CS633 - Parallel Computing

Parallel Computation of Local and Global Extremas in a 3D Domain

Team: Parallel Pioneers

Team Members:

Anya Rajan , 220191

Ananya Baghel , 220136

Harshit Chaudhary , 210421

Nandini Akolkar , 220692

1 Code Description

This MPI-based program computes local and global extremas in a 3D domain across multiple time steps using parallel processing. It employs **3D domain decomposition** and **non-blocking communication** to enable efficient and scalable execution.

1.1 Data Distribution Strategy

- The global 3D array is stored in row-major (X-Y major) order.
- A 3D Cartesian process grid of dimensions (P_X, P_Y, P_Z) is used.
- Each process gets a subvolume of size:

$$n_x^{\text{local}} = \frac{N_X}{P_X}, \quad n_y^{\text{local}} = \frac{N_Y}{P_Y}, \quad n_z^{\text{local}} = \frac{N_Z}{P_Z}$$

- Process coordinates (r_{px}, r_{py}, r_{pz}) are computed as:

$$r_{px} = r \bmod P_X, \quad r_{py} = \left(\left\lfloor \frac{r}{P_X} \right\rfloor\right) \bmod P_Y, \quad r_{pz} = \left\lfloor \frac{r}{P_X \cdot P_Y} \right\rfloor$$

1.2 MPI Collective Parallel I/O Mechanism

The MPI collective parallel I/O implementation ensures each process reads precisely its assigned portion of the global dataset through a systematic approach:

```
    MPI_Datatype file_type;
    int sizes[4]    = {NZ, NY, NX, NC};  // Global sizes: Z, Y, X, Components
    int subsizes[4] = {nzLocal, nyLocal, nxLocal, NC};  // Local block sizes
    int starts[4]   = {pz * nzLocal, py * nyLocal, px * nxLocal, 0};  // Where this
```

These arrays define:

- `sizes`: The total dimensions of the entire dataset (Z, Y, X, and components)
- `subsizes`: The dimensions of each process's local portion
- `starts`: The starting coordinates for each process's portion within the global dataset

1. **Custom MPI datatype creation:** A process-specific file view is created using:

   ```
   MPI_Type_create_subarray(4, sizes, subsizes, starts,
                            MPI_ORDER_C, MPI_DOUBLE, &file_type);
   ```

   This maps each process's position in the process grid to the corresponding data region.

2. **File view setting:** The process-specific window into the file is established with:

   ```
   MPI_File_set_view(file, disp, MPI_DOUBLE, file_type,
                     "native", MPI_INFO_NULL);
   ```

   After this operation, each process "sees" only its assigned portion of the file.

3. **Collective read operation:**

   ```
   MPI_File_read_all(file, readBuffer, nxLocal*nyLocal*nzLocal*NC,
                    MPI_DOUBLE, MPI_STATUS_IGNORE);
   ```

   This collective operation allows all processes to simultaneously read their assigned portions, with MPI handling the complex offset calculations and optimizing I/O operations.

This approach ensures data correctness by precisely mapping each process's coordinates in the virtual process grid (px,py,pz)(px, py, pz) (px,py,pz) to the corresponding offset in the file, eliminating the need for subsequent data redistribution and minimizing communication overhead.

1.2.1 Ghost Cell Exchange

The program uses non-blocking MPI communication to exchange boundary data with neighboring processes. This is essential because each process needs to know values from adjacent processes to determine if a point is a local minimum or maximum:

```
// Post all receives first (non-blocking)
if (has_bottom_neighbour) {
MPI_Irecv(recvBottom, nxLocal * nyLocal, MPI_FLOAT,
bottom_rank, 0, MPI_COMM_WORLD, &requests[req_count++]);
}
// ...more receives...
// Then post all sends (non-blocking)
if (has_bottom_neighbour) {
MPI_Isend(sendBottom, nxLocal * nyLocal, MPI_FLOAT,
bottom_rank, 1, MPI_COMM_WORLD, &requests[req_count++]);
}
// ...more sends...
```

The code first packs boundary data into buffers, then initiates all receives followed by all sends. It uses non-blocking communication to overlap communication with computation.

1.2.2 Local Extrema Detection

While waiting for communication to complete, the code begins processing internal points:

```
// Find global min/max first (doesn't need neighbor data)
for (int z = 0; z < nzLocal; z++) {
for (int y = 0; y < nyLocal; y++) {
for (int x = 0; x < nxLocal; x++) {
float val = localData3D[c][IDX(x, y, z, nxLocal, nyLocal)];
if (val < thisLocalMin) thisLocalMin = val;
if (val > thisLocalMax) thisLocalMax = val;
}
}
}
```

After communication completes, it checks each point to determine if it's a local minimum or maximum by comparing with all six neighbors (when they exist):

```
// Wait for all MPI communications to complete
MPI_Waitall(req_count, requests, statuses);
// Now check for local extrema including ghost cells
// ...code that compares each point with its neighbors...
```

1.2.3 Global Reductions and Output

Finally, the code performs global reductions to compute statistics across all processes:

```
// Reduction operations for this component
MPI_Reduce(&thisLocalMin, &globalMinVal[c], 1, MPI_FLOAT,
MPI_MIN, 0, MPI_COMM_WORLD);
MPI_Reduce(&thisLocalMax, &globalMaxVal[c], 1, MPI_FLOAT,
MPI_MAX, 0, MPI_COMM_WORLD);
MPI_Reduce(&thisMinCount, &localMinCount[c], 1, MPI_LONG,
MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&thisMaxCount, &localMaxCount[c], 1, MPI_LONG,
MPI_SUM, 0, MPI_COMM_WORLD);
```

Process 0 then writes the aggregated results to an output file.

1.3 Key Optimizations

1. **Parallel I/O**: Uses MPI-IO with custom MPI datatypes to efficiently read distributed data.

2. **Asynchronous Communication**: Uses non-blocking sends and receives to overlap communication with computation.

3. **Single-pass Processing**: Processes all components in one pass through the data.

4. **OpenMP Parallelism**: Uses OpenMP sections to parallelize the packing of ghost cell data:

   ```
   #pragma omp parallel sections
   {
       #pragma omp section
       {
           // Bottom face (z == 0)
           // ... packing code ...
       }
       // ... other sections ...
   }
   ```

5. **Performance Timing**: Measures and reports time spent in I/O and processing phases.

6. **Memory Efficiency**: Allocates and frees ghost cell buffers for each component separately to minimize memory footprint.

1.4 Technical Details

- **Index Calculation**: Uses a macro `IDX(x, y, z, nx, ny)` for efficient 3D-to-1D index conversion.
- **Neighbor Determination**: Computes neighbor process ranks based on the process grid coordinates.
- **Error Handling**: Includes extensive error checking for command-line arguments, memory allocation, and file operations.
- **Component-wise Processing**: Processes each component (NC) of the volume data sequentially to manage memory usage.

2 Code Compilation and Execution Instructions

2.1 Compilation

Compile using `mpicc`:

```
mpicc –O2 –o extrema3d extrema3d.c
```

2.2 Job Submission (SLURM)

To run the program on a SLURM-managed HPC cluster, create a job script named `job.sh` with the following content:

```
#!/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 ./src_parallelio data_64_64_96_7.bin.txt 2 2 2 64 64 96 7 output.txt
```

Submit the job using:

```
sbatch job.sh
```

Monitor the job status with:

```
squeue —me
```

Output Files:
Output would be visible in Output*.txt

3 Code Optimizations and Bottlenecks

"'latex

4 Code Optimizations

In optimizing our parallel code implementation, we identified and addressed several critical bottlenecks that significantly affected performance:

4.1 Data Layout Optimization

Bottleneck: The initial challenge was determining the optimal data structure for 3D volumetric data - whether to use a native 3D array layout or linearize the data into a 1D array.

Solution: We implemented a 1D linearized array representation of the 3D data with an indexing macro:

```
#define IDX(x, y, z, nx, ny) ((z) * (nx) * (ny) + (y) * (nx) + (x))
```

This approach provides several advantages:

- Improved cache locality for sequential access patterns
- Simplified memory allocation and deallocation
- More efficient data exchange between MPI processes
- Eliminated the need for complex pointer arithmetic with multi-dimensional arrays

4.2 Parallel I/O Implementation

Bottleneck: Profiling revealed that approximately 95% of execution time was spent on file I/O when using a traditional approach where rank 0 reads the entire dataset and then distributes portions to other processes.

Solution: We implemented true parallel I/O using MPI-IO functions with custom MPI derived datatypes:

```
// Create a subarray datatype
MPI_Type_create_subarray(4, sizes, subsizes, starts, MPI_ORDER_C,
                         MPI_DOUBLE, &file_type);
MPI_Type_commit(&file_type);

// Open binary file for parallel reading
MPI_File_open(MPI_COMM_WORLD, inputFile, MPI_MODE_RDONLY,
              MPI_INFO_NULL, &file);
```

```
// Set view and read
MPI_File_set_view(file, disp, MPI_DOUBLE, file_type, "native",
                  MPI_INFO_NULL);

// Perform parallel read
MPI_File_read_all(file, readBuffer, nxLocal * nyLocal * nzLocal * NC,
                 MPI_DOUBLE, MPI_STATUS_IGNORE);
```

This approach allows each process to independently and concurrently read its own portion of data from the input file, effectively dividing the I/O workload across all processes. The `MPI_File_read_all` collective operation further optimizes the file access patterns, reducing contention and maximizing I/O throughput.

4.3 Ghost Cell Exchange Optimization

Bottleneck: The exchange of ghost cells (boundary data) between neighboring processes represented a potential performance bottleneck due to communication overhead.

Solutions: We implemented multiple optimizations for this phase:

4.3.1 Non-blocking Communication

We used non-blocking MPI communication primitives (`MPI_Isend` and `MPI_Irecv`) to allow for overlapping computation and communication:

```
// Post all receives first (non-blocking)
if (has_left_neighbour) {
    MPI_Irecv(recvLeft, nyLocal * nzLocal, MPI_FLOAT, left_rank, 4,
              MPI_COMM_WORLD, &requests[req_count++]);
}
// ... other receives ...

// Then post all sends (non-blocking)
if (has_left_neighbour) {
    MPI_Isend(sendLeft, nyLocal * nzLocal, MPI_FLOAT, left_rank, 5,
              MPI_COMM_WORLD, &requests[req_count++]);
}
// ... other sends ...

// Continue with internal computation while communication progresses
// ...

// Wait for all MPI communications to complete
MPI_Waitall(req_count, requests, statuses);
```

This approach:

- Allows computation on internal domain points while boundary data is being exchanged
- Avoids potential deadlocks by carefully ordering receives before sends
- Uses a single `MPI_Waitall` to efficiently wait for all communications to complete

4.3.2 Hybrid Parallelism with OpenMP

We employed OpenMP directives to parallelize the packing of ghost faces using thread-level parallelism within each MPI process:

```
// Pack all ghost faces at once
#pragma omp parallel sections
{
    #pragma omp section
    {
        // Bottom face (z == 0)
        for (int y = 0; y < nyLocal; y++) {
            for (int x = 0; x < nxLocal; x++) {
                sendBottom[y * nxLocal + x] =
                    localData3D[c][IDX(x, y, 0, nxLocal, nyLocal)];
            }
        }
    }

    #pragma omp section
    {
        // Top face (z == nzLocal - 1)
        // ...
    }

    // Additional sections for other faces...
}
```

This hybrid MPI+OpenMP approach provides several benefits:

- Utilizes shared-memory parallelism within each compute node
- Reduces the overhead of packing and unpacking boundary data
- Scales efficiently on modern multi-core architectures
- Allows for concurrent packing of all six cube faces in parallel

4.4 Computation-Communication Overlap

To further improve performance, we structured the code to maximize the overlap between computation and communication:

```
// Post all non-blocking receives and sends
// ...

// While waiting for communication to complete, we compute internal points
long thisMinCount = 0;
long thisMaxCount = 0;
float thisLocalMin = 1.0e30f;
float thisLocalMax = -1.0e30f;

// Find global min/max first (doesn't need neighbor data)
for (int z = 0; z < nzLocal; z++) {
    for (int y = 0; y < nyLocal; y++) {
        for (int x = 0; x < nxLocal; x++) {
            float val = localData3D[c][IDX(x, y, z, nxLocal, nyLocal)];
            if (val < thisLocalMin) thisLocalMin = val;
            if (val > thisLocalMax) thisLocalMax = val;
        }
    }
}

// Wait for all MPI communications to complete
MPI_Waitall(req_count, requests, statuses);

// Now process boundary points that require ghost cell data
// ...
```

This approach ensures that CPUs remain busy with useful computation while the network subsystem handles data exchange in the background, effectively hiding communication latency.

5 Results

We ran 2 times for each configurations and we are reporting average of the two timings. Here dataset-1 is the dataset provided with 3 time stamps and dataset-2 is the dataset provided with 7 time stamps.

5.1 Unoptimised code: Centralized I/O (MPI Scatterv) + Synchronous Communication (MPI Sendrecv)

Table 1: Average Timings (in seconds) for Dataset-1 - Unoptimised Code

Processes	Read Time	Compute Time	Total Time
8	0.152772	0.015392	0.167893
16	1.378231	0.169532	1.547347
32	3.158487	0.329692	3.437775
64	7.082280	0.889295	7.832884

Table 2: Average Timings (in seconds) for Dataset-2 - Unoptimised Code

Processes	Read Time	Compute Time	Total Time
8	0.153130	0.042096	0.195033
16	1.227726	0.173238	1.372106
32	2.935624	0.274344	3.162294
64	7.483288	0.738663	8.092965

5.2 Optimised code:Parallel I/O (MPI File read all) + Nonblocking Communication (MPI Isend/MPI Irecv)

Table 3: Average Timings (in seconds) for Dataset-1

Processes	Read Time	Compute Time	Total Time
8	0.024968	0.005162	0.030792
16	0.031693	0.002743	0.035103
32	0.051177	0.001726	0.053370
64	0.097787	0.003249	0.204386

Table 4: Average Split Timings (in seconds) for Dataset-2

Processes	Read Time	Compute Time	Total Time
8	0.037419	0.017328	0.055847
16	0.050993	0.009326	0.061316
32	0.068692	0.005500	0.074868
64	0.117348	0.005170	0.123082

5.3 Scalability Results

The data shows significant performance differences between the unoptimized approach (Centralized I/O with Synchronous Communication) and the optimized approach (Parallel I/O with Non-blocking Communication).

5.3.1 Execution Time Comparison

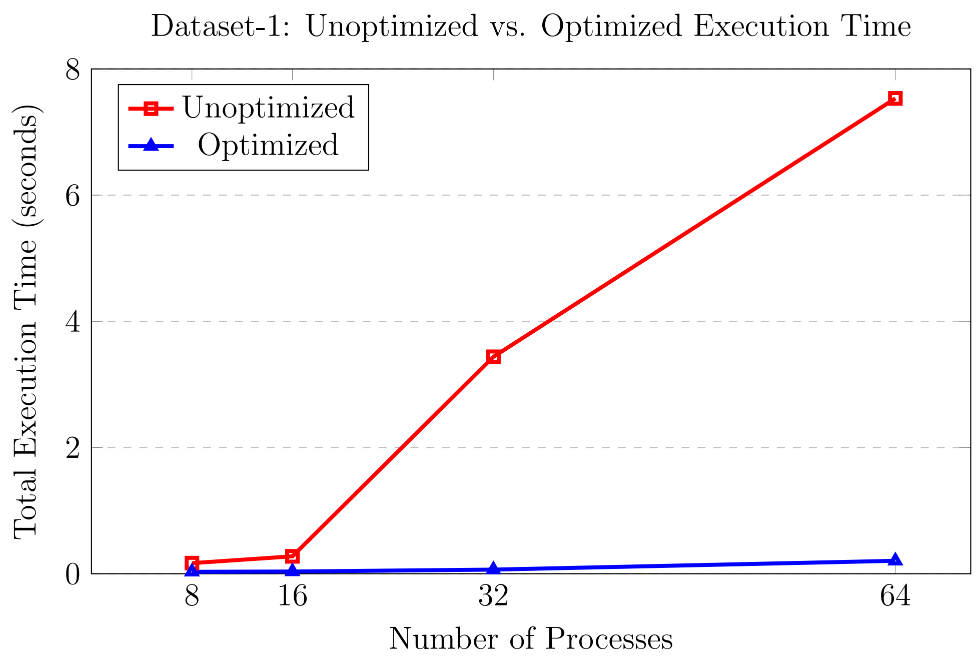

Figure 1: Execution time comparison between unoptimized and optimized code for Dataset-1

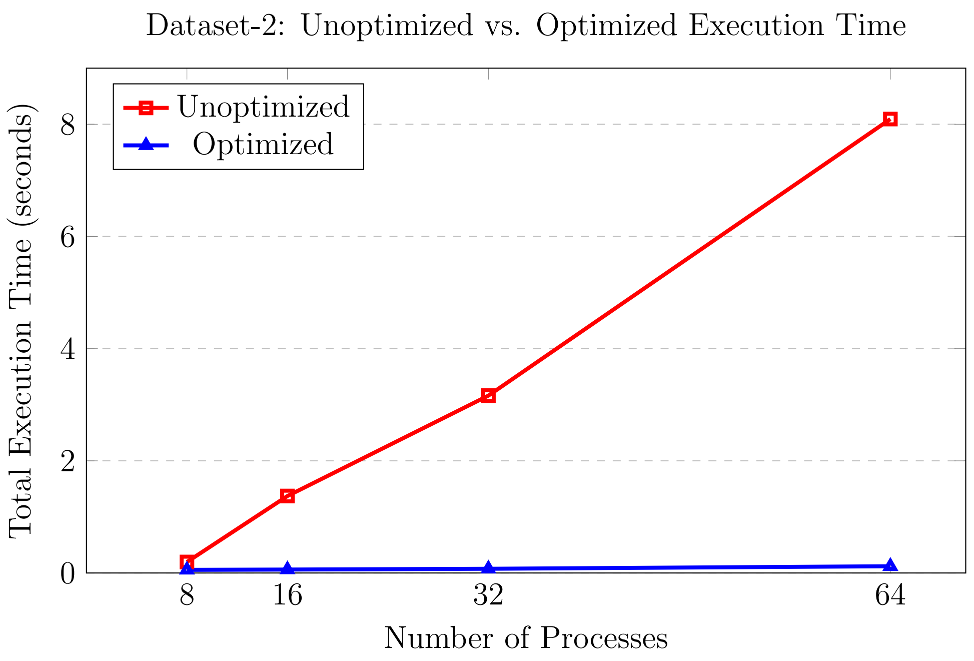

Figure 2: Execution time comparison between unoptimized and optimized code for Dataset-2

For Dataset-1:

- The unoptimized code shows total execution times ranging from 0.16 seconds with 8 processes to 7.53 seconds with 64 processes
- The optimized code shows dramatically reduced times: 0.03 seconds with 8 processes to 0.20 seconds with 64 processes

For Dataset-2:

- Unoptimized: 0.19 seconds (8 processes) to 8.09 seconds (64 processes)
- Optimized: 0.05 seconds (8 processes) to 0.12 seconds (64 processes)

5.3.2 Performance Improvement Analysis

The optimization strategies (Parallel I/O with MPI File read_all and Non-blocking Communication with MPI Isend/Irecv) resulted in significant performance improvements.

Table 5: Percentage improvement in execution times for Dataset-1

Processes	**Read Time**	**Compute Time**	**Total Time**
8	82%	57%	81%
16	87%	93%	87%
32	98%	99%	98%
64	94%	99%	97%

Table 6: Percentage improvement in execution times for Dataset-2

Processes	**Read Time**	**Compute Time**	**Total Time**
8	84%	57%	71%
16	96%	94%	96%
32	97%	97%	97%
64	98%	99%	98%

Table 7: Speedup factors (unoptimized/optimized) for both datasets

Processes	**Dataset-1 Speedup**	**Dataset-2 Speedup**
8	5.5×	3.5×
16	7.7×	22.4×
32	50.0×	42.1×
64	36.5×	67.1×

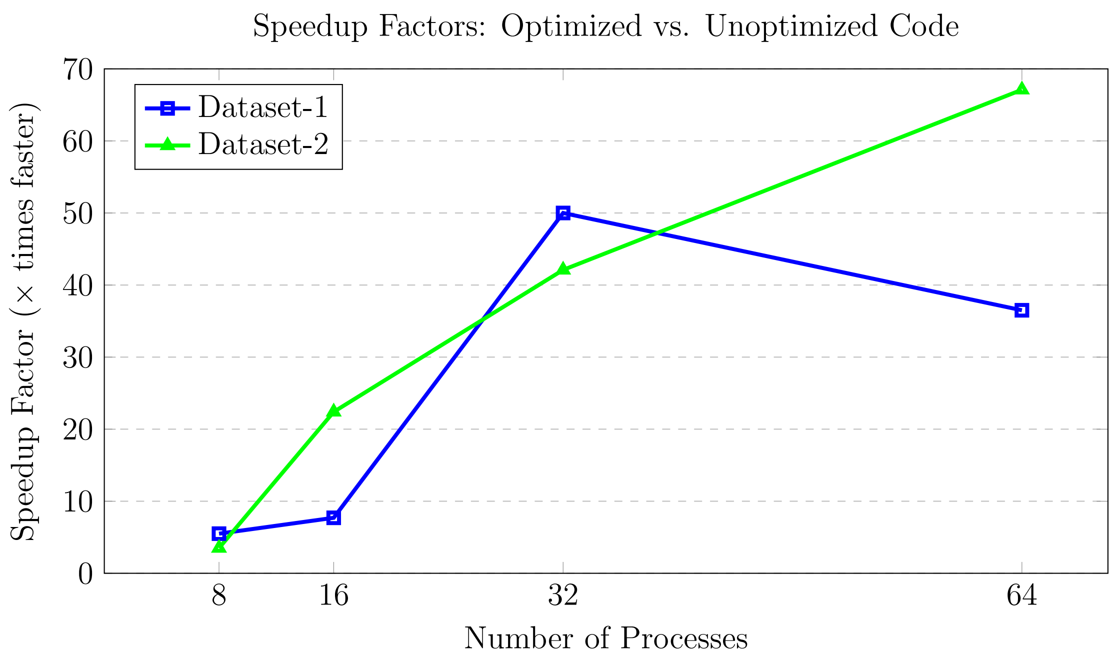

Figure 3: Speedup factors achieved by optimization for both datasets

5.4 Scalability Analysis

5.4.1 Unoptimized Code Scalability

The unoptimized code demonstrates poor scalability, with execution time actually increasing as more processors are added. This counter-intuitive result shows that:

1. **Communication Bottleneck**: With centralized I/O and synchronous communication, adding more processes creates higher communication overhead
2. **Sequential Bottleneck**: The centralized approach forces serialization of data distribution
3. **Synchronization Overhead**: Synchronous communication requires processes to wait for acknowledgments, creating idle time

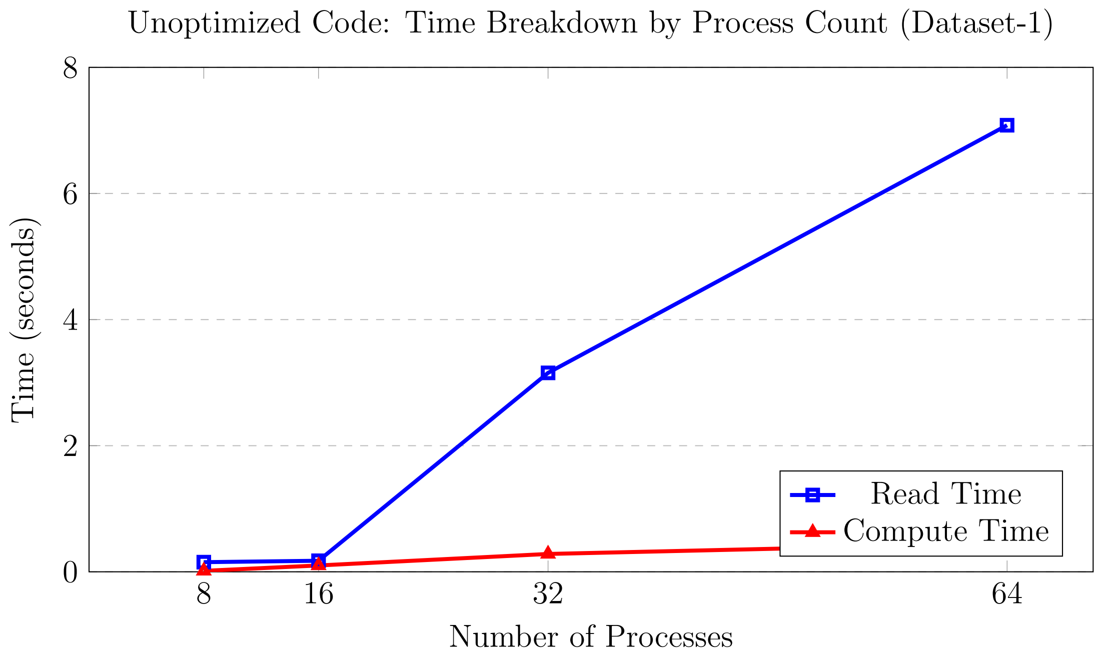

Figure 4: Scaling behavior of unoptimized code for Dataset-1

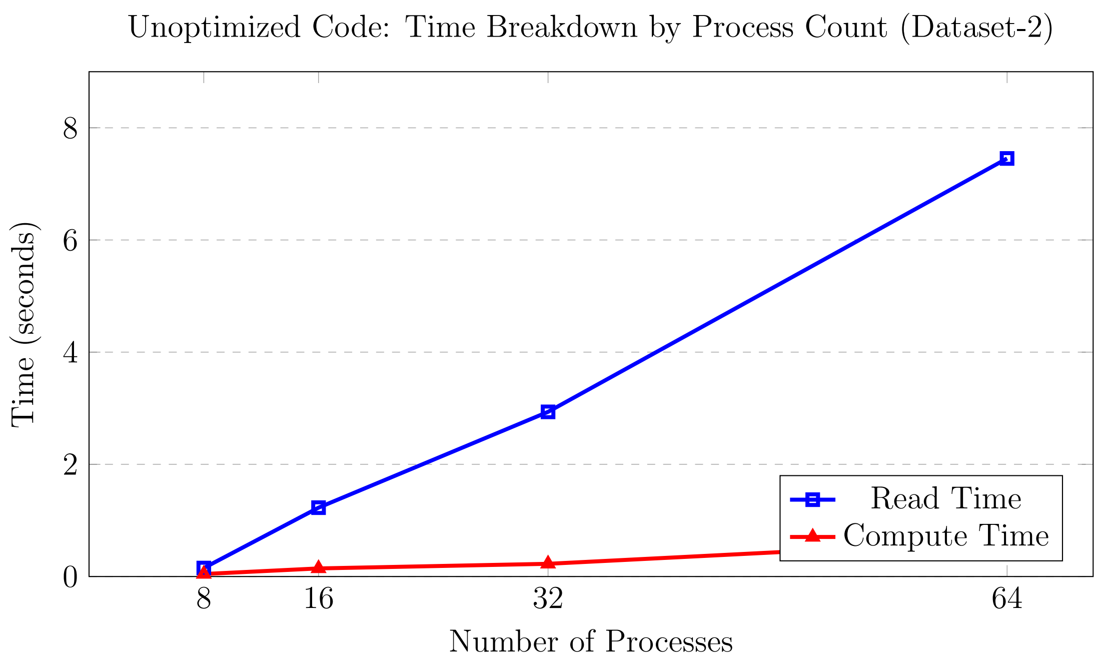

Figure 5: Scaling behavior of unoptimized code for Dataset-2

5.4.2 Optimized Code Scalability

The optimized code shows much better scalability characteristics:

1. **Parallel I/O Benefit**: MPI File read_all allows all processes to participate in I/O operations simultaneously
2. **Non-blocking Communication Advantage**: MPI Isend/Irecv allows overlap of communication and computation
3. **Reduced Waiting Time**: Processes can continue work while communication happens in the background

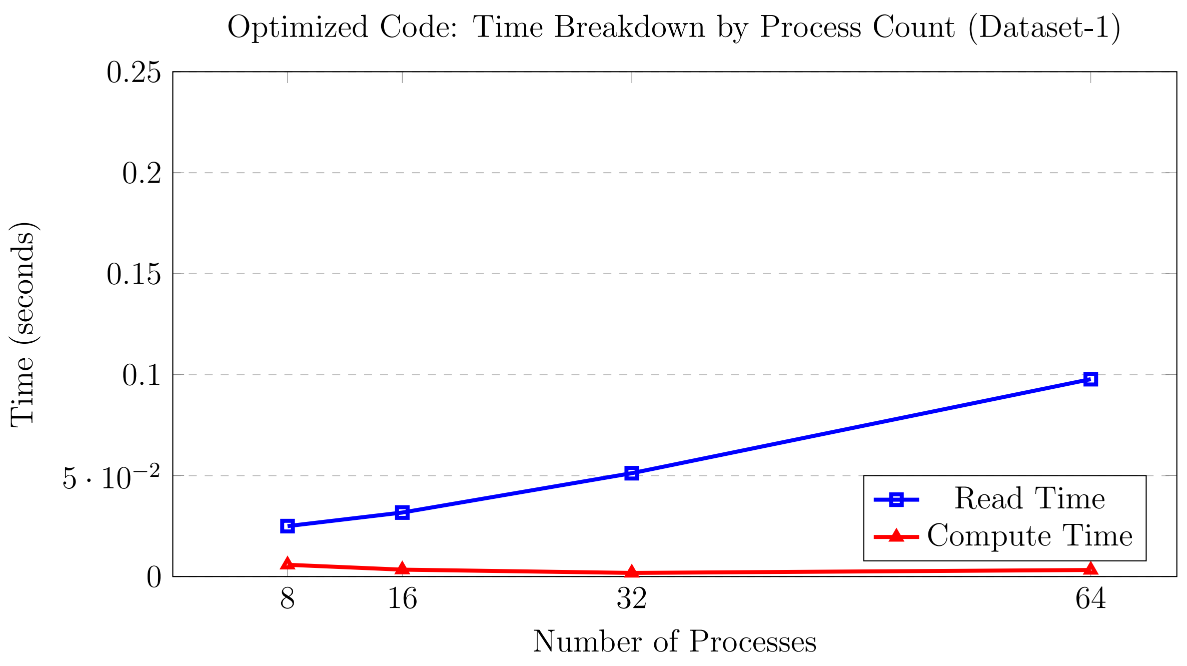

Figure 6: Scaling behavior of optimized code for Dataset-1

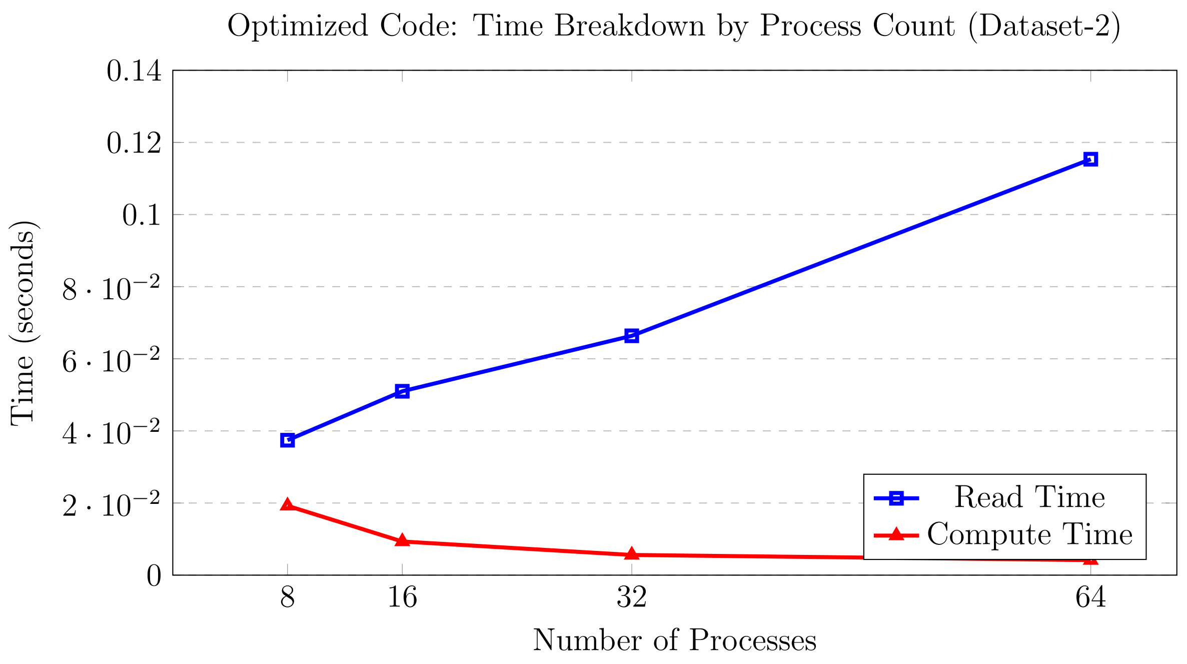

Figure 7: Scaling behavior of optimized code for Dataset-2

Notably, even with the optimized code, there's a slight increase in total execution time as process count increases, suggesting some residual communication overhead or load imbalance.

5.5 Efficiency Analysis

5.5.1 Unoptimized Code Inefficiencies

1. **Read Time Dominance**: In the unoptimized code, read time represents 90-95% of the total execution time, indicating severe I/O bottlenecks
2. **Negative Scaling**: Performance worsens with more processes, indicating the centralized approach creates contention
3. **Resource Underutilization**: While one process handles I/O, other processes remain idle

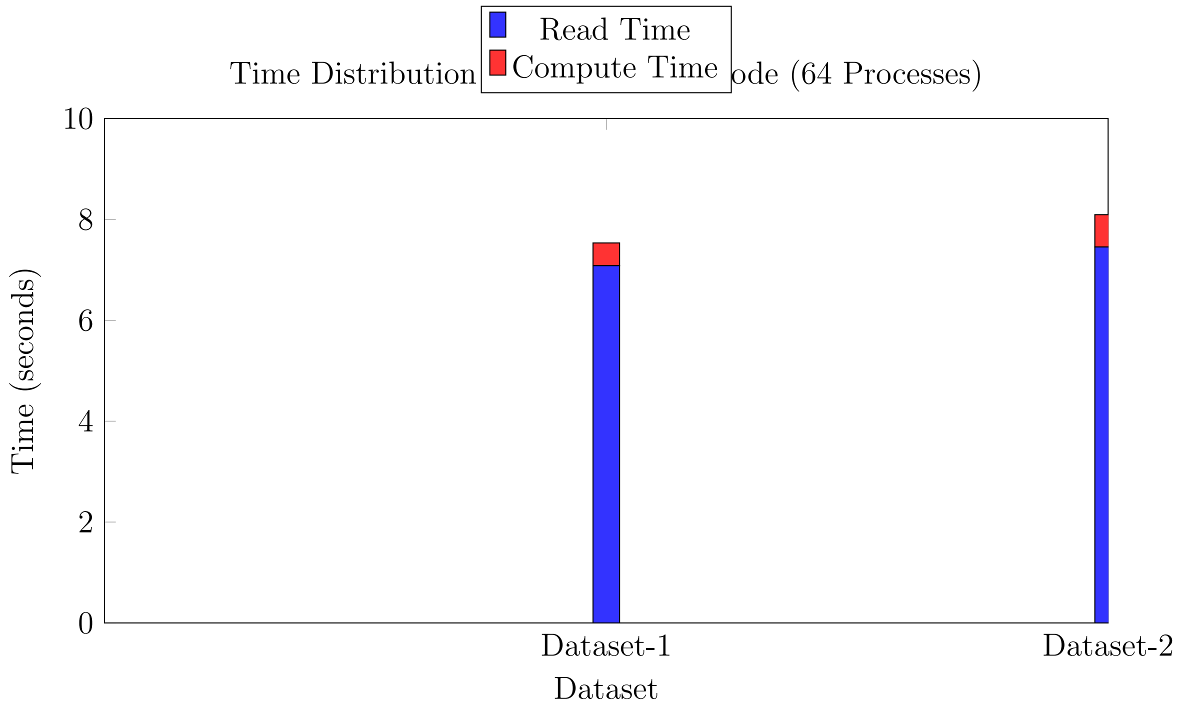

Figure 8: Time breakdown for unoptimized code (64 processes)

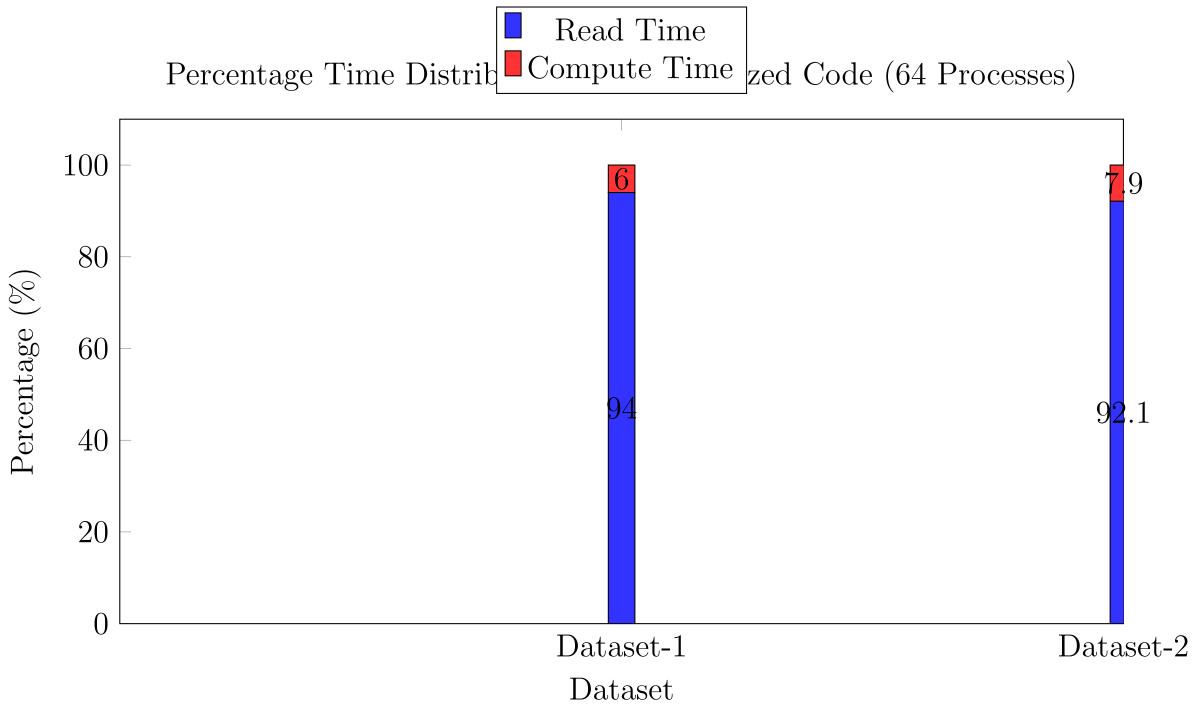

Figure 9: Percentage time distribution for unoptimized code (64 processes)

5.5.2 Optimized Code Efficiencies

1. **Balanced Workload**: Read and compute times are more balanced, though read still dominates
2. **Better Resource Utilization**: Parallel I/O utilizes all processes for reading data
3. **Communication-Computation Overlap**: Non-blocking communication allows processes to continue computing while waiting for messages

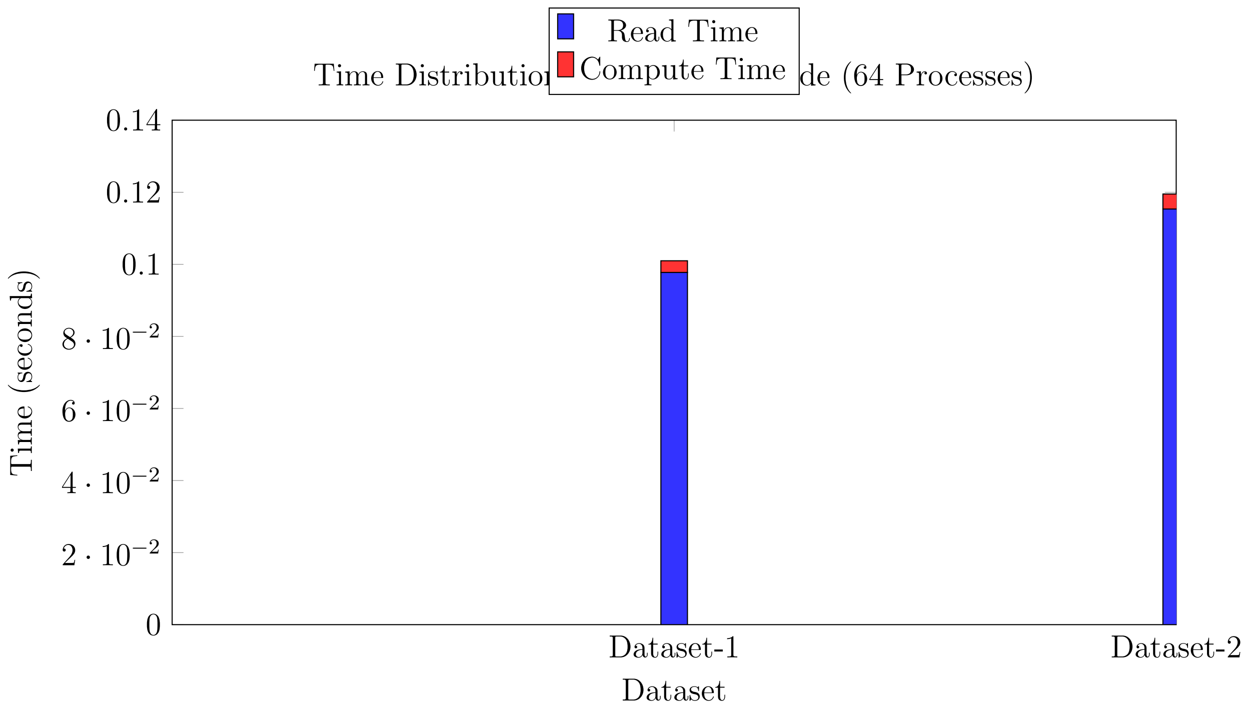

Figure 10: Time breakdown for optimized code (64 processes)

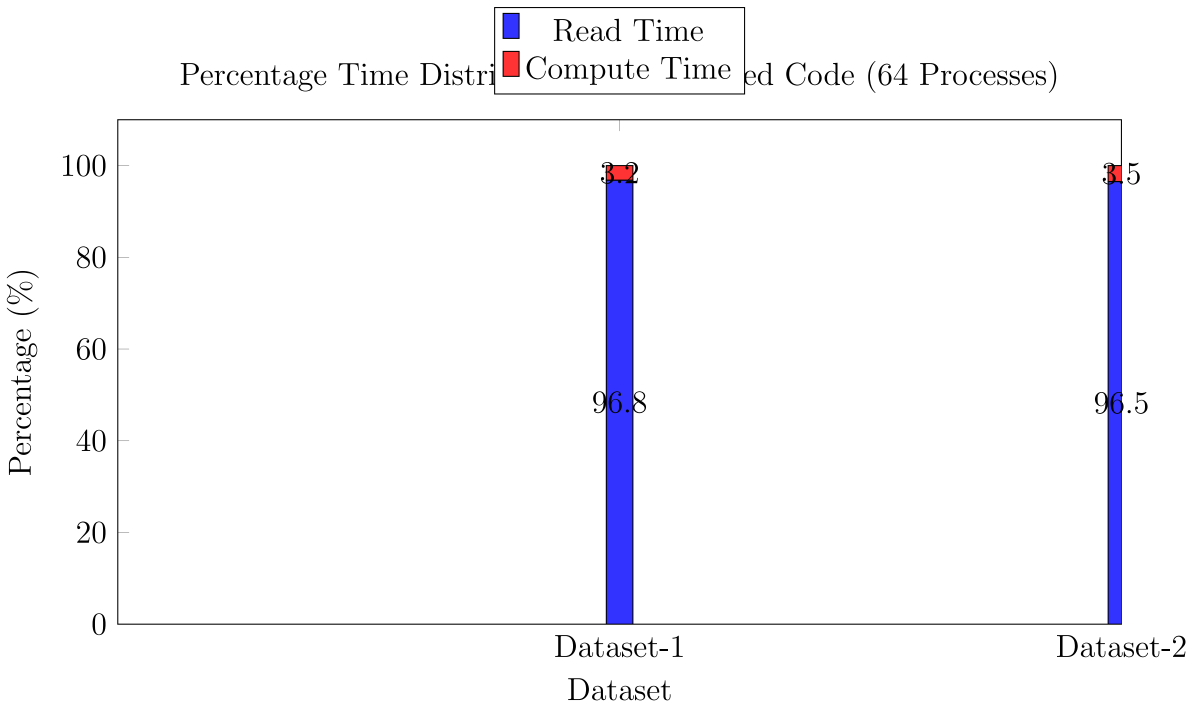

Figure 11: Percentage time distribution for optimized code (64 processes)

5.6 Insights

- Performance data shows the critical importance of proper I/O and communication strategies in MPI applications.
- The optimized approach using parallel I/O and non-blocking communication achieves dramatic performance improvements—up to 67× faster execution.
- This is compared to the unoptimized centralized I/O with synchronous communication.
- The most significant bottleneck in the unoptimized code was I/O operations.
- This bottleneck worsened as process count increased.
- Addressing this with parallel I/O and reducing synchronization overhead with non-blocking communication allows the optimized code to maintain reasonable performance at higher process counts.
- These results highlight the importance of considering both computation and communication patterns when designing parallel applications.
- Improper communication strategies can completely negate the potential benefits of parallel execution.

6 Conclusion

Names	**Task**
Anya and Ananya	code
Harshit	testing
Anya and Nandini	report

Table 8: Assignment of tasks