High Performance Computing Project Report -Ex02-Mandelbrot Set

Roshanak Behrouz SM3800030

1. Introduction

High Performance Computing (HPC) is essential for solving complex computational problems efficiently. This project focuses on leveraging HPC techniques to render the Mandelbrot set, a famous fractal known for its intricate and self-similar patterns. Rendering the Mandelbrot set requires significant computational power, making it an ideal problem for exploring parallel computing techniques using MPI (Message Passing Interface) and OpenMP (Open Multi-Processing). In this project, I will implement a MPI and OpenMP solution to compute the Mandelbrot set. By leveraging the combined strengths of MPI and OpenMP, we can efficiently distribute the computational workload across multiple nodes and cores within the ORFEO cluster, specifically utilizing the THIN partition. This partition consists of several nodes, each equipped with two Intel Xeon Gold CPUs. Each CPU contains 12 cores, providing significant computational power. The nodes are interconnected via a high-speed network, enabling efficient data transfer between nodes. Each processor is organized into multiple Core Complexes (CCXs), each containing several cores with substantial L3 cache, ensuring efficient handling of intensive computations. This setup allows us to perform both strong and weak scaling experiments effectively on the THIN partition.

1.1 Mandelbrot set

The Mandelbrot set is a set of complex numbers that produce a fractal when visualized. The set is defined by iterating the function fc(z)=z^2+c and checking if the magnitude of z remains bounded. To visualize the Mandelbrot set, each point in the complex plane is mapped to a pixel in a two-dimensional grid, and its membership in the set is determined. For efficient storage and processing, this two-dimensional image is often stored as a one-dimensional array of pixels. Each pixel is represented as an unsigned char, with values ranging from 0 to 255.

Grayscale is used to color the image, providing a visual representation of the set. If a point is part of the Mandelbrot set, the corresponding pixel is assigned a value of 0 and is colored black. If a point is not part of the set, the pixel value corresponds to the iteration count at which the point exceeds a certain threshold, with values between 1 and 255 representing different shades of gray. This grayscale representation helps to highlight the complex boundary between the inside and outside of the Mandelbrot set.

2. Objectives

The primary objectives of this project are:

- To understand and implement parallel computing concepts using MPI and OpenMP.
- To optimize the rendering of the Mandelbrot set by distributing the computation across multiple processors.
- To analyze the performance improvements achieved through parallelization in terms of strong and weak scalability.

3. Methodology

3.1 MPI (Message Passing Interface) MPI is a standard for parallel programming that allows multiple processors to communicate with each other. It is particularly useful for distributed computing systems. In this project, MPI is used to divide the Mandelbrot set rendering task among multiple processors, each handling a portion of the image.

3.2 OpenMP (Open Multi-Processing) OpenMP is an API that supports multi-platform shared memory multiprocessing programming. It is used to parallelize the loops within each process, further optimizing the computation.

3.3 Tools and Technologies

- C++ programming language
- MPI library (e.g., OpenMPI)
- OpenMP library
- HPC cluster with SLURM workload manager

4. Implementation

4.1 Code Structure The implementation is divided into several key components:

- **Initialization:** Setting up the MPI environment and distributing tasks among processors.
- **Computation:** Each processor computes its assigned portion of the Mandelbrot set using parallel loops optimized with OpenMP.
- **Communication:** Processors communicate their results back to the master process, which assembles the final image.
- **Finalization:** Cleaning up the MPI environment and outputting the results.

4.2 Codes

4.3 Job Submission Scripts Explnation

4.3.1 Strong and Scalability

Comparison of Strong and Weak Scaling Codes
Objective

- **Strong Scaling:** Measures how the solution time varies with the number of processors while keeping the problem size fixed.
- **Weak Scaling:** Measures how the solution time changes when both the number of processors and the problem size increase proportionally.

Key Differences

- ✓ **Problem Size**
  - **Strong Scaling:** Keeps the problem size constant regardless of the number of processors or threads.

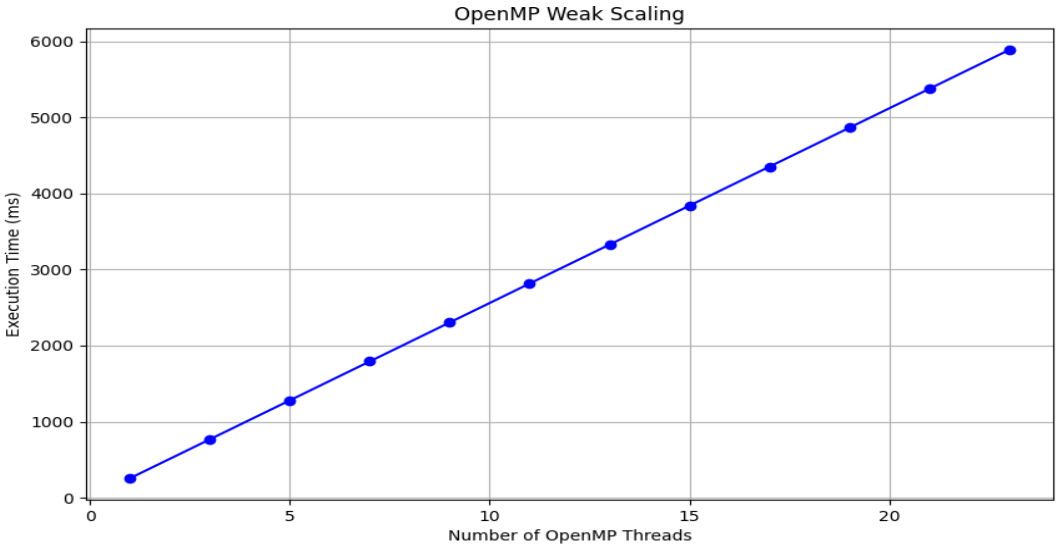
        - The width and height of the image remain fixed.
        - Example: WIDTH and HEIGHT are fixed at 4096 x 4096 pixels.
    - **Weak Scaling:** Increases the problem size proportionally with the number of processors or threads.
        - The height of the image is multiplied by the number of processors or threads.
        - Example: HEIGHT is initially 256 pixels and is multiplied by the number of processors or threads.

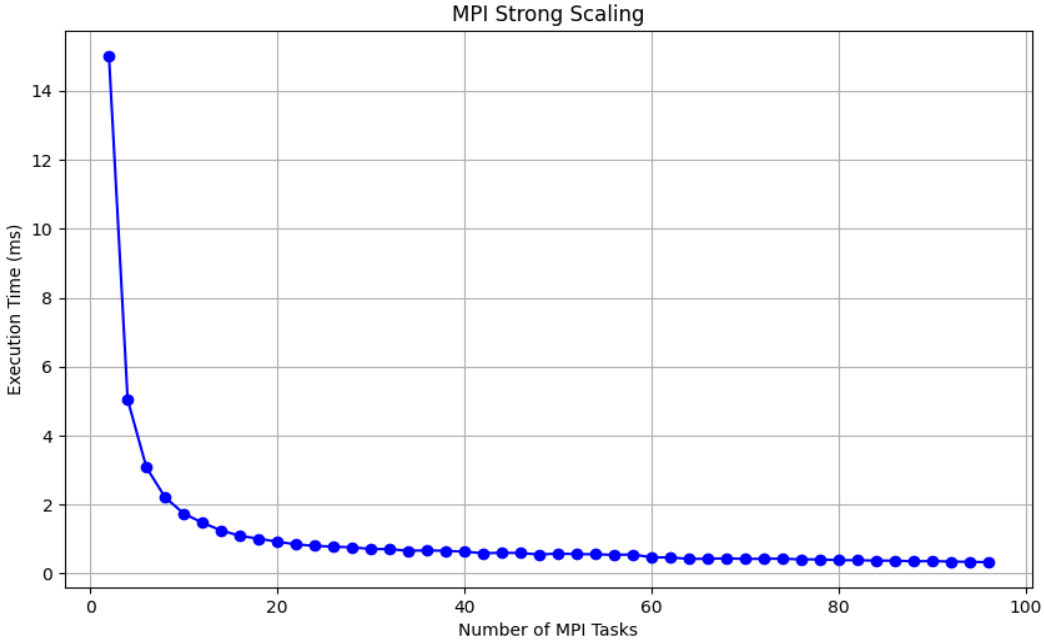
- ✓ **Loop Control Variables**
    - **Strong Scaling:** Varies only the number of processors or threads to measure the effect on execution time.
        - Example: np for MPI processes and n_threads for OpenMP threads are the only variables that change.
    - **Weak Scaling:** Varies both the number of processors or threads and the effective problem size.
        - Example: Increases np for MPI processes or n_threads for OpenMP threads, while also increasing the height of the image accordingly.

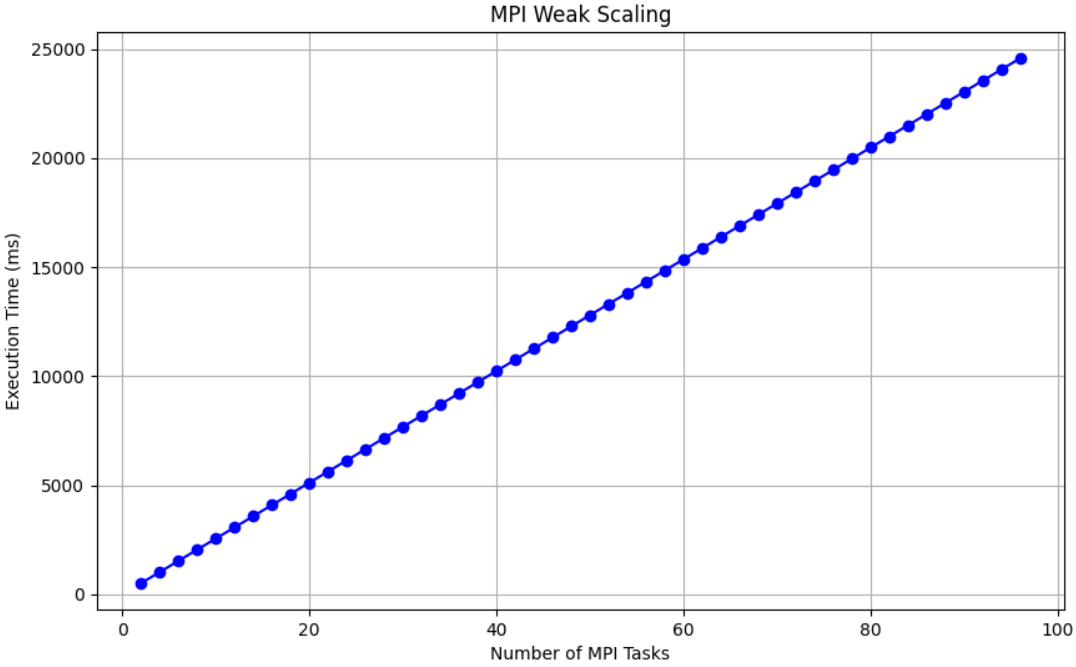
- ✓ **Output Data**
    - **Strong Scaling:** Collects data on how execution time decreases with increasing processors or threads for a fixed problem size.
        - The output file records the number of processors or threads and the corresponding execution time.
    - **Weak Scaling:** Collects data on how execution time changes with increasing problem size and number of processors or threads.
        - The output file records the number of processors or threads, the effective problem size (height), and the corresponding execution time.
- ✓ **Resource Allocation**
    - **Strong Scaling:** Focuses on the efficiency and speedup obtained by adding more computational resources to solve the same problem faster.
        - Tests the ability to decrease execution time as resources are added.
    - **Weak Scaling:** Focuses on maintaining a consistent execution time while increasing both computational resources and problem size.
        - Tests the system's ability to handle larger problems efficiently as resources increase.

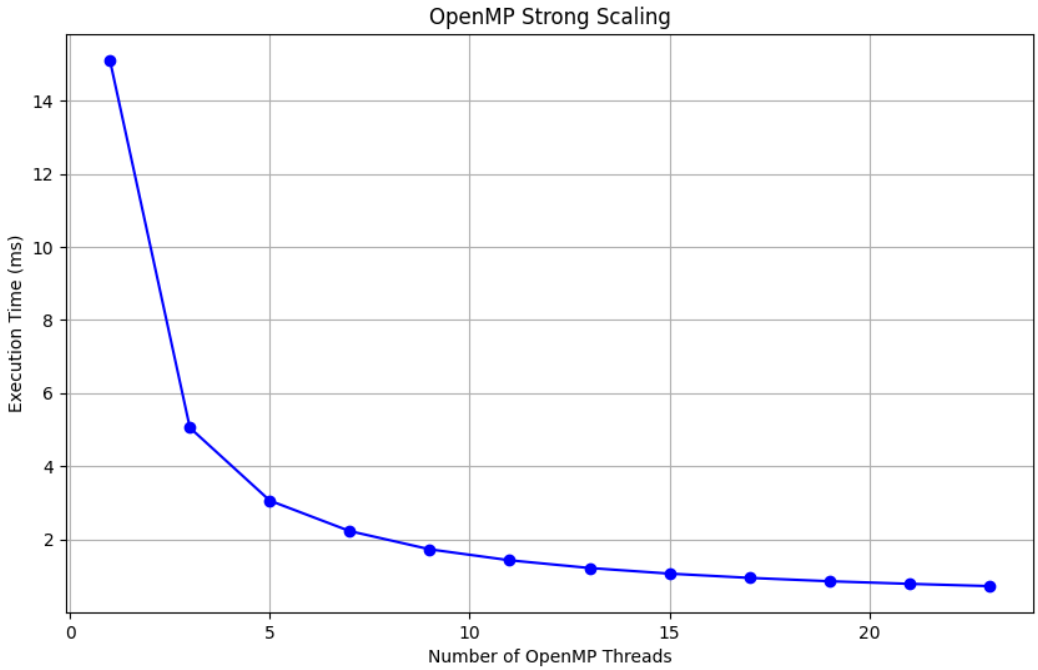
5. Results

The implementation was tested on an HPC cluster with 4 nodes, each with multiple cores. The following performance metrics were observed:


OpenMP Weak Scaling
Execution Time (ms)
Number of OpenMP Threads
0
1000
2000
3000
4000
5000
6000
0
5
10
15
20


MPI Strong Scaling
Execution Time (ms)
Number of MPI Tasks
0
2
4
6
8
10
12
14
0
20
40
60
80
100


MPI Weak Scaling
Execution Time (ms)
Number of MPI Tasks
0
5000
10000
15000
20000
25000
0
20
40
60
80
100



5.1 Strong Scalability The strong scalability tests were performed by keeping the problem size constant while increasing the number of processors. The execution time and the speedup were recorded and plotted.

5.2 Weak Scalability The weak scalability tests were performed by increasing both the problem size and the number of processors proportionally. The execution time and the efficiency were recorded and plotted.

Analysis of Results

MPI Strong Scaling

The graph for MPI Strong Scaling shows the execution time (in milliseconds) against the number of MPI tasks. The goal of strong scaling is to observe how the execution time decreases as more processors are used while keeping the total problem size constant.

- **Initial Execution Time:** The initial execution time with 2 MPI tasks is around 15 ms.
- **Trend:** The execution time decreases rapidly with the addition of more MPI tasks, flattening out as the number of tasks approaches 96.
- **Performance Improvement:** This indicates a significant reduction in execution time with the increasing number of MPI tasks. The curve's rapid drop suggests good strong scalability, meaning that the problem is efficiently divided among the processors.

MPI Weak Scaling

The graph for MPI Weak Scaling shows the execution time (in milliseconds) against the number of MPI tasks. Weak scaling measures how the execution time changes as the problem size increases proportionally with the number of processors.

- **Initial Execution Time:** The initial execution time starts low and increases linearly as the number of MPI tasks increases.
- **Trend:** The execution time increases almost linearly with the number of MPI tasks, indicating that the time taken grows proportionally to the problem size.
- **Performance Observation:** This linear trend suggests that the parallelization is effectively handling the increased workload, maintaining efficiency as the problem size scales up.

OpenMP Strong Scaling

The graph for OpenMP Strong Scaling shows the execution time (in milliseconds) against the number of OpenMP threads. Similar to MPI strong scaling, this test observes the effect of increasing the number of threads while keeping the total problem size constant.

- **Initial Execution Time:** The initial execution time with 1 OpenMP thread is around 14 ms.
- **Trend:** The execution time decreases rapidly as the number of threads increases, flattening out as the number of threads reaches 23.
- **Performance Improvement:** This indicates a significant reduction in execution time with the increasing number of threads. The curve's rapid drop suggests good strong scalability with OpenMP as well.

OpenMP Weak Scaling

The graph for OpenMP Weak Scaling shows the execution time (in milliseconds) against the number of OpenMP threads. Similar to MPI weak scaling, this test measures how the execution time changes as the problem size increases proportionally with the number of threads.

- **Initial Execution Time:** The initial execution time starts low and increases linearly as the number of OpenMP threads increases.
- **Trend:** The execution time increases almost linearly with the number of OpenMP threads, indicating that the time taken grows proportionally to the problem size.
- **Performance Observation:** This linear trend suggests that the parallelization is effectively handling the increased workload, maintaining efficiency as the problem size scales up.

Discussion

MPI Strong Scaling:

- The strong scaling results show a steep decrease in execution time as the number of processors increases, which indicates efficient distribution of tasks among the processors.

- The flattening of the curve at higher processor counts suggests that overhead and communication costs start to balance out the benefits of adding more processors.

MPI Weak Scaling:

- The weak scaling results show a linear increase in execution time, which is expected as the problem size increases proportionally with the number of processors.
- This indicates that the parallel implementation is handling the increased workload effectively, maintaining consistent performance as the system scales.

OpenMP Strong Scaling:

- The strong scaling results for OpenMP show a rapid decrease in execution time with the addition of more threads, similar to the MPI strong scaling results.
- The flattening of the curve at higher thread counts indicates the overhead of thread management and synchronization starts to become significant.

OpenMP Weak Scaling:

- The weak scaling results for OpenMP show a linear increase in execution time with the number of threads, indicating effective handling of the increased problem size.
- This consistent performance across increasing thread counts suggests that the implementation scales well with increasing computational resources.

Conclusion

The results demonstrate the effectiveness of both MPI and OpenMP in parallelizing the Mandelbrot set rendering task. Both strong and weak scaling results indicate that the implementations are efficient and can handle increasing computational loads effectively. The strong scaling results show significant improvements in execution time with more processors/threads, while the weak scaling results maintain consistent performance as the problem size scales up. These findings highlight the potential of parallel computing techniques in solving computationally intensive problems. Future work could focus on optimizing the communication and synchronization overhead to further enhance scalability.