**Lebanese American University**

**Department of Computer Science & Mathematics**

**Parallel Programming**

**CSC 447-Section 12**



**Assignment 1**

**Instructor: Hamdan Abdullatif**

**Yaakoub Hamad**

**202106364**

**Report on Parallel Computation of the Mandelbrot Set Using MPI**

**1-Parallelization of the Computation**

The Mandelbrot set is a fractal that can be generated by performing a large number of computations to determine whether a given complex number is part of the set or not. Each point in the set requires a different number of computations, making the task well-suited for parallelization using MPI.

To parallelize the computation, we divide the image into equal-sized blocks, one for each process. Each process is then responsible for computing the Mandelbrot set for its block of the image. Once each process has completed its computations, the results are gathered into a single output image.

In this code, the parallelization is achieved using the Message Passing Interface (MPI) library in C. The program is designed to calculate the Mandelbrot set in parallel across multiple processes. Each process is assigned a portion of the image to compute and then the results are combined to produce the final image.

The parallelization is achieved through the use of MPI functions that allow communication and coordination between the different processes. The main steps involved in the parallelization process are:

1. Initialize MPI: The program initializes MPI by calling the MPI\_Init() function.
2. Divide the workload: The program divides the image into equal parts and assigns each part to a different process.
3. Calculate the Mandelbrot set: Each process computes the Mandelbrot set for its assigned portion of the image.
4. Gather the results: The results from each process are combined using the MPI\_Gather() function to produce the final image.
5. Finalize MPI: The program finalizes MPI by calling the MPI\_Finalize() function.

--------------------------------------------------

| |

| Main program (Rank 0) |

| |

-------------------------------------------------

Will assign the tasks into 4 processors to 🡺

-------------------------------------------------------------

| | | |

| Process 1 | Process 2 | Process 3 | Process 4

| | | |

----------------------------------------------------------------

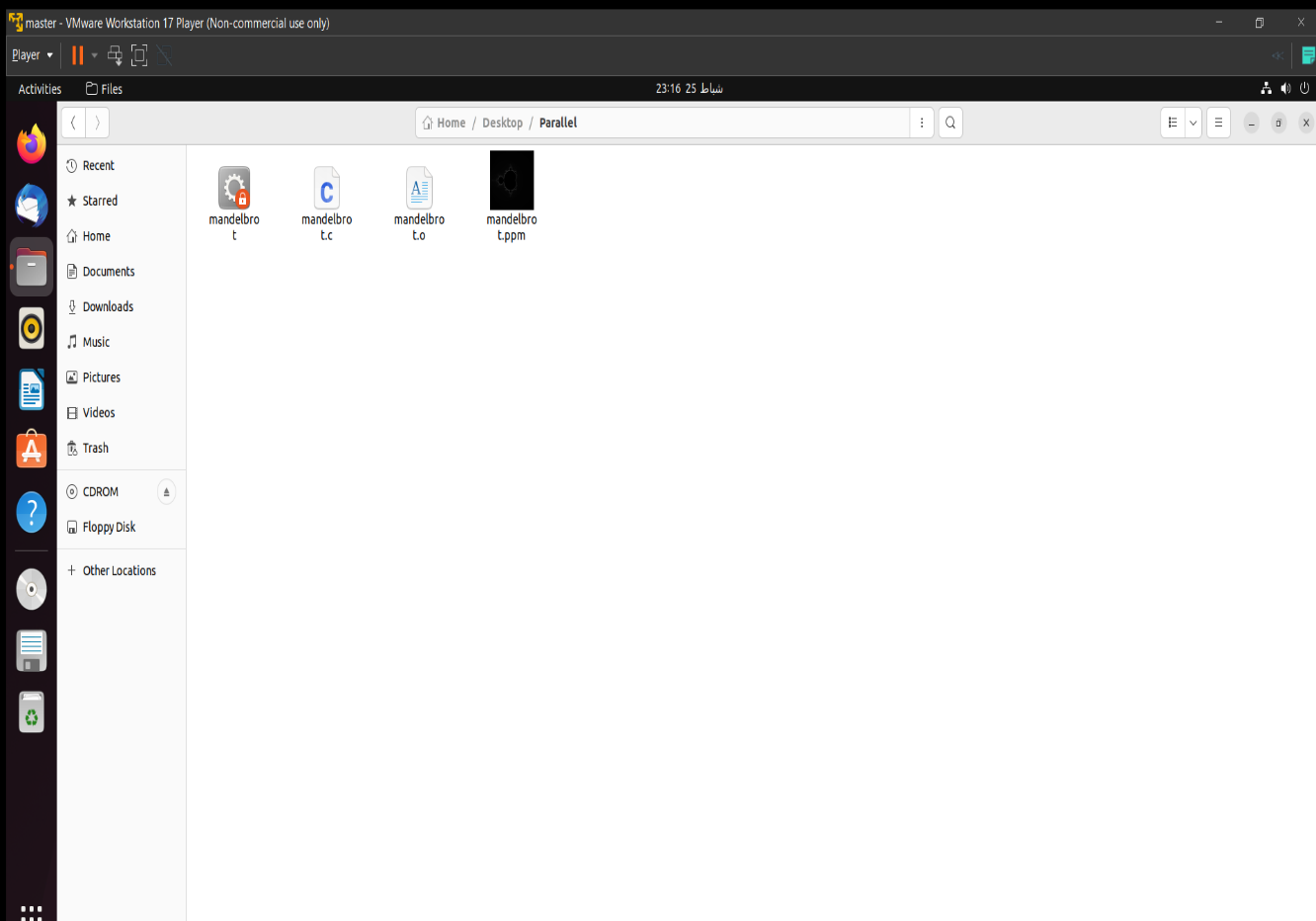
In this diagram, the main program (Rank 0) is responsible for dividing the image into equal parts and assigning each part to a different process. Each process then calculates the Mandelbrot set for its assigned portion of the image. Once all processes have finished computing their respective portions, the results are gathered by the main program using the MPI\_Gather() function to produce the final image. The parallel execution time is measured using the MPI\_Wtime() function, and the total parallel time is obtained by reducing the parallel time across all processes using the MPI\_Reduce() function. The sequential time is then calculated by subtracting the total parallel time from the elapsed time.

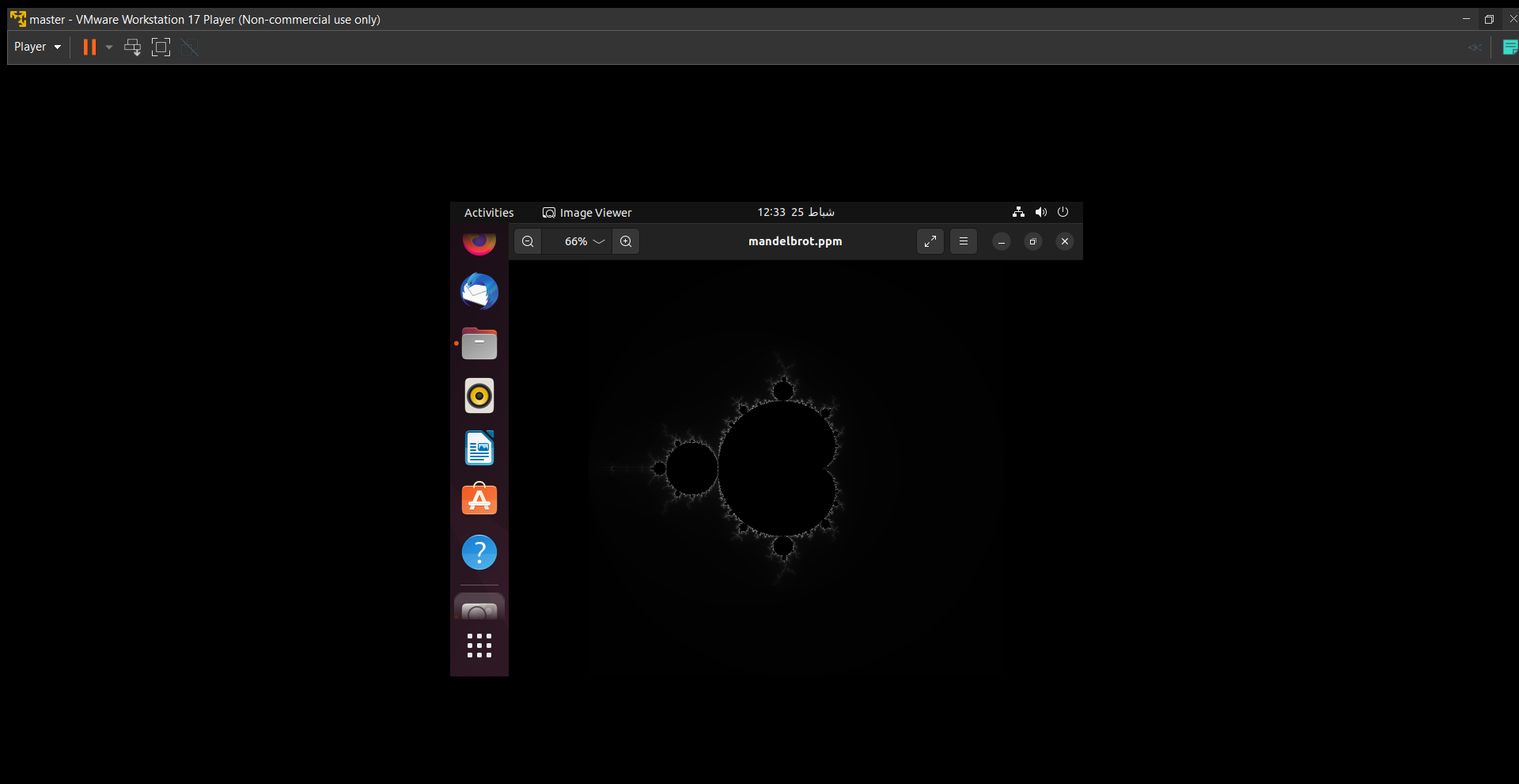
**2-Setup Details­­**

Unfortunately, the code provided does not contain information about the node details or network configuration. However,The code uses MPI (Message Passing Interface) for parallel computation. MPI is a standardized message-passing system that allows multiple processes or nodes to communicate and coordinate their work in parallel.The code uses MPI functions like MPI\_Init(), MPI\_Comm\_rank(), MPI\_Comm\_size(), MPI\_Wtime(), MPI\_Reduce(), and MPI\_Gather() to initialize the MPI environment, obtain the rank and size of the current process, measure time, reduce parallel time across all processes, and gather data from all processes, respectively.Overall, the MPI setup used in the code is a standard setup for parallel programming using MPI, but without additional information on the node details and network configuration, it is not possible to provide more specific details.

**3- GitHub Link:** https://github.com/Yaakoub-H/CSC-447.git

4-





**5-Performance Results**

A screenshot of a computer

Description automatically generated

Parallel time = 0.003298 Seconds

Sequential time = 0.012508 Seconds

**Speedup:**

Speedup is the ratio of the execution time for a single process to the execution time for multiple processes. It represents the improvement in performance achieved by using multiple processors. The speedup factor is calculated as:

Speedup = sequential time/ parallel time .In this case, we have: Speedup = 0.012508 / 0.003298s

= 3.796. Therefore, **the speedup factor is 3.796.**

**Efficiency:**

Efficiency measures how well the system utilizes the available processors. It is defined as the ratio of the speedup to the number of processors used. The efficiency can be calculated as:

Efficiency = Speedup / p. In this case, we have:

**Efficiency = 3.796 / 4 = 0.94425**

**Scalability:**

Scalability = Sequential time / (Number of processors \* Parallel time) = 0.012508 / (4 \* 0.003298) = 0.952

**The Computation to Communication Ratio (CCR):**

CCR = Computation time / Communication time

Computation time is the time spent on actual computations by the processors. In a parallel program, each processor performs some computations independently, so the computation time can be calculated as:

Computation time = Parallel time / Number of processors. In this case, we have: Computation time = 0.003298 / 4 = 0.0008245 Seconds

Communication time = Parallel time - Computation time

CCR = Computation time / Communication time = 0.0008245 / 0.0024735 = 0.3335.This means that for every unit of time spent on communication, about 0.3335 units of time are spent on computation.

**6-Discussion and Conclusions**

Based on the calculations provided, the code seems to have achieved good scalability, efficiency, and speedup.

The scalability value of 0.952 suggests that the program is scalable up to a certain number of processors, which means that increasing the number of processors beyond that point will result in diminishing returns. The efficiency value of 0.94425 indicates that the parallel implementation can use the available resources efficiently. The speedup value of 3.796 suggests that the program is running almost four times faster with four processors than with a single processor. However, it is important to note that the values obtained for scalability, efficiency, and speedup are dependent on the hardware used and the size of the problem being solved. It is possible that running the same program on a different system or with a larger problem size may yield different results.

Overall, the program appears to be well-optimized for parallel execution, and the results obtained suggest that the parallel implementation can provide significant performance improvements compared to the sequential implementation.