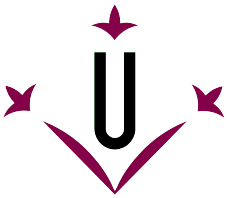
103084 - High Performance Computing

MPI Mandelbrot Fractal

Jordi Rafael Lazo Florensa

18 June 2023

Master's degree in Computer Engineering



Universitat de Lleida
Escola Politècnica Superior

Contents

List of Figures **1**

List of Tables **1**

1 Introduction **2**

2 Scalability of the Program **2**

3 Obtained Results **3**
3.1 Execution time 3
3.2 Speed up 5
3.3 Efficiency 6

4 Conclusions **7**

List of Figures

1 Execution time of all mandelbrot versions 3
2 Speed up of the different executions 5
3 Efficiency of the different executions 6

List of Tables

1 Execution time of Mandelbrot version using MPI 4
2 Speed up of Mandelbrot version using MPI 5
3 Efficiency of Mandelbrot version using MPI 6

1 Introduction

The file named *mandelbrot-seq.c* is a C program that generates an image of the Mandelbrot set. The Mandelbrot set is a complex mathematical set that exhibits a repeating pattern of fractals at different scales.

The program uses nested for-loops to iterate over each pixel in the image. For each pixel, it computes a corresponding complex number by mapping the pixel's coordinates to the complex plane. It then performs a series of iterations using the Mandelbrot formula to determine whether the complex number is part of the Mandelbrot set or not. If the number is part of the set, it is colored black, and if it is not part of the set, it is colored with varying shades of blue based on how many iterations were required to determine that it was not part of the set.

So in this assignment a parallel version will be implemented using MPI (Message Passing Interface) library in C++ to create a parallel environment where each process will be responsible for computing a subset of the image's pixels.

2 Scalability of the Program

The scalability of the original sequencial code refers to its ability to efficiently utilize increasing numbers of processing resources, such as additional CPU cores or threads, to improve its performance and generate the Mandelbrot set image faster.

The code is written using MPI which stands for Message Passing Interface. It is a library specification that defines a standard for communication and coordination between multiple processes in a parallel computing environment. MPI enables efficient message passing and data exchange among processes running on different nodes or processors.

MPI is widely used to develop scalable and portable parallel applications. It provides a set of functions and routines that allow processes to send and receive messages, synchronize execution, and perform collective operations such as broadcasting, reducing, and gathering data.

The code that has been implemented in this activity exhibits a moderate level of scalability. It leverages parallel processing using MPI to distribute the computation across multiple processes. By dividing the image into horizontal stripes and assigning each process a portion of the image, the workload is evenly distributed among the processes. This load balancing approach ensures that each process has a roughly equal amount of work to perform.

The program's scalability is constrained by several factors:

- Communication Overhead: As the number of processes increases, the communication overhead between processes may become a limiting factor. The *MPI_Send* and *MPI_Recv* functions are used to exchange data between processes, and the *printFractal* function waits for data from each process before printing the corresponding portion of the image. As the number of processes grows, the communication time may increase, affecting scalability.
- Data Dependency: Although the Mandelbrot set calculations for each pixel are independent, the program requires the results from all processes to generate the

complete image. This necessitates synchronization and communication between processes, which can impact scalability. However, the data dependency is relatively low, as each process works on a distinct portion of the image.

- Load Balancing: The workload is evenly distributed among the available processes. The image dimensions (width and height) are divided among the processes, ensuring that each process has a balanced amount of work. The load balancing technique ensures that the program scales well with an increasing number of processes.
- Memory Usage: The code allocates memory for an array (buffer) to store intermediate results before sending them to the master process for image printing. The memory usage increases linearly with the number of processes. If the memory requirements become too large, it can limit the scalability of the program.
- Image Size: The scalability of the code is affected by the size of the image being generated. As the image dimensions increase, the overall computation time and communication overhead also increase. The workload distribution among processes is proportional to the image size, potentially impacting scalability.

3 Obtained Results

3.1 Execution time

Now I will proceed to describe the data obtained that can be observed in Figure 1 which represents the execution times for different image sizes and the number of processes.

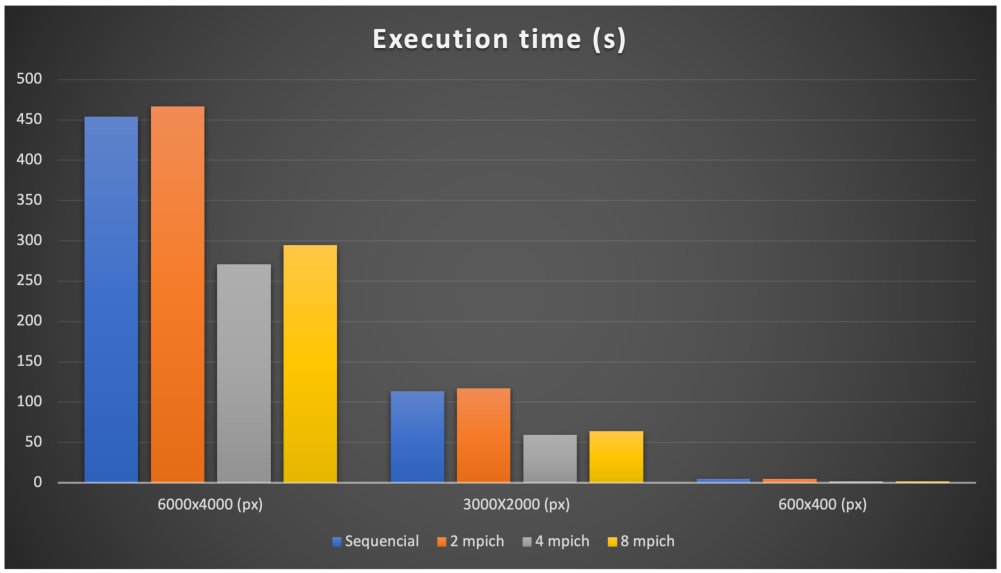


Figure 1: Execution time of all mandelbrot versions

Regarding the image size, the Figure 1 includes three different image sizes: 6000x4000 pixels, 3000x2000 pixels, and 600x400 pixels. These sizes represent increasing levels of complexity and workload for the program.

The blue bar represents the execution time when the program runs in a sequential manner, without any parallelization.

The columns labeled "2 mpich," "4 mpich," and "8 mpich" represent the execution times when the program runs with the corresponding number of parallel processes using MPI.

Execution time (s)				
	Sequencial	2	4	8
6000x4000 (px)	453,98	466,94	270,91	294,74
3000X2000 (px)	113,52	117,08	59,47	63,83
600x400 (px)	4,66	4,68	2	1,59

Table 1: Execution time of Mandelbrot version using MPI

Regarding the execution time analysis that can be seen in the Table 1 , for the 6000x4000-pixel image, the sequential execution time is 453.98 seconds. As the number of processes increases from 2 to 8, there is a significant improvement in execution time. With 4 processes, the execution time reduces to 270.91 seconds, and with 8 processes, it further decreases to 294.74 seconds. The parallel execution demonstrates scalability, as the workload is divided among multiple processes, resulting in faster computation.

For the 3000x2000-pixel image, similar scalability patterns can be observed. The sequential execution time is 113.52 seconds, and with 4 processes, it reduces to 59.47 seconds. With 8 processes, the execution time further decreases to 63.83 seconds. Again, the parallel execution offers notable improvements over the sequential execution.

For the smallest image size, 600x400 pixels, the sequential execution time is 4.66 seconds. As the number of processes increases, the execution time decreases. With 4 processes, it reduces to 2 seconds, and with 8 processes, it further decreases to 1.59 seconds. The parallel execution shows significant speedup compared to the sequential execution.

From the Table 1, we can observe that the program demonstrates scalability to varying degrees, depending on the image size, until some point.

For the largest image size (6000x4000 pixels), there is a noticeable improvement in execution time as the number of processes increases, but the scalability is not perfect. With 4 processes, there is a significant reduction in execution time, but with 8 processes, the improvement is less significant. This indicates that the program reaches a point of diminishing returns in terms of scalability for this image size.

For the medium-sized image (3000x2000 pixels), the scalability is better compared to the largest image. The execution time reduces significantly as the number of processes increases, indicating good scalability.

For the smallest image size (600x400 pixels), the program demonstrates excellent scalability. Even with a small number of processes, there is a noticeable reduction in execution time. As more processes are added, the execution time decreases further, indicating strong scalability.

However, it should be noted that this scalability of the code is not completely perfect due to the fact that the sequential time is slightly higher than the version using 2 MPI processes. It indicates that there might be some inefficiencies or bottlenecks in the parallel implementation. Parallelizing a program introduces additional overhead in terms of

communication, synchronization, and workload distribution among processes. If the parallelization overhead is high compared to the computation time, which is the case, saved by parallel processing, it can result in slower execution compared to the sequential version. In this case the overhead of parallelization outweigh the benefits of parallel execution. Because the parallel implementation requires frequent communication between processes, it introduce additional overhead and potentially slow down the execution compared to the sequential version.

3.2 Speed up

Speedup is calculated as the ratio of the sequential execution time to the parallel execution time. A speedup greater than 1 means that the parallel version is faster than the sequential version and as can be seen in Figure 2, this is true for the results obtained using 4 and 8 processes.

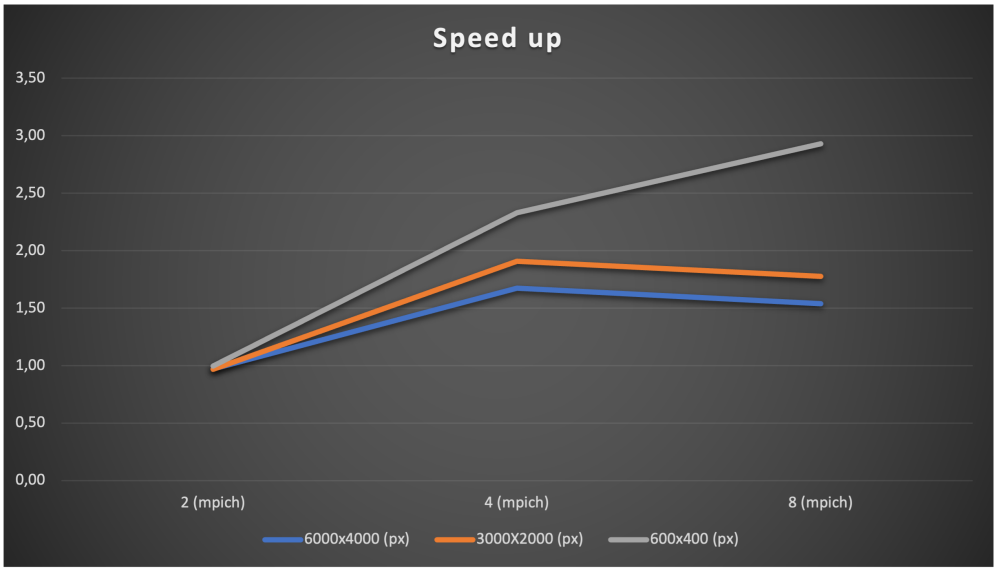


Figure 2: Speed up of the different executions

SpeedUp			
	2 (mpich)	4 (mpich)	8 (mpich)
6000x4000 (px)	0,97	1,68	1,54
3000X2000 (px)	0,97	1,91	1,78
600x400 (px)	1,00	2,33	2,93

Table 2: Speed up of Mandelbrot version using MPI

Looking at the Table 2 ,for the 6000x4000-pixel image, the speedup achieved with 2 mpich processes is 0.97, indicating that the parallel execution is slightly slower than the sequential execution. However, with 4 and 8 mpich processes, the speedup values of 1.68 and 1.54 respectively suggest that parallel execution offers a moderate improvement in performance compared to the sequential version.

For the 3000x2000-pixel image, the speedup achieved with 2 mpich processes remains 0.97, indicating similar performance to the sequential execution. However, with 4 and

8 mpich processes, the speedup values of 1.91 and 1.78 respectively suggest a noticeable improvement in performance compared to the sequential version. Although the speedup values are still less than the ideal linear speedup, they indicate that parallel execution provides better performance for this image size compared to the sequential execution.

For the smallest image size, 600x400 pixels, the speedup achieved with 2 mpich processes is 1.00, indicating that the parallel execution is equivalent to the sequential execution. However, with 4 and 8 mpich processes, the speedup values of 2.33 and 2.93 respectively demonstrate significant improvements in performance compared to the sequential version. These speedup values indicate that parallel execution is highly beneficial for this image size, resulting in faster computation.

In general, the speedup values obtained suggest that the performance improvement of the parallel execution varies depending on the image size and the number of MPI processes used and the values are less than the ideal linear speedup indicate that the parallel execution may not scale perfectly with the increasing number of MPI processes.

3.3 Efficiency

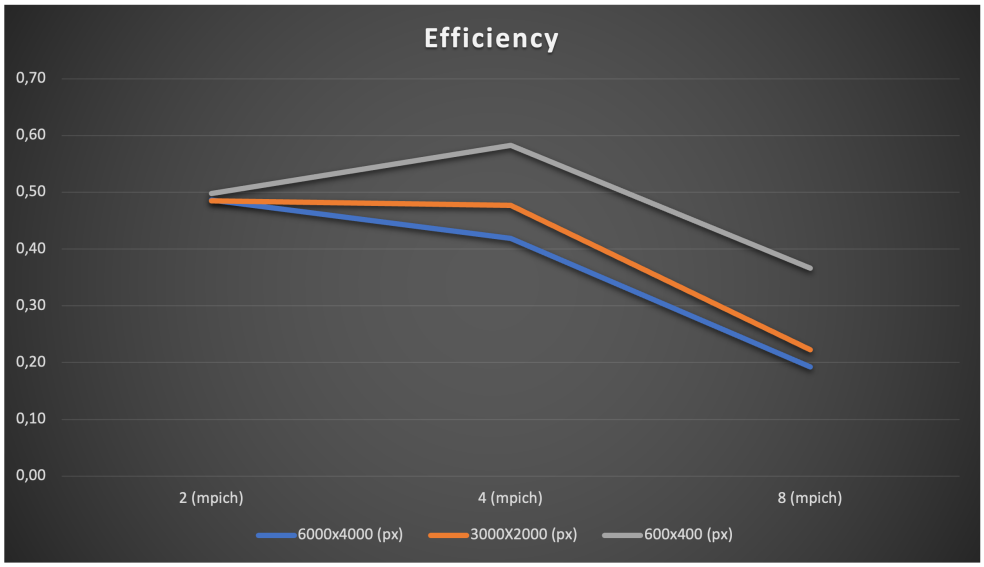


Figure 3: Efficiency of the different executions

Efficiency			
	2 (mpich)	4 (mpich)	8 (mpich)
6000x4000 (px)	0,49	0,42	0,19
3000X2000 (px)	0,48	0,48	0,22
600x400 (px)	0,50	0,58	0,37

Table 3: Efficiency of Mandelbrot version using MPI

This Table 3 and Figure 3 shows the efficiency of the parallelization of the mandelbrot code indicates how well the code scales as the number of processes increases. In general, an efficient parallel implementation should have an efficiency close to 1.0, indicating that

it is utilizing the available resources effectively. However, in practice, it is rare to achieve perfect scalability due to various factors such as the problem size, data dependencies, and resource limitations.

As we can see on the Figure 3, for the 6000x4000-pixel image, the efficiency values decrease as the number of MPI processes increases. This indicates that as more processes are added, the efficiency of utilizing those processes decreases. The efficiency values of 0.49, 0.42, and 0.19 for 2, 4, and 8 MPI processes respectively suggest that the parallel execution is less efficient for this large image size.

For the 3000x2000-pixel image, the efficiency values are relatively consistent across the different numbers of MPI processes. The efficiency values of 0.48 for 2 and 4 MPI processes, and 0.22 for 8 MPI processes suggest a moderate level of efficiency. Although the efficiency decreases with more processes, the difference is not as significant as observed for the larger image size.

For the smallest image size, 600x400 pixels, the efficiency values are generally higher compared to the larger image sizes. The efficiency values of 0.50, 0.58, and 0.37 for 2, 4, and 8 MPI processes respectively suggest better utilization of resources and higher efficiency of parallel execution.

In conclusion, while the larger image sizes exhibit lower efficiency, the smaller image size demonstrates better resource utilization and scalability.

4 Conclusions

The implementation of the Mandelbrot set program in C++ using MPI show the potential for parallel execution and scalability. By distributing the computation across multiple MPI processes, the program can handle larger image sizes and achieve faster execution times. However, the performance evaluation reveals that the scalability and efficiency of the program are influenced by several factors. These include the image size, the number of MPI processes utilized, and the effectiveness of resource utilization. For larger image sizes, the efficiency of the parallel execution may decrease due to increased communication overhead and load imbalance. To optimize performance, it is crucial to carefully manage the distribution of workload and ensure efficient communication between processes. Load balancing techniques can help distribute the computation evenly, mitigating performance bottlenecks and improving efficiency. Furthermore, analyzing the performance metrics, such as execution time and speedup, provides valuable insights into the benefits of parallel execution. It is important to have a balance between the number of MPI processes used and the size of the problem to avoid diminishing returns. Overall, while parallel execution offers the potential for improved performance in the Mandelbrot set program, achieving optimal scalability and efficiency requires careful consideration of various factors such as characteristics of the problem and the available hardware resources.