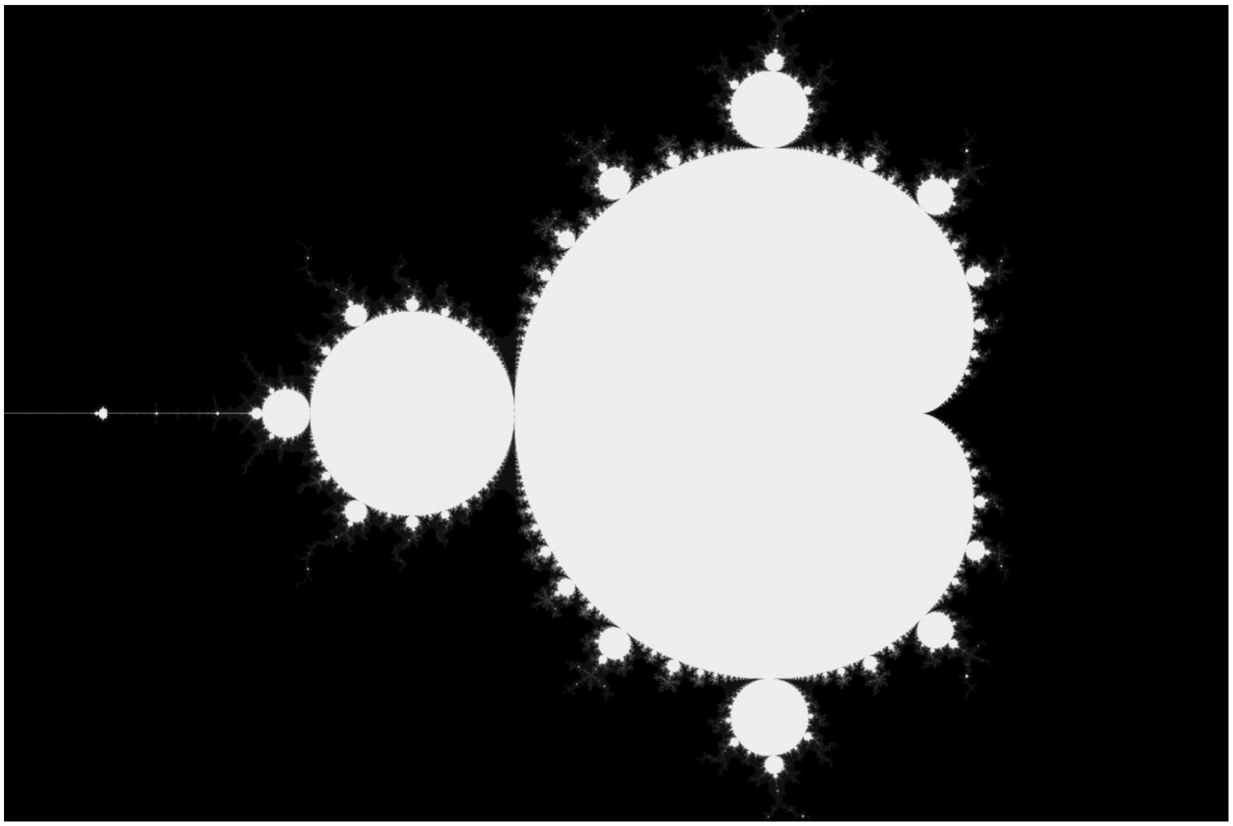
1. Introduction

The Mandelbrot set is a well-known fractal, defined by the set of complex numbers c for which the iterative function $z_{n+1} = z\frac{2}{n} + c_1$, starting with $z_0 = 0$, remains bounded as n approaches infinity. This set, discovered by Benoît Mandelbrot, has become a central object of study in the field of complex dynamics. To render the Mandelbrot set visually, a rectangular region of the complex plane is mapped onto a grid of pixels, where each pixel corresponds to a specific complex number $c = x + iy$.

Parallelizing the computation of the Mandelbrot set introduces the challenge of load imbalance. This issue arises because the number of iterations required varies significantly across different regions of the complex plane.



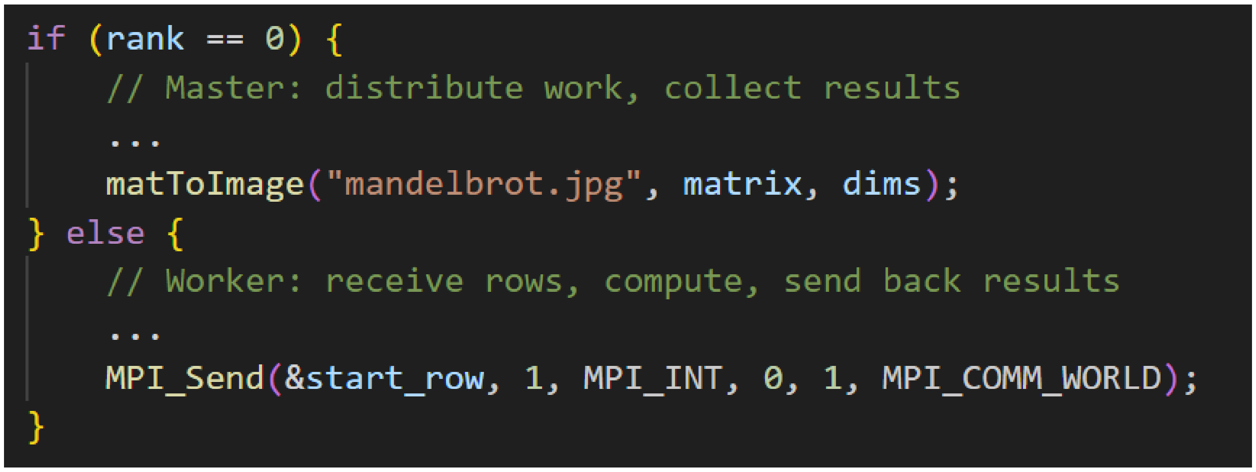
Mandelbrot Image: 3000 x 2000

Pixels near the boundary of the Mandelbrot set often demand more computation compared to those far from the boundary. Consequently, when tasks are distributed evenly among processors, some processors may complete their work much earlier than others, leading to inefficient resource management. To mitigate this issue, a dynamic workload distribution strategy was employed. This approach aims to distribute the computational load more evenly across all processors, thereby improving the overall efficiency and performance of the parallelized computation.

2. Implementation Details

The Mandelbrot set calculation was parallelized using MPI (Message Passing Interface) for process-level parallelism and OpenMP for thread-level parallelism. The program follows a Master-Worker model where the master process distributes tasks (rows of pixels) to worker processes, which calculate the Mandelbrot set for their assigned rows. Once all rows are processed, the master collects the results and assembles the final image.

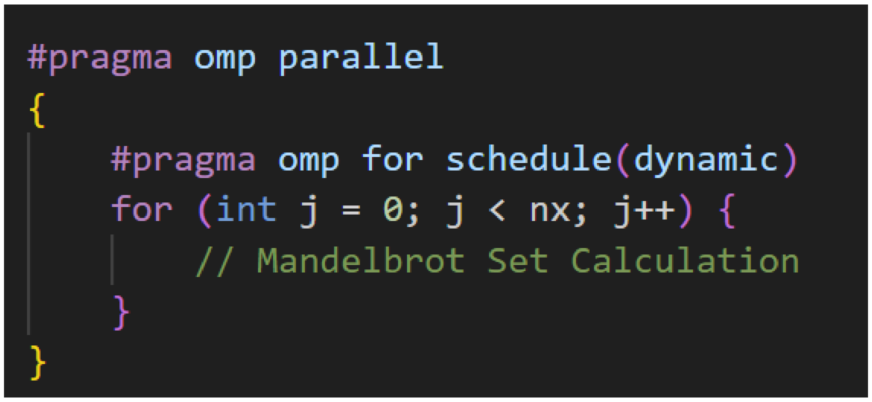
The master process (rank 0) is responsible for distributing rows of the image to be computed by the workers. Each worker receives a row index, computes the Mandelbrot set for that row, and sends the results back to the master. Once all rows are processed, the master collects the results and constructs the final image.



```
if (rank == 0) {
    // Master: distribute work, collect results
    ...
    matToImage("mandelbrot.jpg", matrix, dims);
} else {
    // Worker: receive rows, compute, send back results
    ...
    MPI_Send(&start_row, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
```

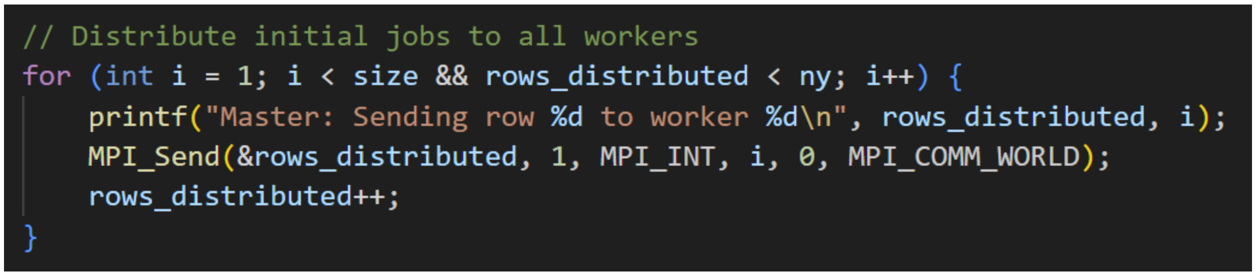
Within each MPI process, OpenMP is used to parallelize the computation of each row's pixels. The loop that computes the Mandelbrot set for each pixel is parallelized with

#pragma omp parallel for, enabling multi-threaded execution. OpenMP dynamically schedules iterations to balance the workload between threads, as pixels near the boundary of the Mandelbrot set require more iterations.



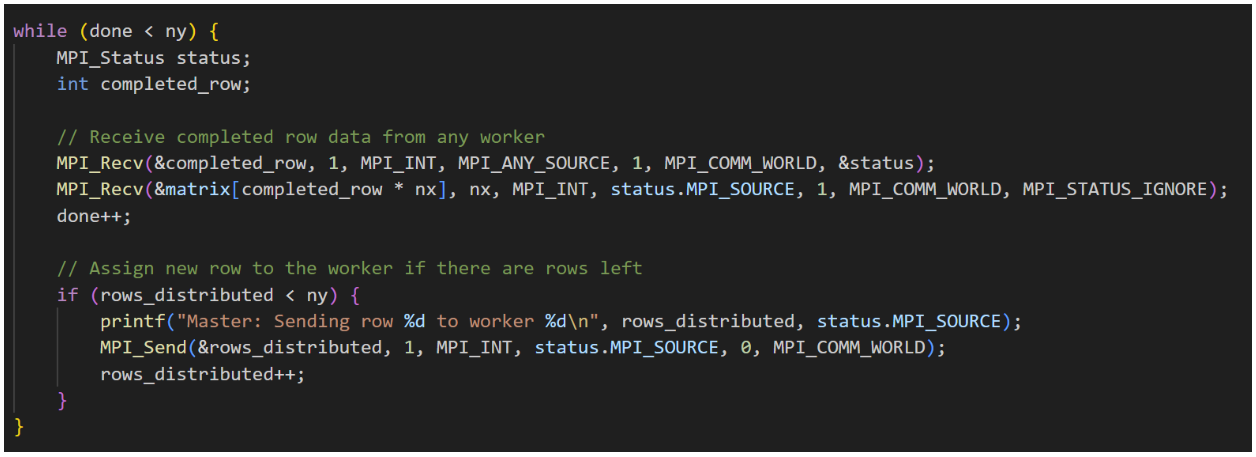
```
#pragma omp parallel
{
    #pragma omp for schedule(dynamic)
    for (int j = 0; j < nx; j++) {
        // Mandelbrot Set Calculation
    }
}
```

The master starts by sending each worker a row to compute. This is done using MPI_Send(). Each worker receives a row index, which corresponds to the row of pixels in the Mandelbrot image that the worker must calculate.



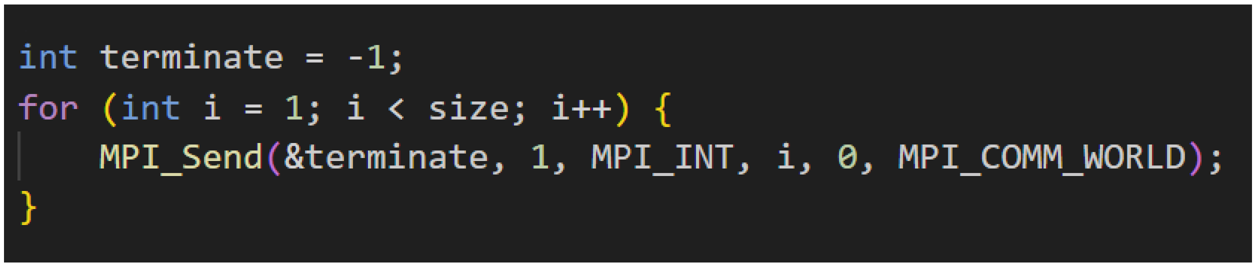
```
// Distribute initial jobs to all workers
for (int i = 1; i < size && rows_distributed < ny; i++) {
    printf("Master: Sending row %d to worker %d\n", rows_distributed, i);
    MPI_Send(&rows_distributed, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
    rows_distributed++;
}
```

Once a worker completes a row, it sends the row's data back to the master. The master receives this data using MPI_Recv() and then assigns a new row to the worker if more rows remain to be processed. This dynamic reassignment ensures that no worker remains idle while other workers are still busy.



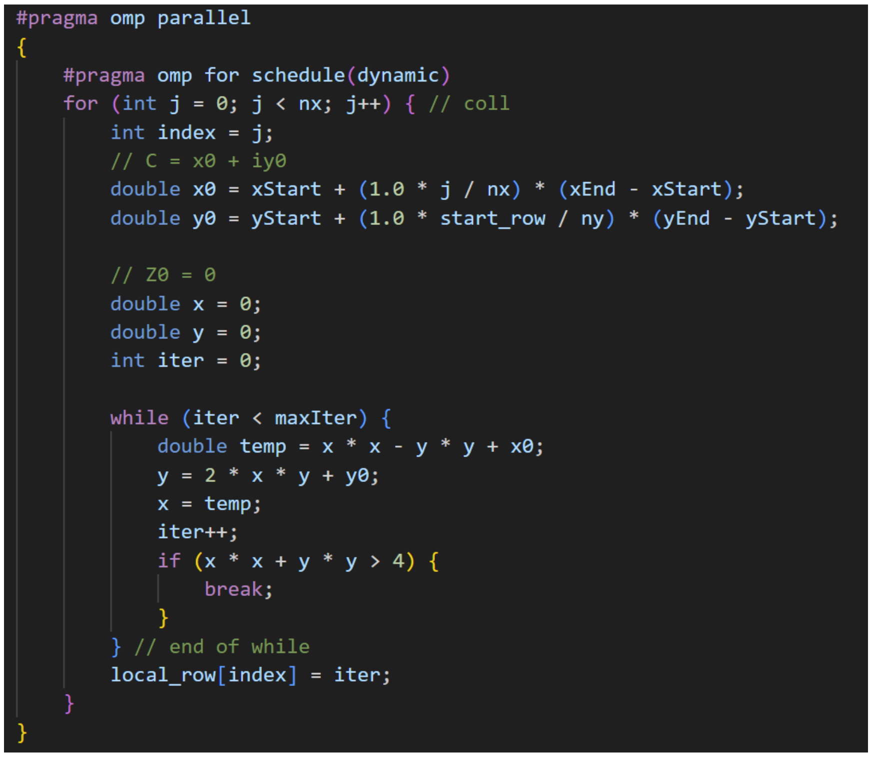
```
while (done < ny) {
    MPI_Status status;
    int completed_row;

    // Receive completed row data from any worker
    MPI_Recv(&completed_row, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &status);
    MPI_Recv(&matrix[completed_row * nx], nx, MPI_INT, status.MPI_SOURCE, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    done++;

    // Assign new row to the worker if there are rows left
    if (rows_distributed < ny) {
        printf("Master: Sending row %d to worker %d\n", rows_distributed, status.MPI_SOURCE);
        MPI_Send(&rows_distributed, 1, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
        rows_distributed++;
    }
}
```

After all rows have been processed, the master sends a termination signal (a special message, such as -1) to each worker, indicating that there are no more rows to process.



```
int terminate = -1;
for (int i = 1; i < size; i++) {
    MPI_Send(&terminate, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
}
```

Once a row is received, the worker computes the Mandelbrot set for each pixel in that row. This computation is parallelized across multiple threads using OpenMP. Each thread processes a different pixel in the row, helping speed up the computation.



```
#pragma omp parallel
{
    #pragma omp for schedule(dynamic)
    for (int j = 0; j < nx; j++) { // coll
        int index = j;
        // C = x0 + iy0
        double x0 = xStart + (1.0 * j / nx) * (xEnd - xStart);
        double y0 = yStart + (1.0 * start_row / ny) * (yEnd - yStart);

        // Z0 = 0
        double x = 0;
        double y = 0;
        int iter = 0;

        while (iter < maxIter) {
            double temp = x * x - y * y + x0;
            y = 2 * x * y + y0;
            x = temp;
            iter++;
            if (x * x + y * y > 4) {
                break;
            }
        } // end of while
        local_row[index] = iter;
    }
}
```

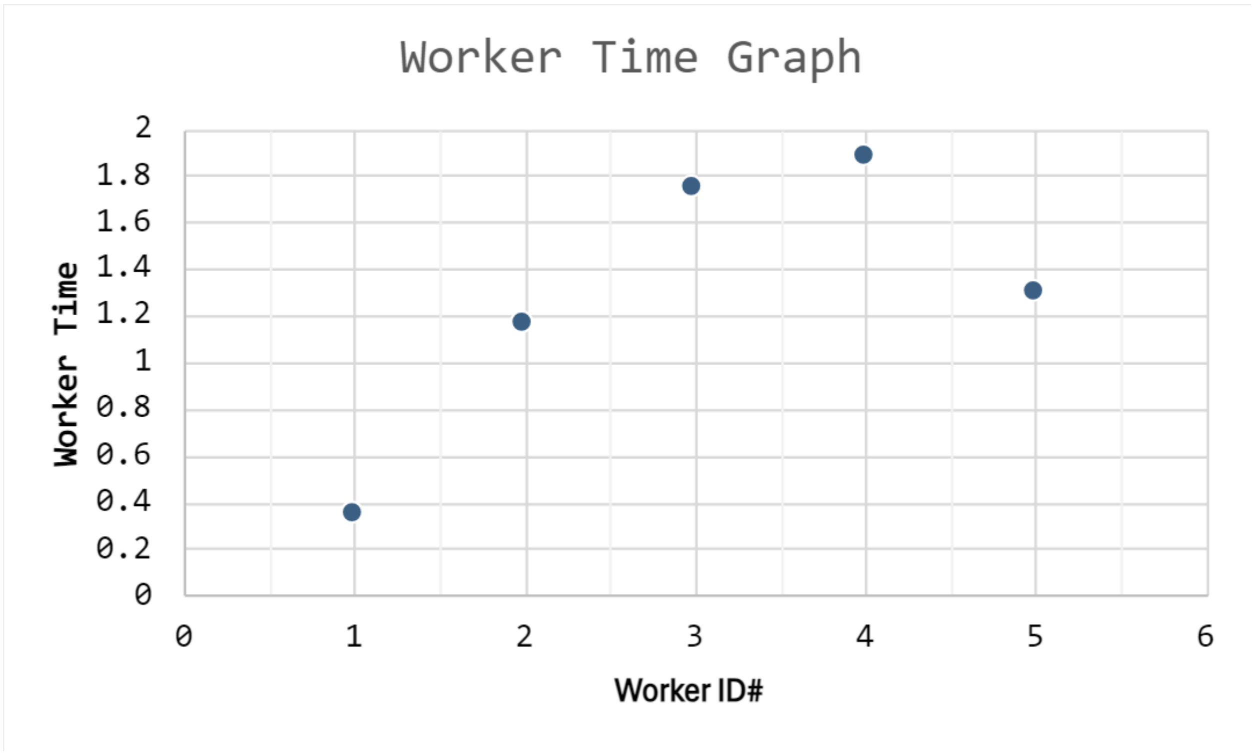
This Master-Worker model is effective for parallelizing the computation of the Mandelbrot set because it addresses the problem of load imbalance. Pixels near the boundary of the Mandelbrot set are more computationally expensive than those far from the boundary. By dynamically assigning rows to workers as they become available, the master ensures that all workers are kept busy, even if some rows take longer to compute than others. This results in better load balancing and improved performance.

3. Results and Discussion

Before optimization, the program exhibited significant load imbalance due to varying computational demands across the Mandelbrot set. Pixels near the set's boundary required more iterations, causing some MPI ranks to finish much earlier than others.

Figure 1 shows the computation time per MPI rank/worker before optimization using the original size of 600 by 400 pixels. Even on a much smaller scale than the implemented code the imbalance is clear, with Worker 4 taking [1.871718] seconds, while Worker 1 took only [0.344664] seconds, highlighting the uneven workload distribution.

Figure 1: Computation time per MPI rank before optimization



To address this, a dynamic workload distribution was implemented, assigning rows to workers based on availability. Figure 3 shows the computation times after optimization, demonstrating a significant reduction in time differences between ranks. The load is now well-balanced, with only [0.194801] seconds in-between the fastest and slowest ranks even on the larger Mandelbrot set of 3000 x 2000 pixels.

Figure 3: Computation time per MPI rank after optimization

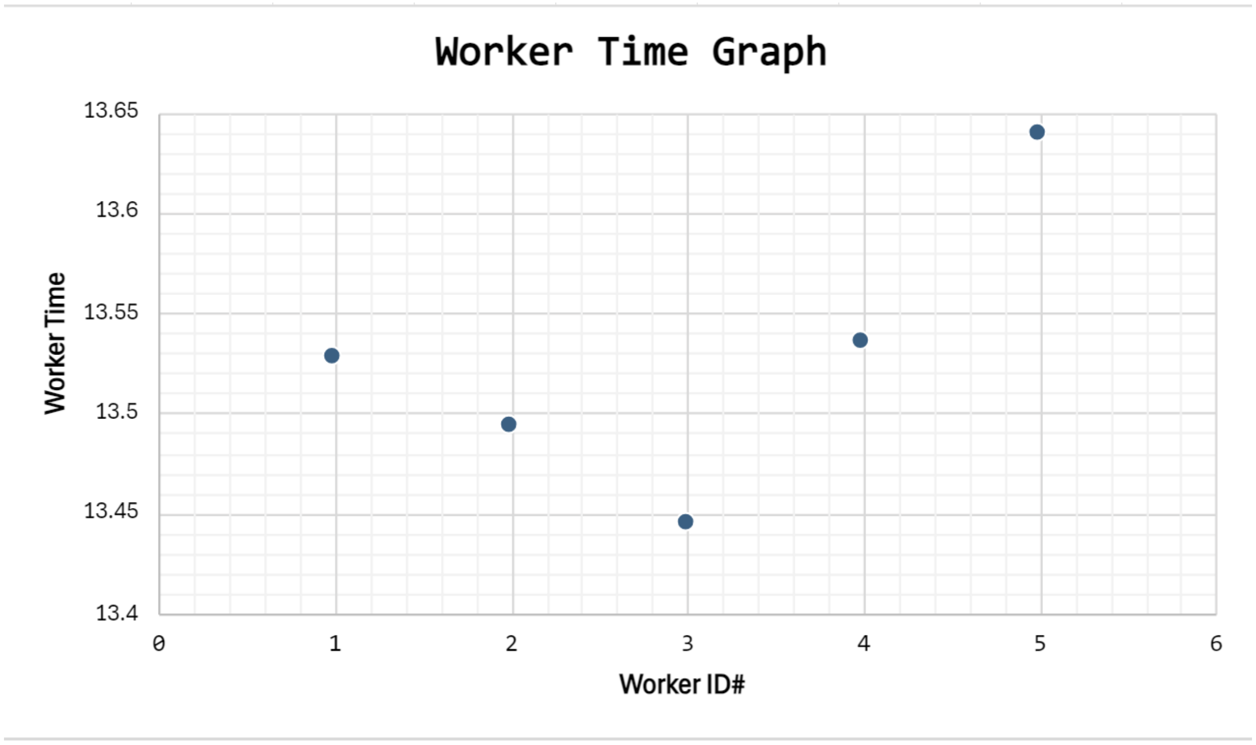
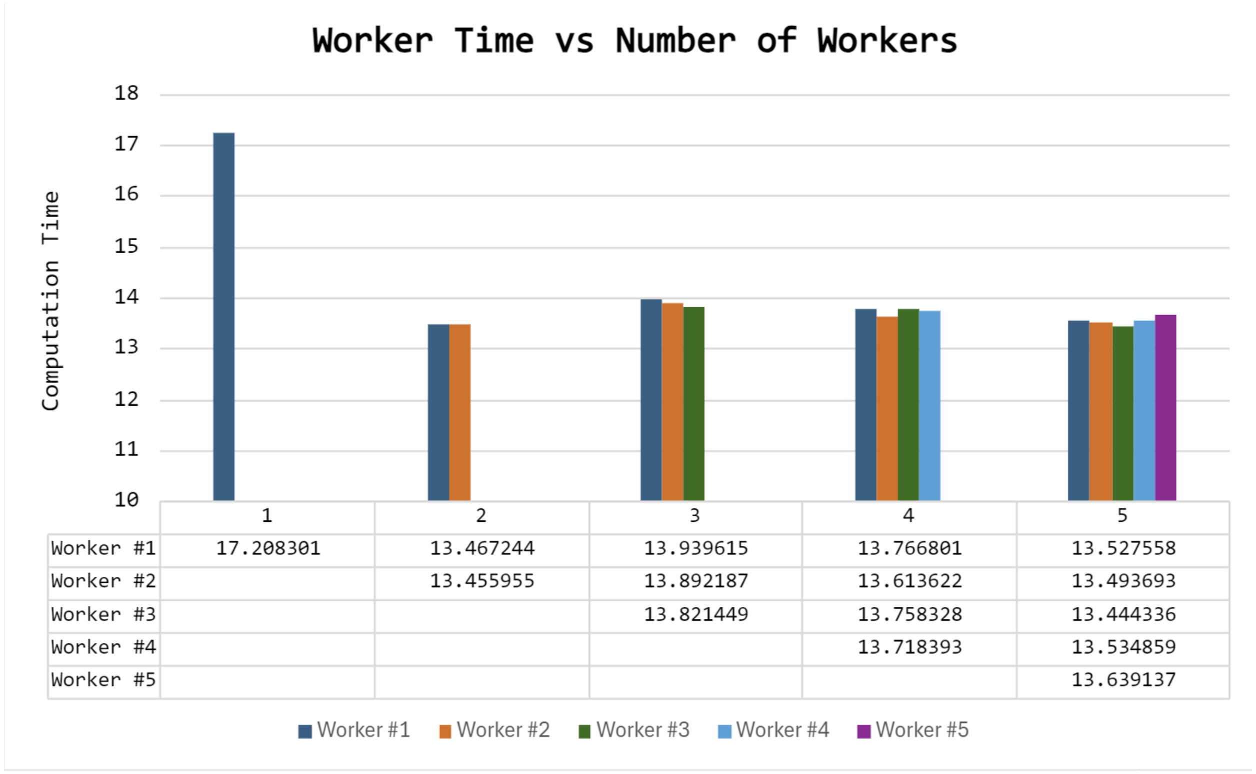


Figure 4, indicates improved speedup as the number of MPI ranks increases.

Figure 4: Computation time decreasing as MPI Rank/Workers increase



	1	2	3	4	5
Worker #1	17.208301	13.467244	13.939615	13.766801	13.527558
Worker #2		13.455955	13.892187	13.613622	13.493693
Worker #3			13.821449	13.758328	13.444336
Worker #4				13.718393	13.534859
Worker #5					13.639137

4. Conclusion

This project successfully parallelized the computation of the Mandelbrot set using MPI and OpenMP. Initial load imbalances were identified and resolved through dynamic workload distribution. The optimized implementation achieved near-uniform computation times across both MPI ranks and OpenMP threads, as evidenced by the balanced workloads. The improvements demonstrate efficient resource utilization and effective parallelization for rendering the Mandelbrot set.