, nY, nZ
    , timeSteps);
} else {
    // Method 2: Root reads and distributes
    if (rank == 0) {
        globalData = readInputData(inputFile, totalDomainSize, timeSteps);
    }
    localData = distributeData(rank, &subdomain, globalData, nX, nY, nZ,
                               timeSteps, pX, pY, pZ);
}
```

2.2.1. Method 1: Parallel I/O with Derived Datatypes

When the number of processes is less than or equal to 24, we adopt a parallel I/O strategy that utilizes MPI derived datatypes and independent file views. Each process reads its assigned 4D subvolume (Z, Y, X, time) directly from the binary file, including ghost regions.

Key Implementation

```
// Define dimensions for file view
int globalSizes[4] = {nZ, nY, nX, timeSteps};
int subSizes[4] = {subdomain->tempDepth, subdomain->tempHeight,
                   subdomain->tempWidth, timeSteps};
int starts[4] = {subdomain->tempStartZ, subdomain->tempStartY,
                 subdomain->tempStartX, 0};

// Create derived datatype for 4D subarray and set file view
MPI_Datatype filetype;
MPI_Type_create_subarray(4, globalSizes, subSizes, starts,
                        MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native", MPI_INFO_NULL);

// Read data in a single operation
MPI_File_read(fh, localData, localDataSize, MPI_FLOAT, &status);
```

Logic:

- The function `readInputDataParallel_Level2` constructs a 4D subarray datatype using `MPI_Type_create_subarray` to represent each process's data region.
- A file view is set using this datatype to allow a single `MPI_File_read` call to retrieve all relevant data.

2.2.2. Method 2: Root-based Data Distribution

For higher process counts, we implement a two-phase approach: the root process reads the entire dataset, then distributes relevant portions to each process.

Phase 1: Root Process Reads Complete Dataset The `readInputData` function is executed only on the root process to read the entire dataset efficiently:

Key Implementation (Root Process Reading):

```c
// Allocate memory for entire dataset
float* data = (float*)malloc(totalDomainSize * timeSteps * sizeof(float));

// Open binary file
FILE* fp = fopen(inputFile, "rb");

// Optimize file I/O with large buffer
char* buffer = (char*)malloc(8192 * 1024); // 8MB buffer
if (buffer) {
    setvbuf(fp, buffer, _IOFBF, 8192 * 1024);
}

// Read data in large chunks for better performance
const int BLOCK_SIZE = 1024 * 1024;  // Read 1M floats at a time
for (int offset = 0; offset < totalDomainSize * timeSteps; offset +=
   BLOCK_SIZE) {
    int itemsToRead = min(BLOCK_SIZE, totalDomainSize * timeSteps - offset);
    fread(data + offset, sizeof(float), itemsToRead, fp);
}
```

Optimizations in `readInputData`:

- Uses an 8MB buffer with `setvbuf` to reduce system call overhead
- Reads data in large blocks (1M floats) to optimize I/O performance
- Performs a single memory allocation to avoid reallocation costs

Phase 2: Data Distribution to All Processes After reading the full dataset, the root process extracts and sends subdomain data to each process:

Key Implementation (Root Process Distribution Function):

```c
// Inside distributeData function
if (rank == 0) {
    // Prepare non-blocking sends to other processes
    MPI_Request* requests = (MPI_Request*)malloc(numProcs * sizeof(MPI_Request
   ));

    // For each destination process
    for (int p = 1; p < pX * pY * pZ; p++) {
        // Calculate subdomain for this process
        SubDomain recvSubdomain;
        calculateSubDomainBoundaries(p, pX, pY, pZ, nX, nY, nZ, &recvSubdomain
   );

        // Extract data for this process from global data
        // ...

        // Send data using non-blocking communication
        MPI_Isend(sendBuffers[reqIdx], sendDataSize, MPI_FLOAT, p, 0,
                  MPI_COMM_WORLD, &requests[reqIdx]);
    }

    // Wait for all sends to complete
    MPI_Waitall(numProcs, requests, MPI_STATUSES_IGNORE);
} else {
    // Non-root processes simply receive their data
    MPI_Recv(localData, localDataSize, MPI_FLOAT, 0, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Logic:

- Root process extracts and packages data for each target process, including ghost zones.
- Uses non-blocking `MPI_Isend` operations to overlap communication.
- All sends are initiated before waiting for completion, maximizing bandwidth utilization.
- Each receiving process uses a simple blocking `MPI_Recv` with pre-allocated buffer.

2.3. Computational Core

2.3.1. Local Extrema Detection

The `isLocalMinimum` function determines whether a specific point in a 4D spatiotemporal dataset (3D space over multiple time steps) is a local minimum at a given time. It does so by comparing the value at the specified point to its six immediate spatial neighbors (±x, ±y, ±z directions) for the same time index.

Key Implementation:

```
// Get the value at the current point
float value = data[idx * timeSteps + time];

// Check X-axis neighbors
if (x > 0 && data[getLinearIndex(x-1, y, z, width, height, depth) * timeSteps
    + time] <= value)
     return false;
if (x < width-1 && data[getLinearIndex(x+1, y, z, width, height, depth) *
    timeSteps + time] <= value)
     return false;

// Similar checks for Y and Z-axis neighbors
// (omitted for brevity)

// If no neighbor has a smaller value, this is a local minimum
return true;
```

Parameters:

- `data`: Pointer to the flattened 4D dataset stored in 1D array format.
- `x, y, z`: Spatial coordinates of the point under consideration.
- `width, height, depth`: Dimensions of the 3D spatial grid.
- `time`: Time index at which the comparison is performed.
- `timeSteps`: Total number of timesteps in the dataset.

Note: The implementation of the `isLocalMaxima` function follows the same structure, except that it checks whether all neighbors have values *greater than or equal to* the current point, instead of less than.

2.3.2. Processing Local Subdomain Data

The `processLocalData` function processes local data within a subdomain to identify local minima and maxima at each time step. It also keeps track of the extreme values (minimum and maximum) encountered during the process.

Key Implementation:

```
void processLocalData(float* localData, const SubDomain* subdomain,
                      TimeSeriesResults* results, int timeSteps) {
    for (int t = 0; t < timeSteps; t++) {
        int minimaCount = 0, maximaCount = 0;

        // Iterate through subdomain
        for (int z = 0; z < subdomain->tempDepth; z++)
            for (int y = 0; y < subdomain->tempHeight; y++)
                for (int x = 0; x < subdomain->tempWidth; x++) {

                    // Check for local minma/maxima count and update thier values.
                }

        // Store counts
        results->minimaCount[t] = minimaCount;
        results->maximaCount[t] = maximaCount;
    }
}
```

Parameters:

- `localData`: Local data array for the subdomain.
- `subdomain`: Defines the boundaries of the local subdomain.
- `results`: Stores the results, including the minimum and maximum values and counts of local minima and maxima.
- `timeSteps`: Number of time steps.

Logic: The function iterates over each time step, and for each spatial point within the actual subdomain (excluding ghost zones), it checks if the point is a local minimum or maximum. It updates the minimum and maximum values encountered and counts the local minima and maxima for each time step.

2.4. Result Aggregation and Output

2.4.1. Combining Results Across All Processes

After each process completes its local computations, we use `MPI_Reduce` to efficiently combine results:

```
// In main()
// Reduce minima counts from all processes
MPI_Reduce(localResults->minimaCount,
          rank == 0 ? globalResults->minimaCount : NULL,
          timeSteps, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

// Find global minimum values across all processes
MPI_Reduce(localResults->minValues,
          rank == 0 ? globalResults->minValues : NULL,
          timeSteps, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);

// Similar reductions for maxima counts and maximum values
```

We use:

- `MPI_SUM` to add up all the local minima/maxima counts.
- `MPI_MIN` and `MPI_MAX` to determine global minimum and maximum values.
- A separate `MPI_Reduce` to calculate timing information across all processes.

2.4.2. Writing Output to File

The `writeResults` function generates the output file in the required format:

Usage in Main Program:

```
// Only the root process writes output
if (rank == 0) {
    writeResults(outputFile, globalResults, timeSteps, &maxTiming);
}
```

The function `writeResults` writes the computed results to an output file in a human-readable format. It writes the local minima and maxima counts, the global minimum and maximum values, and the timing information for reading and processing the data.

Parameters:

- `outputFile`: The path to the output file where results will be saved.
- `globalResults`: Contains the aggregated results (minima counts, maxima counts, min and max values).
- `timeSteps`: The number of time steps for which results are computed.
- `timing`: Stores the timing information for different phases of the computation (e.g., reading, main computation, total).

Output Format:

- Line 1: Comma-separated pairs of local minima and maxima counts for each time step
- Line 2: Comma-separated pairs of global minimum and maximum values for each time step
- Line 3: Three timing measurements - Read Time, Main Code Time, and Total Time

3. Code Compilation and Execution Instructions

This section provides detailed instructions for compiling and executing our parallel code implementation. We cover both the compilation and execution of our best-performing method as well as instructions for generating comprehensive benchmark results.

3.1. Compilation and Execution for Best Method

3.1.1. Compilation Process

Our project includes multiple implementation variants optimized for different scenarios. To compile all implementations, navigate to the source directory and use the provided Makefile:

```
cd src/
make all
```

This command compiles all implementation variants and places the resulting executable files in the `src/bin/` directory.

3.1.2. Executing the Best Method

After compilation, you can run our best implementation using the provided job script:

```
cd jobs/
sbatch job7.sh
```

The `job7.sh` script contains optimized parameters and environment settings for running our code on the target HPC system. It executes the implementation with various input datasets and process configurations.

3.1.3. Accessing Results

After job completion, the results are organized as follows:

- A new directory `job7_results` is created in the `results/` folder
- Within this directory, a `raw/` subfolder contains all output files
- These output files follow the naming convention specified in the assignment: `output_NX_NY_NZ_NC_P.txt`

Each output file contains:

- Line 1: Pairs of local minima and maxima counts for each time step
- Line 2: Pairs of global minimum and maximum values for each time step
- Line 3: Time measurements (Read Time, Main Code Time, Total Time)

3.2. Compilation, Execution and Generation of Benchmark Results

3.2.1. Available Implementations

Our project includes several implementation variants to evaluate different optimization strategies:

3.2.2. Running Benchmark Tests

To comprehensively evaluate our implementation and generate benchmark data, we provide multiple job scripts (`job1.sh` through `job6.sh`). Each script tests different aspects of our implementation:

1. Execute each job script using sbatch:

```
cd jobs/
sbatch job1.sh
sbatch job2.sh
```

Implementation	Description
`send.c`	Basic implementation using blocking MPI_Send operations
`isend.c`	Implementation using non-blocking MPI_Isend operations
`bsend.c`	Implementation using buffered MPI_Bsend operations
`collectiveIO.c`	Implementation using collective MPI I/O operations
`collectiveIO_derData.c`	Enhanced collective I/O with file view
`independentIO.c`	Implementation using independent I/O operations
`independentIO_derData.c`	Independent I/O with file view
`independentIO_derData_and_isend.c`	Best-performing implementation combining independent I/O with file view and non-blocking communication

Table 1. Available implementation variants

```
# Continue for job3.sh through job6.sh

```

2. Each job creates a corresponding result directory (e.g., `job1_results`) in the `results/` folder
3. Within each result directory, a `benchmark_results.csv` file is generated containing performance metrics

3.2.3. Generating Visualizations

After collecting benchmark data, visualizations can be generated using the provided Python script:

```
cd scripts/
python3 ./visualize.py ../results/job1_results/benchmark_results.csv ../assets
    /job1_images
```

Similarly, visualizations for other benchmark results can be generated by specifying the appropriate input and output paths:

```
# For job2 through job6
python3 ./visualize.py ../results/job2_results/benchmark_results.csv ../assets
    /job2_images
python3 ./visualize.py ../results/job3_results/benchmark_results.csv ../assets
    /job3_images
# Continue for other jobs...
```

The generated visualizations in the `assets/` directory provide graphical representations of various performance metrics, including:

- I/O performance comparison between implementations
- Total time scaling with process count
- Total time scaling with datasize

These visualizations help identify performance bottlenecks and validate the effectiveness of our optimization strategies.

4. Code Optimizations

Our optimization strategy focused on identifying and addressing key performance bottlenecks in the parallel processing of 3D time-series data. Through systematic benchmarking and analysis, we identified three critical areas for optimization: domain decomposition, I/O operations, and inter-process communication. Our final implementation employs a hybrid approach that adaptively selects the optimal strategy based on process count and system characteristics.

4.1. Domain Decomposition with Ghost Zones

Bottleneck Addressed: Communication overhead when accessing neighbor data across processes required for checking if boundary points are local minima/maxima.

Implementation: We implemented a domain decomposition strategy that includes ghost zones (or halo regions) containing copies of neighboring processes' boundary data.

```c
typedef struct {
    // Core domain boundaries
    int startX, startY, startZ;
    int endX, endY, endZ;
    int width, height, depth;

    // Extended domain with ghost zones
    int tempStartX, tempStartY, tempStartZ;
    int tempEndX, tempEndY, tempEndZ;
    int tempWidth, tempHeight, tempDepth;
} SubDomain;
```

Performance Impact: This approach eliminates the need for communication during the computation phase, as each process has local access to all required data.

4.2. Parallel I/O Optimization

Bottleneck Addressed: File I/O operations (reading the data from binary file)

Implementation: One way is to read the whole file from one process and then distribute the appropriate data sections to other processes. However, we observed that this is very inefficient due to the excessive communication overhead involved in distribution. Even with little optimizations such as using `MPI_Isend()`, it did not improve significantly. Thus, we use parallel I/O.

We found that independent reads with file view provides the best performance among other options. The file view restricts visibility to only the designated portion of the file for each process, enabling efficient access to the non-contiguous/interleaved data required by the process. Otherwise, it leads to contention. To our surprise, we also obtained poorer performance from collective I/O with file view, as compared to independent I/O with file view. This might be because the data access pattern does not benefit much from the aggregation introduced by collective I/O. It's two-phase process and synchronization overhead negates the little gain it achieves over independent I/O with file view.

```c
MPI_Datatype filetype;
MPI_Type_create_subarray(4, globalSizes, subSizes, starts,
                         MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native", MPI_INFO_NULL);
```

Performance Impact: Using Independent I/O with file view significantly enhanced performance as compared to independent I/O.

4.3. Adaptive Hybrid I/O Strategy

Bottleneck Addressed: However, there still remains a problem. When we increase the number of processes above 24, we observed that simple `MPI_Isend()` outperforms independent I/O with file view. In fact, all implementations experience a dramatic performance cliff, but `MPI_Isend()` degrades more gracefully. This might be because Independent I/O with file view creates numerous simultaneous file requests as the process count increases, overwhelming the filesystem's ability to handle concurrent operations. In particular, we have to note that requests are for very small data chunks, which is not favorable. It might also be because the process count exceeds the stripe count of PARAM Rudra's file system implementation. On the other hand, `MPI_Isend()` bypasses the filesystem complications and leverages PARAM Rudra's InfiniBand HDR100 network more efficiently at high process counts.
Implementation: Thus, our final implementation uses a hybrid model, Independent I/O with file view for process count less than or equal to 24 and `MPI_Isend()` for process count more than 24.

```
if (size <= 24) {
    // Use Independent I/O with derived datatypes
    localData = readInputDataParallel_Level2(inputFile, &subdomain,
                                              nX, nY, nZ, timeSteps);
} else {
    // Use root-based distribution with non-blocking sends
    float* globalData = NULL;
    if (rank == 0) {
        globalData = readInputData(inputFile, totalDomainSize, timeSteps);
    }
    localData = distributeData(rank, &subdomain, globalData,
                               nX, nY, nZ, timeSteps, pX, pY, pZ);
}
```

Performance Analysis: Our benchmarking revealed that at process counts above 24, the filesystem's ability to handle concurrent I/O operations deteriorated significantly. We attribute this to:

- Small, non-contiguous data access patterns becoming increasingly inefficient
- Potential mismatch between process count and filesystem stripe count
- Network contention in the parallel filesystem

In these cases, leveraging the high-performance InfiniBand network for point-to-point communication proved more efficient than parallel file access.

4.4. Efficient Result Aggregation

Bottleneck Addressed: Communication overhead when gathering distributed results.
Implementation: We utilized MPI collective operations for efficient result aggregation:

```
// Reduce results
MPI_Reduce(localResults->minimaCount,
           rank == 0 ? globalResults->minimaCount : NULL,
           timeSteps, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Performance Impact: This approach minimizes communication overhead by using optimized tree-based reduction algorithms instead of point-to-point communications.

5. Results

5.1. Scalability Results

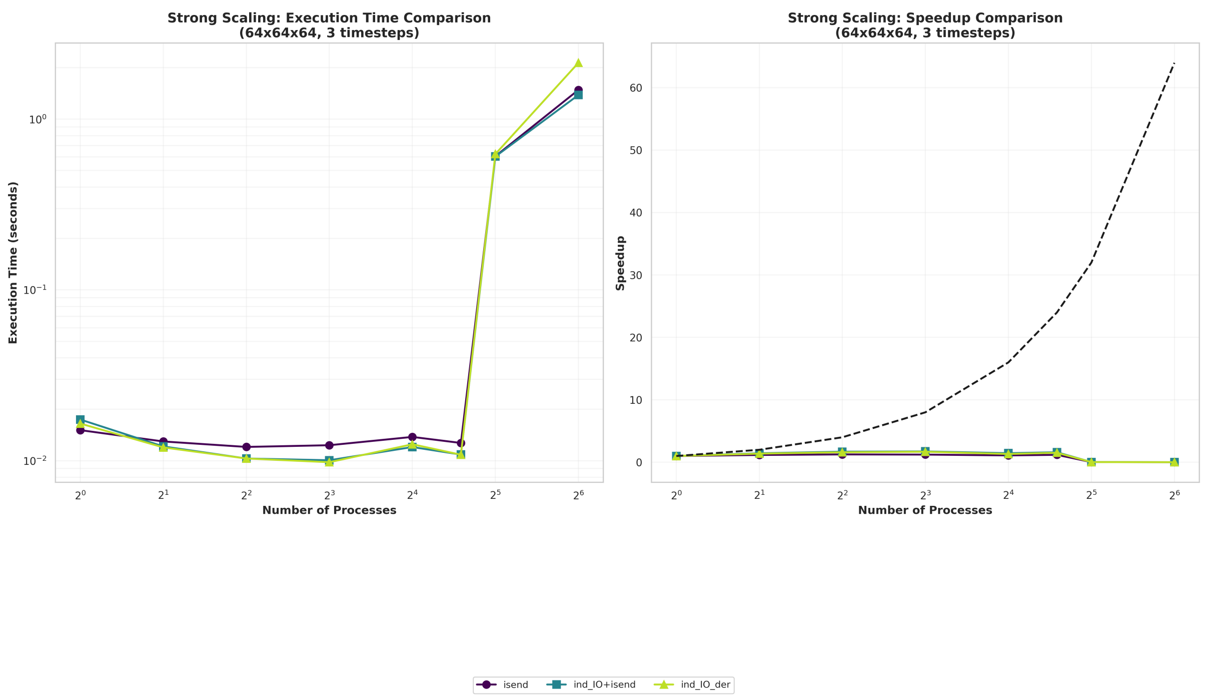

Figure 1. Data size $= 64 \times 64 \times 64$, Number of timesteps $= 3$

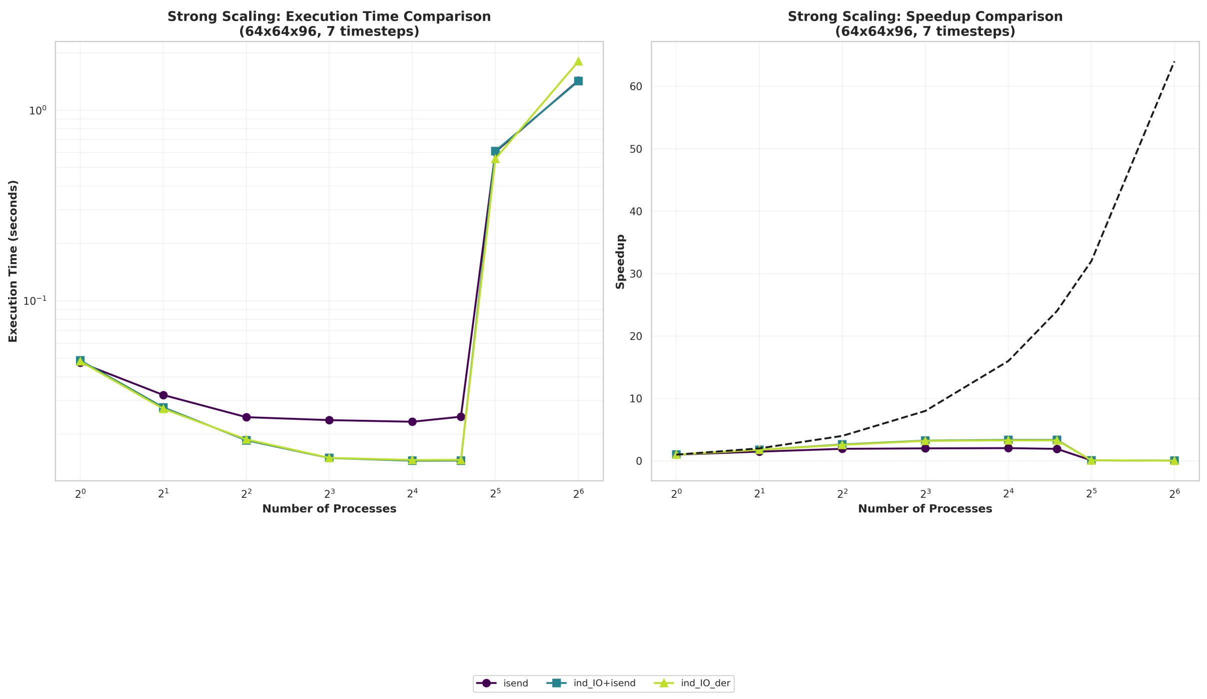

Figure 2. Data size $= 64 \times 64 \times 96$, Number of timesteps $= 7$

The graphs provided show scaling results for different strategies on PARAM Rudra for the two given datasets. The dashed line in the speedup graph represents ideal scaling (linear speedup), which none of the implementations achieve due to parallelization overheads. All implementations initially show

improved performance as the number of processes increases (1 $\rightarrow$ 16 processes). Beyond 24 processes, a performance cliff occurs for all methods, with execution times increasing dramatically. `ind_IO_der` (Independent I/O with file view) performs best with up to 24 processes. `MPI_Isend` degrades gracefully at higher process counts, outperforming `ind_IO_der` beyond 24 processes. This justifies our final hybrid model `ind_IO+isend`, as already described under code optimizations (2).
Another interesting thing to note is that performance optimum occurs at 8 processes for the smaller dataset, while it occurs at 16 processes for the larger dataset. This is because over-decomposition begins to occur as the number of processes crosses 8, for the smaller dataset. The peak speedup is also high for the larger dataset as compared to the smaller dataset.

5.2. Results of Various Implementations

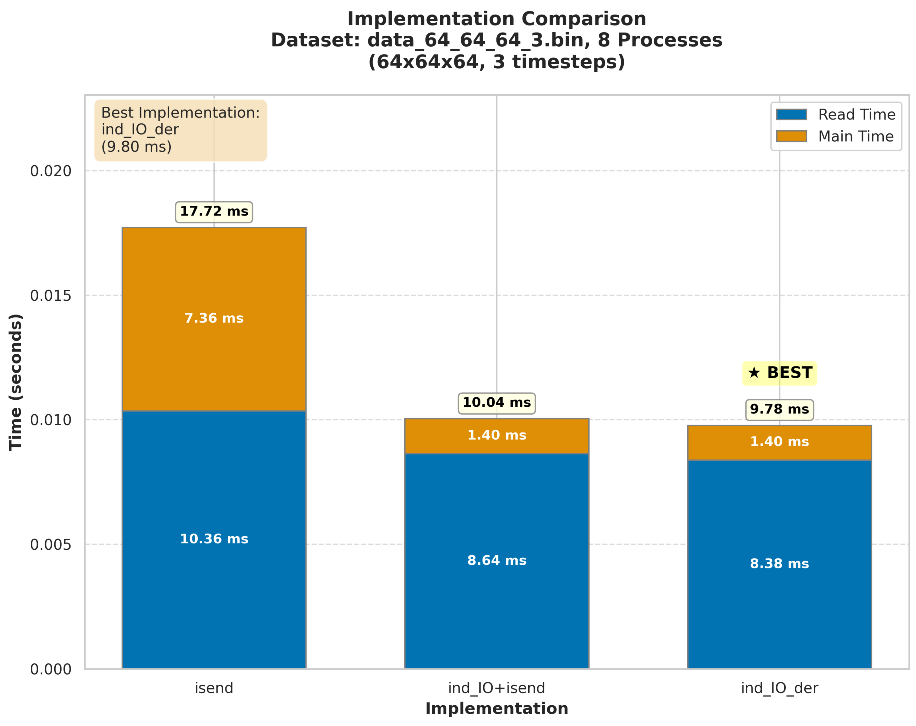

Figure 3. Data size = 64 × 64 × 64, Number of timesteps = 3, Number of processes = 8

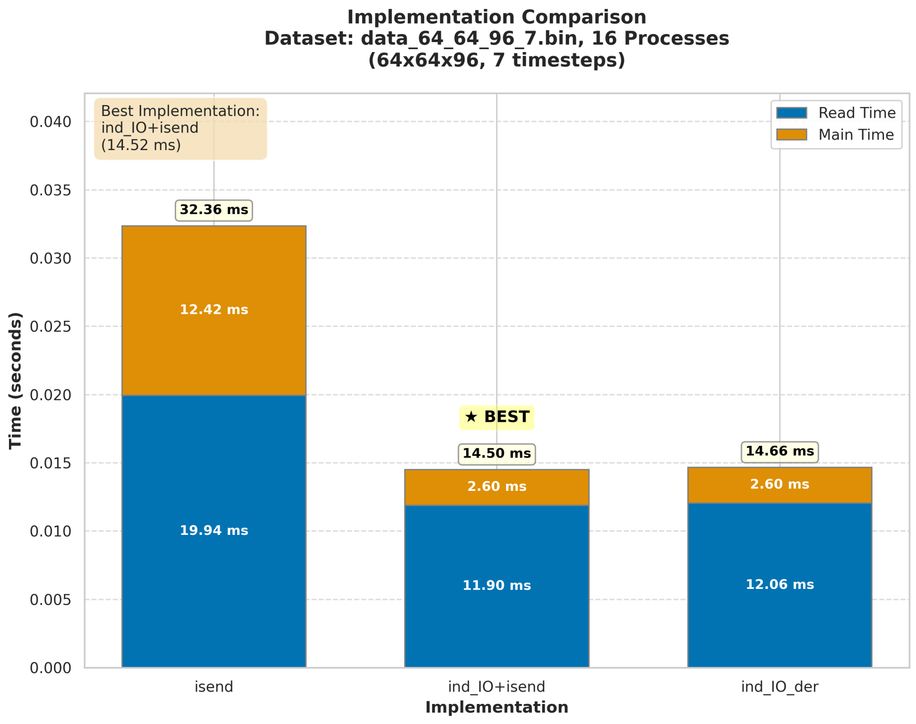

Figure 4. Data size = 64 × 64 × 96, Number of timesteps = 7, Number of processes = 16

These figures, once again, indicate the efficiency of the hybrid model. We also show below a comparison of all methods that we tried out, indicating how we finally arrived at the hybrid model. The labels indicate the type of implementation it refers to. `coll_I/O_der` and `coll_I/O_der2` are two implementations of collective I/O with file view but using different Info objects.

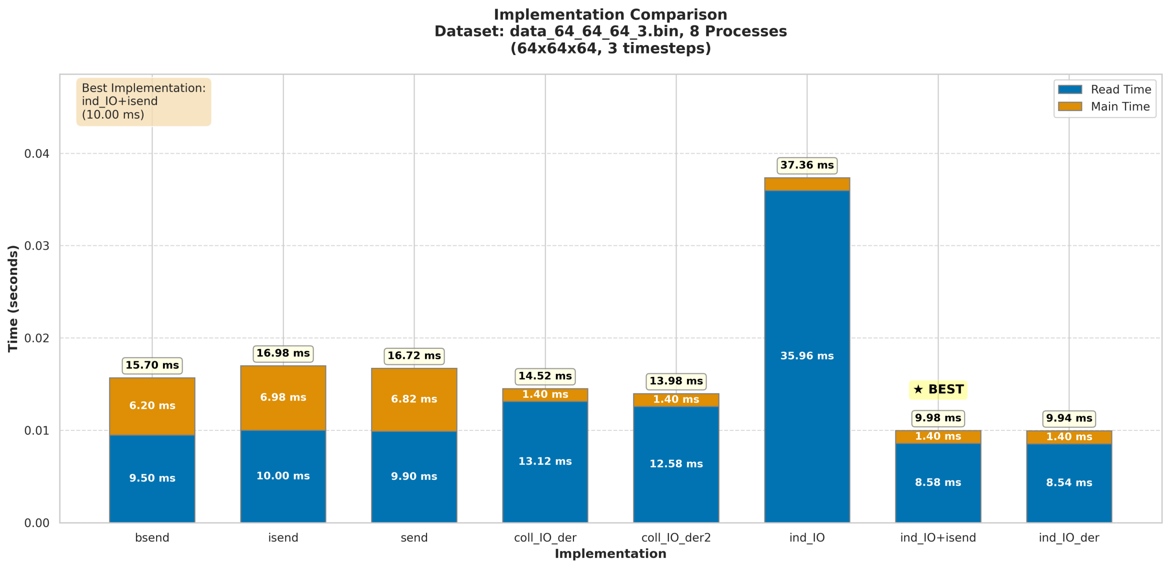

Figure 5. Data size = 64 × 64 × 64, Number of timesteps = 3, Number of processes = 8

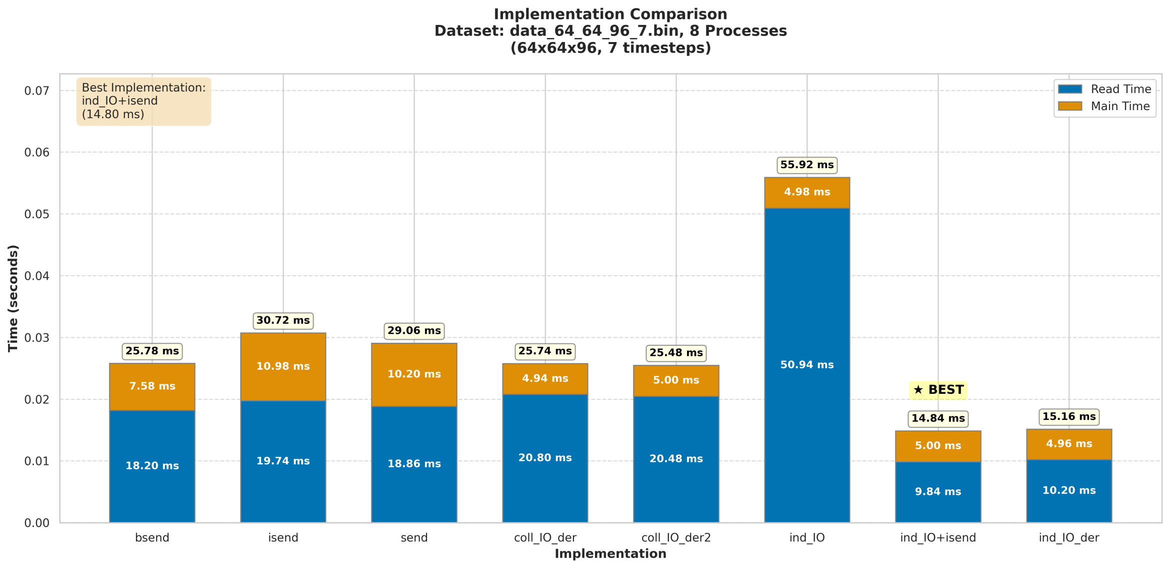

Figure 6. Data size = 64 × 64 × 96, Number of timesteps = 7, Number of processes = 8

For further understanding, we created two more larger datasets and tried our implementations on them. Below is a more extensive comparison. As described earlier, we can observe that in case of 16 processes, our hybrid implementation is same as independent I/O with file view and in in case of 64 processes, it is same as the `MPI_Isend` implementation. It remains the best (marked with a star) in each case.

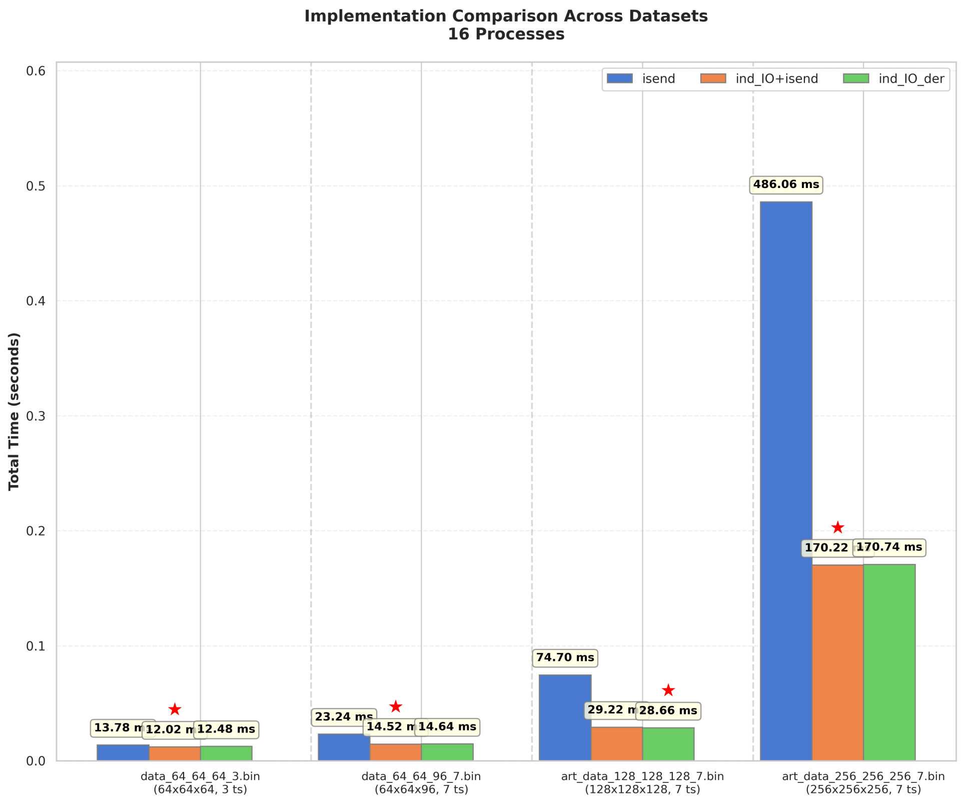

Figure 7. Number of processes = 16

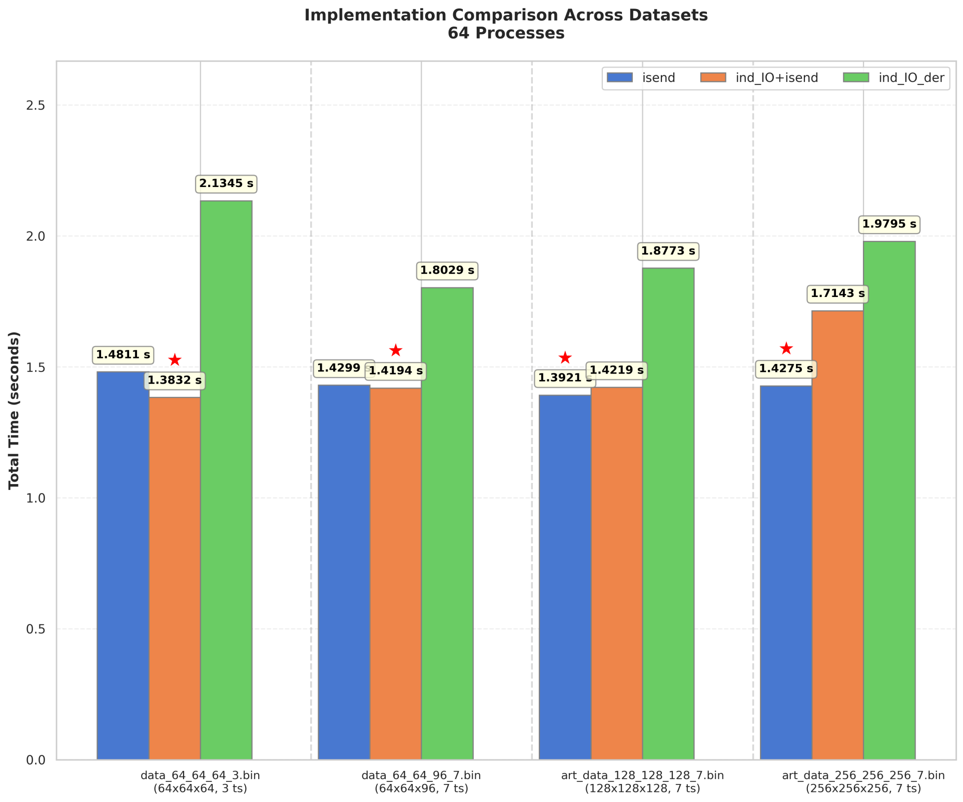

Figure 8. Number of processes = 64

5.3. Timings

Below is a table indicating the timings of two runs for each configuration. These timings show the maximum value among all processes.

Table 2. Parallel I/O Performance Comparison (Times in milliseconds)

Dataset	Implementation	Read Time	Main Time	Total Time
64×64×64×3	*8 Processes*			
	ind_IO+isend	0.0090	0.0014	0.0104
	ind_IO+isend	0.0082	0.0014	0.0096
	ind_IO_der	0.0082	0.0014	0.0096
	ind_IO_der	0.0084	0.0014	0.0098
	isend	0.0104	0.0076	0.0123
	isend	0.0108	0.0074	0.0125
	16 Processes			
	ind_IO+isend	0.0118	0.0008	0.0126
	ind_IO+isend	0.0091	0.0008	0.0099
	ind_IO_der	0.0125	0.0008	0.0133
	ind_IO_der	0.0116	0.0008	0.0124
	isend	0.0123	0.0090	0.0139
	isend	0.0128	0.0091	0.0142
	32 Processes			
	ind_IO+isend	0.6073	0.6225	0.6298
	ind_IO+isend	0.5705	0.5663	0.5720
	ind_IO_der	0.6379	0.0004	0.6383
	ind_IO_der	0.5848	0.0005	0.5852
	isend	0.6046	0.6023	0.6073
	isend	0.6074	0.6032	0.6103
	64 Processes			
	ind_IO+isend	1.3409	1.0613	1.3430
	ind_IO+isend	1.2866	1.2384	1.2869
	ind_IO_der	2.1584	0.0003	2.1587
	ind_IO_der	2.1598	0.0003	2.1600
	isend	1.6630	0.5549	1.6634
	isend	1.4237	0.9801	1.4240

Table 3. Parallel I/O Performance Comparison (Times in milliseconds)

Dataset	Implementation	Read Time	Main Time	Total Time
64×64×96×7	*8 Processes*			
	ind_IO+isend	0.0096	0.0050	0.0146
	ind_IO+isend	0.0098	0.0050	0.0148
	ind_IO_der	0.0107	0.0050	0.0157
	ind_IO_der	0.0098	0.0050	0.0148
	isend	0.0194	0.0113	0.0245
	isend	0.0181	0.0103	0.0239
	16 Processes			
	ind_IO+isend	0.0110	0.0026	0.0136
	ind_IO+isend	0.0117	0.0026	0.0143
	ind_IO_der	0.0121	0.0026	0.0147
	ind_IO_der	0.0120	0.0026	0.0146
	isend	0.0226	0.0143	0.0253
	isend	0.0199	0.0139	0.0247
	32 Processes			
	ind_IO+isend	0.5840	0.5757	0.5854
	ind_IO+isend	0.6318	0.6213	0.6333
	ind_IO_der	0.6513	0.0015	0.6527
	ind_IO_der	0.5927	0.0015	0.5942
	isend	0.5640	0.5739	0.5857
	isend	0.6387	0.6242	0.6403
	64 Processes			
	ind_IO+isend	1.4099	1.1089	1.4160
	ind_IO+isend	1.3239	1.1446	1.3248
	ind_IO_der	1.8413	0.0008	1.8420
	ind_IO_der	1.8392	0.0008	1.8398
	isend	1.3017	1.1696	1.3030
	isend	1.4955	1.1535	1.5055

Summing up, we can observe that all the results lead to the conclusions that we stated in the code optimizations section. We have also reasoned why such observations have been obtained.

6. Conclusions

S. No.	Name	Contribution
1	Pankaj Nath	Mainly worked on the code related to parallelization strategies. Implemented the core computation part of the code and contributed to some data distribution strategies.
2	Saagar K V	Worked on testing various implementations to determine the best-performing one under different conditions. Also wrote the Code Optimization section of the report.
3	Venkatesh	Designed customized datatype, strategies and functions used across all implementations, which were crucial for writing the code. Contributed to the Code Description section of the report.
4	Sai Nikhil	Focused on the implementation of data distribution strategies. Contributed several strategies for comparison and performance evaluation.
5	Rushikesh Chary	Worked on debugging, error handling, and analyzing critical sections of the code. Contributed to the Results section of the report.

Thank You