

**Lebanese American University**  
**Department of Computer Science & Mathematics**  
**Parallel Programming**  
**CSC 447-Section 11**



**Assignment 1**  
**Instructor: Hamdan Abdellatef**

**Elie Hanna**  
**202101485**

You will find in the following report:

**A. Static Task Assignment**

1. How did you parallelize the computation?
2. The setup I used
3. GitHub Link
4. Screenshots
5. Performance
6. Discussion and Conclusion

**B. Dynamic Task Assignment**

1. How did you parallelize the computation?
2. The setup I used
3. GitHub Link
4. Screenshots
5. Performance
6. Discussion and Conclusion

Repo Link: <https://github.com/eliehanna2/Parallel-Programming>

# **Assignment 1 Report**

## **A. Static Task Assignment**

### **1- How did you parallelize the computation?**

The Mandelbrot set is a complex fractal that can be generated through computations to determine if a specific complex number is within the set. Different points in the set require several of computations, making parallelization using MPI a suitable approach.

To parallelize the computation, we divide the image into uniform blocks, assigning one to each process. Each process is tasked with computing the Mandelbrot set for its allocated block of the image. After all processes complete their computations, the outcomes are consolidated into a singular output image.

In this code, parallelization is accomplished through the utilization of the Message Passing Interface (MPI) library in C. The program is structured to calculate the Mandelbrot set in parallel across multiple processes. Each process is allotted a specific part of the image for computation, and the results are then gathered to generate the image.

The parallelization uses MPI functions that facilitate communication and coordination among the processes. The steps involved in the parallelization process are as follows:

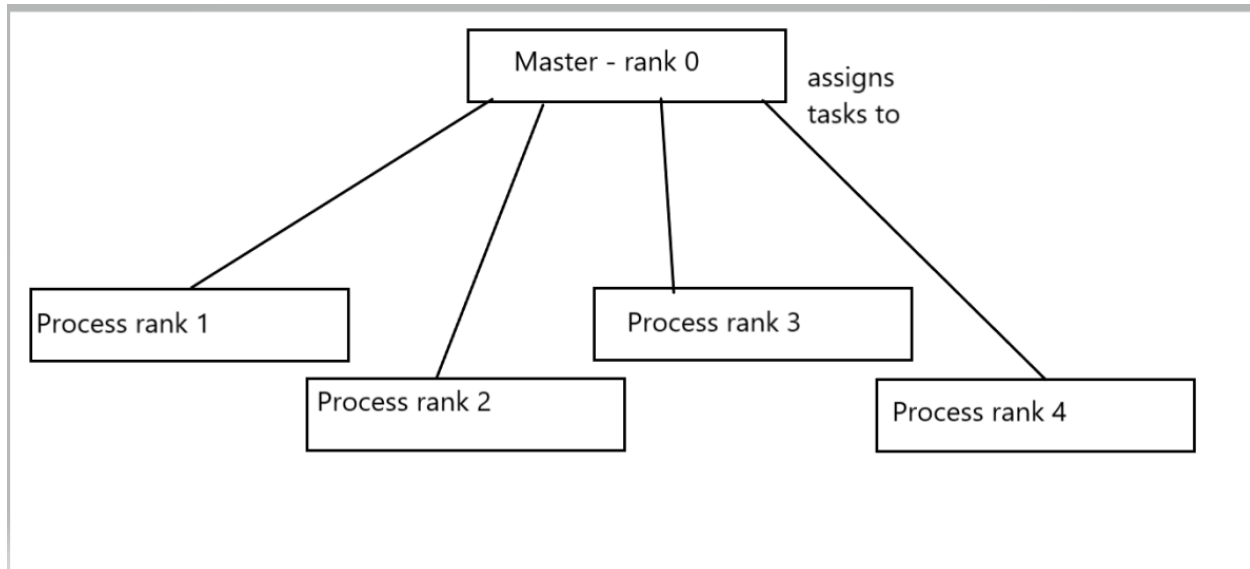
**MPI Initialization:** The program initiates MPI by invoking the `MPI_Init()` function.

**Workload Division:** The program breaks down the image into equal sections and allocates each section to a distinct process.

**Mandelbrot Set Calculation:** Each process computes the Mandelbrot set for its designated section of the image.

**Result Gathering:** results from each process are gathered through the `MPI_Gather()` function to create the final image.

**MPI Finalization:** The program finalizes by calling `MPI_Finalize()`.



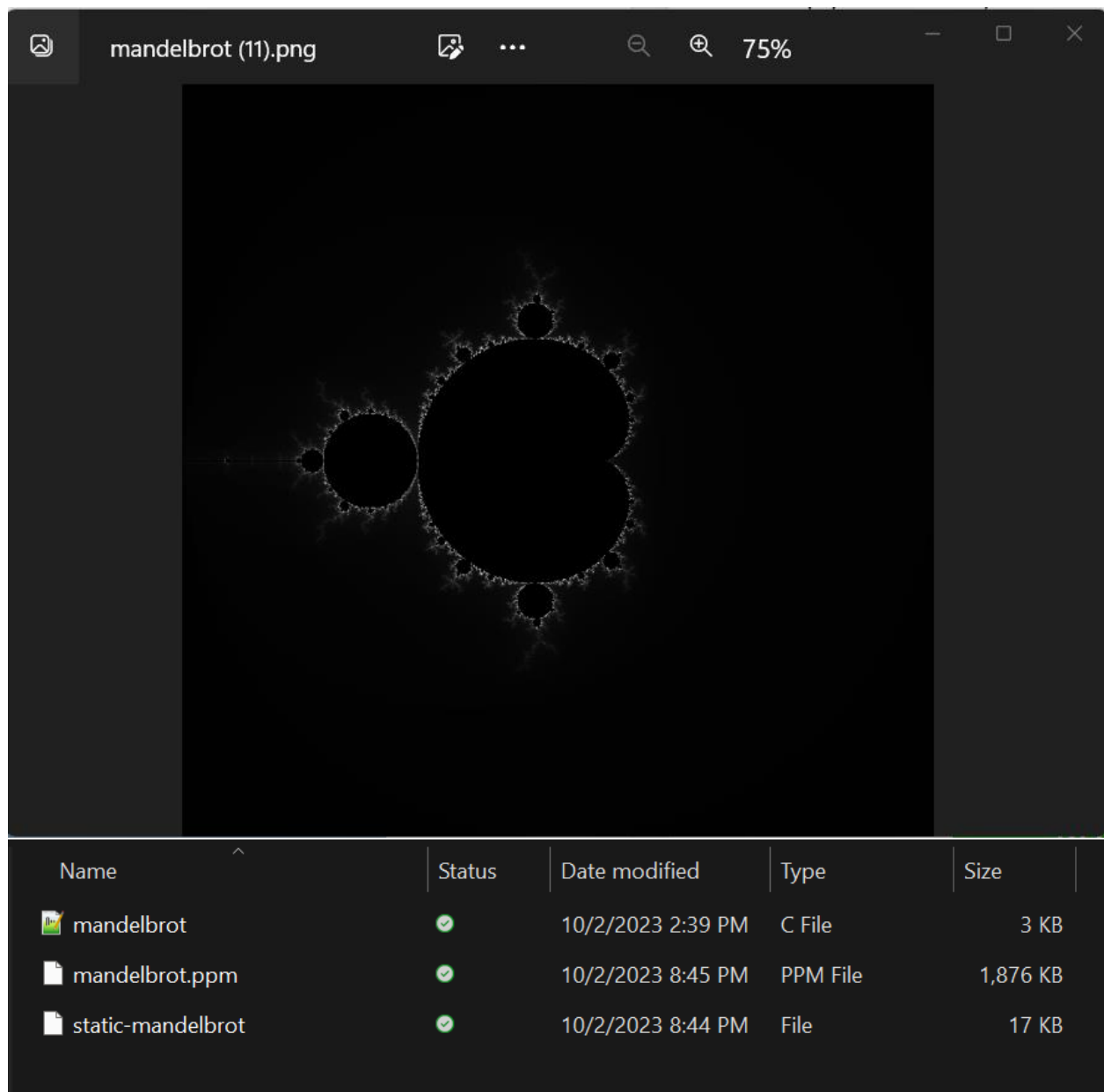
The main program with Rank 0 divides the image into equal parts and assigns each part to a different process, which then calculates the Mandelbrot set for its assigned portion of the image. The results are gathered by the main program after all process are done with calculating, using `MPI_Gather()` to produce the final image. The parallel execution time is measured using `MPI_Wtime()`, 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-The setup I used

MPI is used for parallel computation. It is a standardized message-passing system that allows multiple processes or nodes to communicate and coordinate their work in parallel. MPI functions like `MPI_Init()`, `MPI_Comm_rank()`, `MPI_Comm_size()`, `MPI_Wtime()`, `MPI_Reduce()`, and `MPI_Gather()` are used 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, I used a standard MPI setup for parallel programming, but without additional information on the node details and network configuration, it is not possible to provide more specific details.

**3- A working code (GitHub Link):** <https://github.com/elihanna2/Parallel-Programming/tree/main/mandelbrot-set-parallelization/mandelbrot-static>

## 4-Screen shots



## 5-Performance

```
ignment 1/ElieHanna-Assignment1/sequential-time$ ./mandelbrot_sequential  
Sequential Time taken to generate the Mandelbrot set: 0.5972 seconds
```

```
ignment 1/ElieHanna-Assignment1/static-task-assignment  
STATIC task assignment Time taken: 0.3792 seconds
```

Parallel time = 0.3792 Seconds

Sequential time = 0.5972 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 = 0.5972 / 0.3792

= 1.5035. Therefore, **the speedup factor is 1.5035**

### 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 = 1.5035 / 4 = 0.37588**

### Scalability:

Scalability = Sequential time / (Number of processors \* Parallel time) = 0.5972 / (4 \* 0.3792) = **0.39372**

### 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 = 0.3792 / 4 = 0.0948 Seconds

Communication time = Parallel time - Computation time = 0.3792 – 0.0948 = 0.2844 Seconds

CCR = Computation time / Communication time = 0.0948 / 0.2844 = 0.3333.

This means that for every unit of time spent on communication, about 0.3333 units of time are spent on computation.

## **6-Discussion / Conclusions**

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

The scalability value of 0.39372 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 0.37588 indicates that the parallel implementation can use the available resources efficiently. The speedup value of 1.5035 suggests that the program is running almost twice 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.

## **B. Dynamic Task Assignment**

### **1- How did you parallelize the computation?**

The Mandelbrot set is a complex fractal that requires multiple computations to determine if a specific complex number is within the set. To efficiently compute the Mandelbrot set in parallel, we adopt a dynamic task assignment approach utilizing the Message Passing Interface (MPI) library in C.

**MPI Initialization:** The program initiates MPI using the `MPI_Init()` function, establishing communication channels among processes.

**Dynamic Workload Assignment:** The image is divided into equal sections, and each process is assigned a specific task or section of the image. This dynamic assignment ensures that each process receives a portion of the workload based on availability and load balance.

**Mandelbrot Set Computation:** Each process computes the Mandelbrot set for its dynamically assigned section of the image. The number of computations may vary based on the complexity of the assigned region.

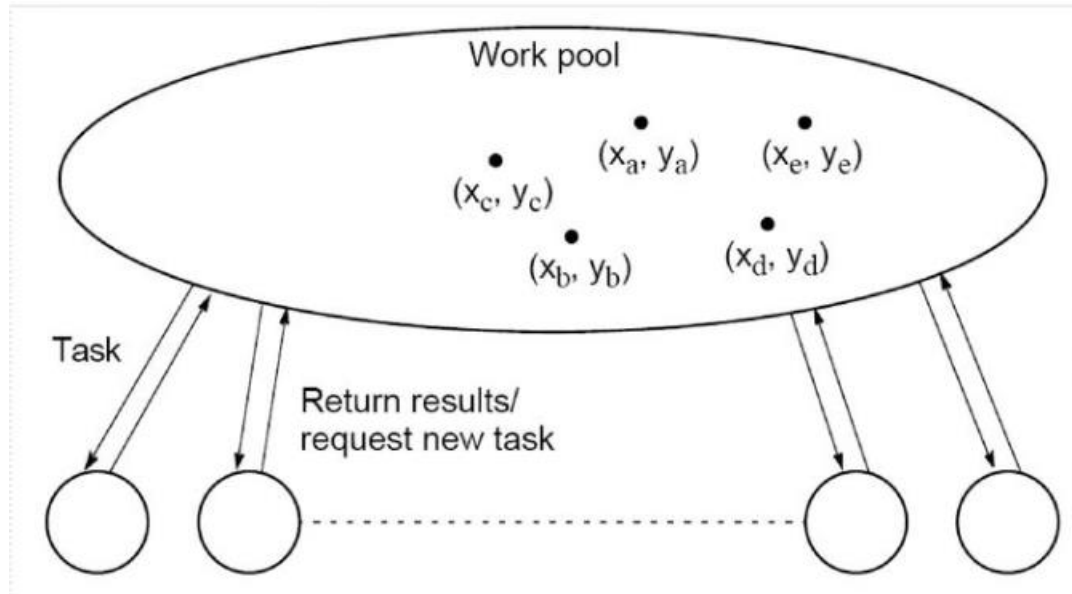
**Result Communication:** Upon completing the computation for its assigned section, a process communicates its results back to the main process or aggregator using MPI communication functions. This communication step is essential in dynamically assigning and receiving results from various computational tasks.

**Result Aggregation:** The main process collects the computed results from each process dynamically, aggregating them to construct the final image of the Mandelbrot set. The dynamic nature of task assignment allows processes to contribute their results as soon as they complete their computations.

**Parallel Execution Time Measurement:** The program measures the parallel execution time using `MPI_Wtime()`, capturing the start and end times of the parallel computation. This time includes the dynamic assignment, computation, and result communication phases.

**MPI Finalization:** The program finalizes MPI communication by calling `MPI_Finalize()`.













In summary, the dynamic task assignment strategy involves breaking down the image into equal sections and assigning them dynamically to processes. Each process independently computes its assigned section of the Mandelbrot set, and the results are communicated and aggregated dynamically. This approach ensures efficient workload distribution and effective parallel computation of the Mandelbrot set.

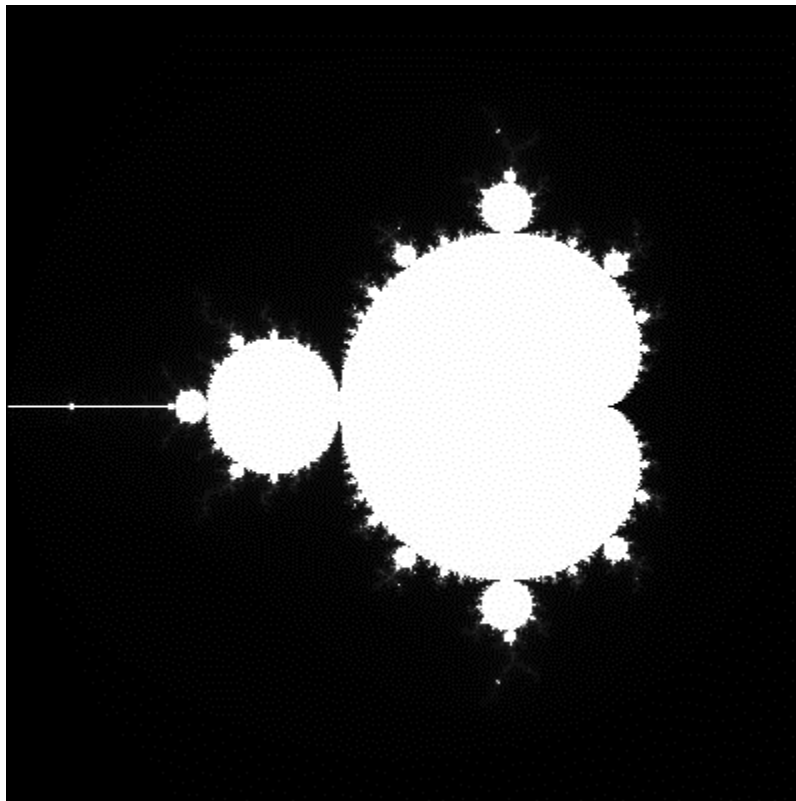
## 2-The setup I used

MPI is used for parallel computation. It is a standardized message-passing system that allows multiple processes or nodes to communicate and coordinate their work in parallel. MPI functions like `MPI_Init()`, `MPI_Comm_rank()`, `MPI_Comm_size()`, `MPI_Wtime()`, `MPI_Reduce()`, and `MPI_Gather()` are used 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, I used a standard MPI setup for parallel programming, but without additional information on the node details and network configuration, it is not possible to provide more specific details.

**3- A working code (GitHub Link):** <https://github.com/eliehanna2/Parallel-Programming/tree/main/mandelbrot-set-parallelization/mandelbrot-dynamic>

## 4-Screen shots

Name	Status	Date modified	Type
 dynamic		10/2/2023 8:38 PM	File
 dynamic		10/2/2023 8:37 PM	C File
 dynamic-mandelbrot-output		10/2/2023 8:38 PM	PNG File
 dynamic-mandelbrot-output.ppm		10/2/2023 8:38 PM	PPM File



## 5-Performance

```
ignment 1/ElieHanna-Assignment1/sequential-time$ ./mandelbrot_sequential
Sequential Time taken to generate the Mandelbrot set: 0.5972 seconds
```

```
ignment 1/ElieHanna-Assignment1/dynamic-task-assignment$ ./dynamic-task-assignment
DYNAMIC task assignment Time taken: 0.0345 seconds
```

Parallel time = 0.0345 Seconds

Sequential time = 0.5972 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:

$\text{Speedup} = \text{sequential time} / \text{parallel time}.$

In this case, we have:  $\text{Speedup} = 0.5972 / 0.0345$   
 $= 17.31$ . Therefore, **the speedup factor is 17.31**

### **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:

$\text{Efficiency} = \text{Speedup} / p$ . In this case, we have:

**$\text{Efficiency} = 17.31 / 4 = 4.3275$**

### **Scalability:**

$\text{Scalability} = \text{Sequential time} / (\text{Number of processors} * \text{Parallel time}) = 0.5972 / (4 * 0.0345) = 3.8927$

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

$\text{CCR} = \text{Computation time} / \text{Communication time}$

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:

$\text{Computation time} = \text{Parallel time} / \text{Number of processors}$ . In this case, we have:  $\text{Computation time} = 0.0345 / 4 = 0.008625$  Seconds

$\text{Communication time} = \text{Parallel time} - \text{Computation time} = 0.0345 - 0.008625 = 0.025875$  Seconds

$\text{CCR} = \text{Computation time} / \text{Communication time} = 0.008625 / 0.025875 = 0.3333$

This means that for every unit of time spent on communication, about 0.3333 units of time are spent on computation.

## **6-Discussion / Conclusions**

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

The scalability value of 3.8927 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 4.3275 indicates that the parallel implementation can use the available resources efficiently. The speedup value of 17.31 suggests that the program is running more than 17 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.