High Performance Computing final project
Exercise 2

Barrasso Marco

May 4, 2024

1 Introduction

The aim of this project was to present an hybrid OpenMP+MPI implementation to generate the Mandelbrot set followed by an analysis on the strong and weak scaling.

The Mandelbrot set is generated on the complex plane $\mathbb{C}$ by iterating the complex function $f_c(z)$ whose form is:

$$f_c(z) = z^2 + c$$

for a complex point $c = x + iy$ and starting from the complex value $z = 0$ so to obtain the series

$$z_0 = 0, z_1 = f_c(0), z_2 = f_c(z_1), ..., f_c^n(z_{n-1})$$

The Mandelbrot set is defined as the set of complex points for which the above sequence is bounded. It may be proved that once an element of the series is more distant than 2 from the origin, the series is then unbounded. Hence, the simple condition to determine whether a point c is in the set $\mathbb{M}$ is the following

$$|z_n = f_c^n(0)| < 2 \quad \vee \quad n > I_{max}$$

where I_{max} is a parameter that sets the maximum number of iteration after which you consider the point c to belong to $\mathbb{M}$. The implementation was written in C language and the code has been tested on ORFEO cluster at the Area science park of Trieste.

2 OpenMP Implementation

Initially, a 2D matrix of $n_x \times n_y$ of integers is defined via the command line. Each element in this matrix represents a pixel in an image. The process involves iterating over each pixel until a specified condition is met or the maximum number of iterations is reached.

For the OpenMP parallelization, my initial strategy was to assign one row per thread. However, I soon reconsidered this approach upon realizing that each pixel can be computed independently. This led me to adjust the strategy, allowing each thread to independently process individual pixels, optimizing computational efficiency and resource utilization.

The pseudocode below tries to give a general idea about the main loop of the function to compute the Mandelbrot set. In this loop, parallelization is achieved using the **omp parallel for**, which divides the computation across multiple threads. The outer loop iterates over the rows of the matrix, while the inner loop processes the columns, thereby assigning each pixel to a thread. The clause **schedule (dynamic)** has been used to reduce work imbalance, this scheduling approach allows threads to dynamically receive new tasks as they complete their current ones. This is particularly useful for this case, where the computational demand can vary significantly from pixel to pixel.

Algorithm 1 Calculate Mandelbrot

```
1: for i = 0 to n_y do
2:     #pragma omp parallel for schedule(dynamic)
3:     for j = 0 to n_x do
4:         c ← (x_L + j · Δ_x) + i · (y_L + i · Δ_y)
5:         val ← c
6:         k ← 0
7:         while k < Imax and |val| < 2 do
8:             val ← val^2 + c
9:             k ← k + 1
10:        end while
11:        if k == Imax then
12:            image[i, j] ← 0
13:        else
14:            image[i, j] ← k
15:        end if
16:    end for
17: end for
```

After this computation the matrix *image* will contain $n_x \times n_y$ **short int** values from 0 to I_{max}, that were converted in a PGM image through a specific function.

3 MPI Implementation

Starting from the previous OpenMP implementation i tried to integrate an MPI version achieving the hybrid approach mentioned at the begging of the report. The stategy was initially to assign a chunk of the image, so $n_x/n_{processes}$ rows. For example with $n_x = 20$ and $n_{processes} = 5$ the process 0 will have the first 5 rows of the image matrix, the process 1 from the fifth to the tenth and so on.

Upon assigning the rows, each MPI process would have invoked the previously implemented Mandelbrot function, which utilizes OpenMP to parallelize the computation across the pixels in each row. In this way the image would have been subdivided row-wise among the MPI processes, with each process further parallelizing the computation across its assigned rows using multiple threads. This would have worked and the implementation would have been quite straightforward but after thinking about this process i recognized potential issues with the work imbalance between MPI processes.This distribution of rows does not account for the varying complexity in different parts of the Mandelbrot set. Some processes, depending on the location of its assigned rows within the set, might require significantly more computation than others, leading to inefficiencies and potential bottlenecks.

After this consideration, I decided to adopt a round-robin approach for distributing the rows across the MPI processes to mitigate potential work imbalance. In a round-robin fashion, each process receives rows that are spaced out by the total number of processes, ensuring a more even distribution of workload. For example, if we have $n_x = 20$ and $n_{processes} = 3$ the rows assignement will be:

- Process 0 will have rows 0, 3, 6, 9, 12, 15 and 18.
- Process 1 will have rows 1, 4, 7, 10, 13, 16 and 19.
- Process 2 will have rows 2, 5, 8, 11, 14, and 17.

After assigning rows to each process using a round-robin approach, further parallelization was implemented within each row using OpenMP, as described before. To aggregate the results into a single image I utilized MPI_GATHERv that is ideal in scenarios like this where each process might compute a variable number of rows. However, simply gathering these rows at the root does not immediately result in a correctly assembled Mandelbrot set image. Since the rows are distributed non-sequentially among the processes, the final step involves carefully reordering these rows in the root process. This reordering is crucial as it reconstructs the original matrix layout, ensuring the correct representation of the Mandelbrot set. An image produced with this implementation is shown in Appendix 1 A.

4 Scalability

The scalability of the program has been tested on THIN nodes on the ORFEO cluster, that are configured with 2 CPU with 12 cores each. Since the assignment required to do separate scaling for the MPI and OpenMP implementation there will be two part about this topic that follow an identical approach but, of course, with different results. For both scaling the I_{max} has been set to 255 to take the results more quickly, 20 repetitions has been performed and the average time has been taken.

4.1 MPI Strong Scaling

For the MPI scaling it was required to run the program with a single OMP thread per MPI task and increase the number of MPI task, in this part the analysis was conducted on four THIN nodes. For the strong scaling the avarage time has been taken running the program with a $2 \cdot 10^3 \times 2 \cdot 10^3$ matrix and $I_{max} = 255$ varying the number of processes.

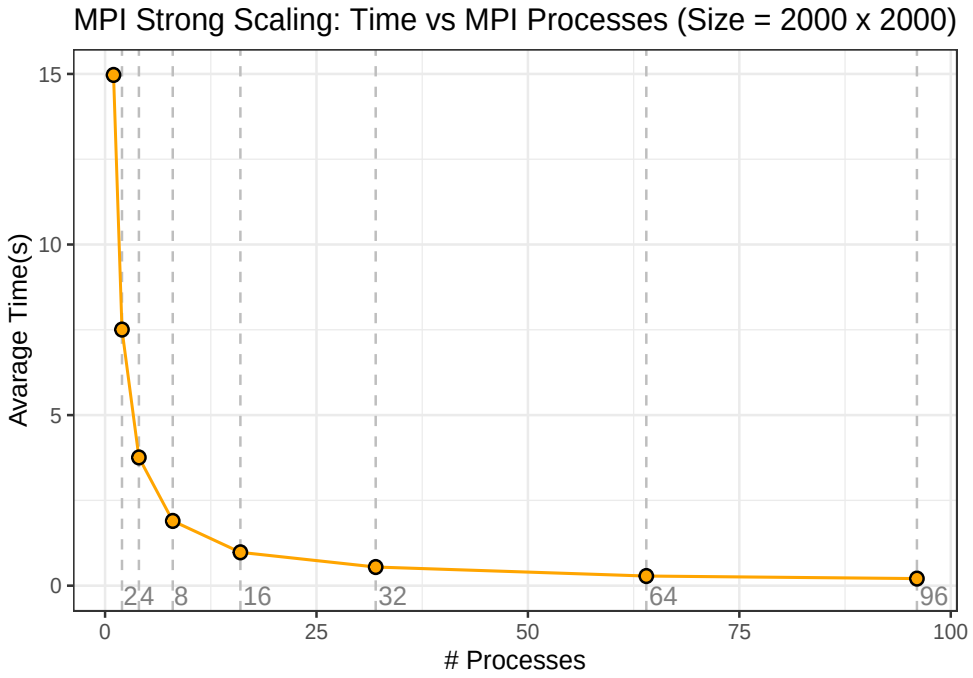

Figure 1: Strong scalability: Time vs MPI Processes.

The strong scalability was also measured in terms of Speed-up: $Sp(n,p) = \frac{T_1}{T_p}$ where T_1 is the computational time for running the program using one processor and T_p is the computational time running the same program with p processors.

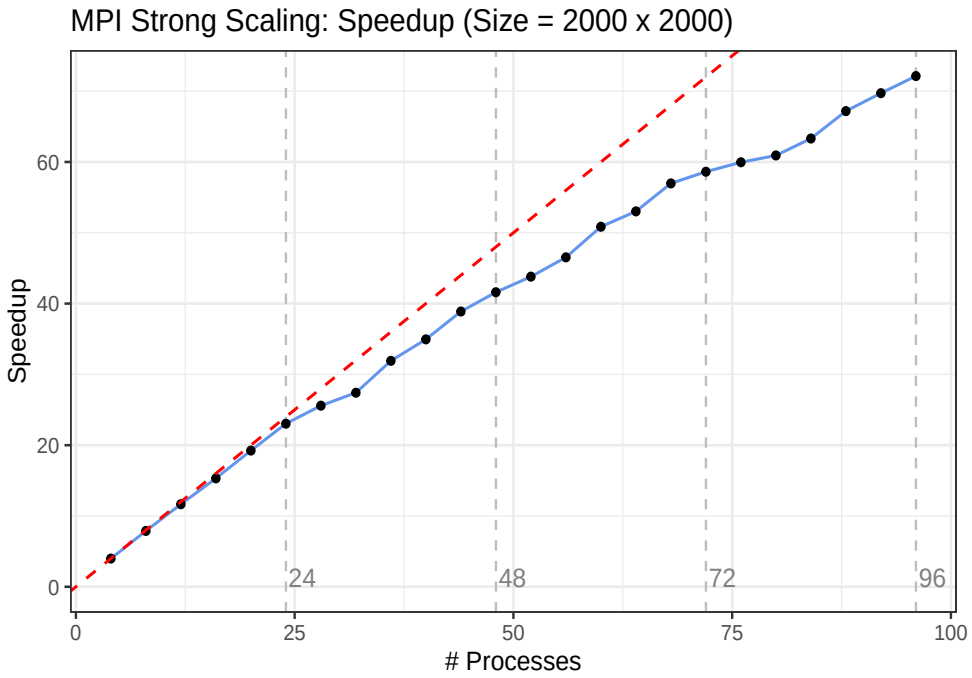

Figure 2: Strong scalability: Speedup vs MPI Processes, the red dashed line represent the ideal speedup.

4.2 MPI Weak Scaling

The weak scalability was carried out changing the height of the image proportionally to the number of processes in this way:

Height	Width	Processor	Time
1000	1000	1	3.75178
2000	1000	2	3.78700
3000	1000	3	3.76739

The weak scalability was measured in terms of Efficiency: $Efficiency = \frac{T_1}{T_p}$ where T_1 is the time taken to process the workload on one processor and T_p is the time taken to process the proportionally increased workload on p processors.

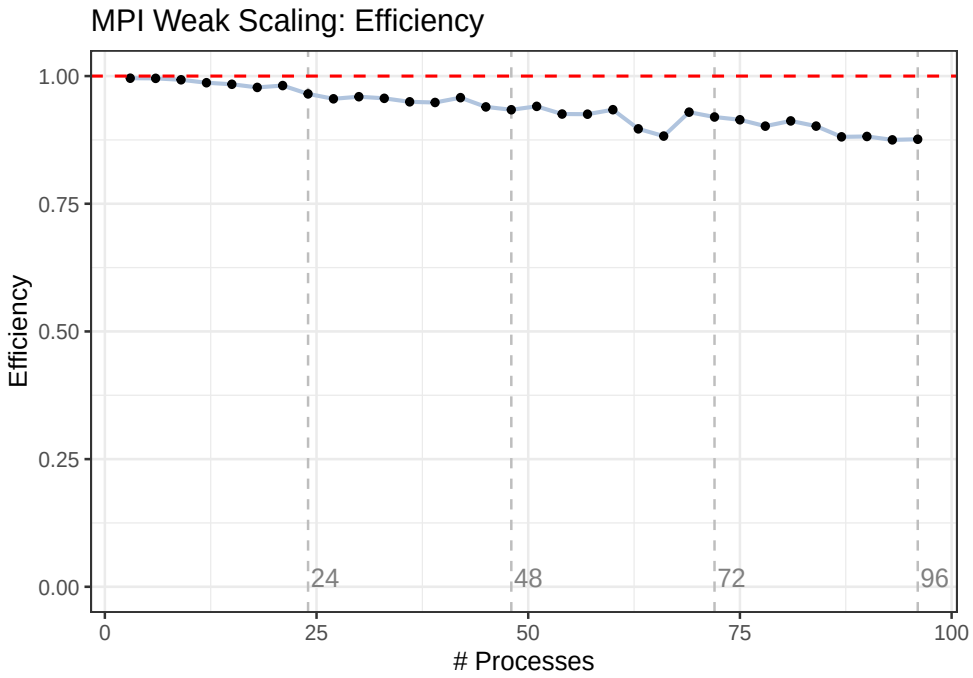

Figure 3: Weak scalability: Efficiency vs MPI Processes, the red dashed line represent the ideal efficiency.

The results indicate that, overall, the program scales efficiently, though there are deviations from the ideal scenario. In the case of strong scaling, the speed-up (Figure 2) appears to be sub-linear rather than linear, particularly evident before reaching 24 processes. Within a single node (i.e., up to 24 processes), the speed-up closely aligns with the ideal performance. However, as the communication extends to inter-node, the speed-up diverges from the ideal trend.

For weak scaling (Figure 3), despite a noticeable decrease in efficiency, the performance remains robust: efficiency does not fall below approximately 85 % up to 96 processes. This level of efficiency is quite satisfactory, demonstrating the program's capability to handle scaling effectively, even across multiple nodes.

4.3 OpenMP Stong Scaling

For the OpenMP scaling was required to run a single MPI task and increase the number of OMP threads. This scaling was performed on a THIN node. Also in this case the strong scaling measurement has been done fixing the size of the matrix to $2 \cdot 10^3 \times 2 \cdot 10^3$ matrix and $I_{max} = 255$.

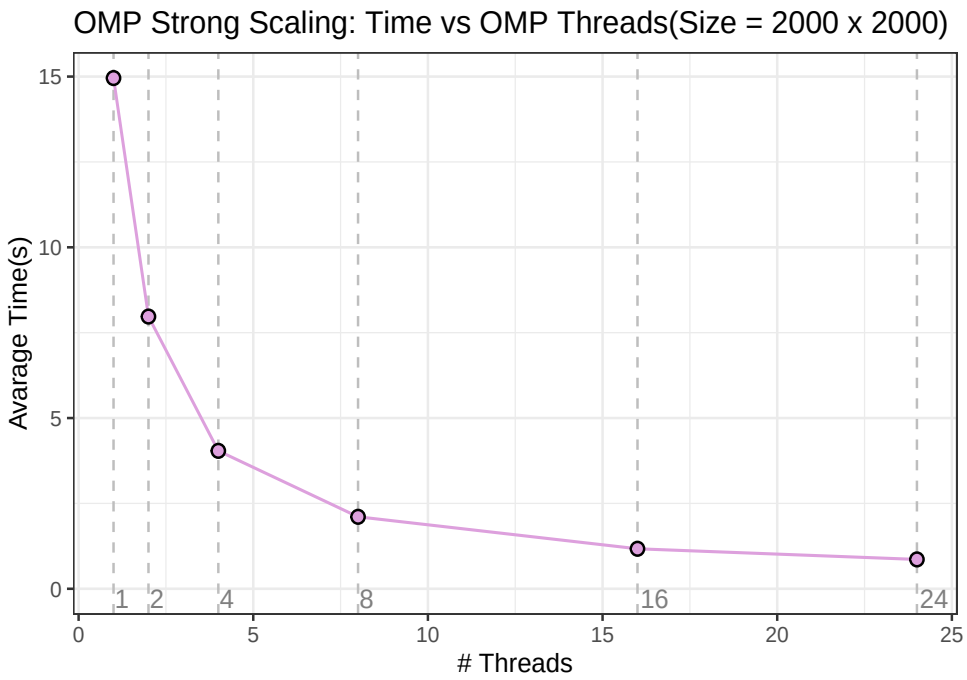

Figure 4: Strong scalability: Time vs OMP Threads.

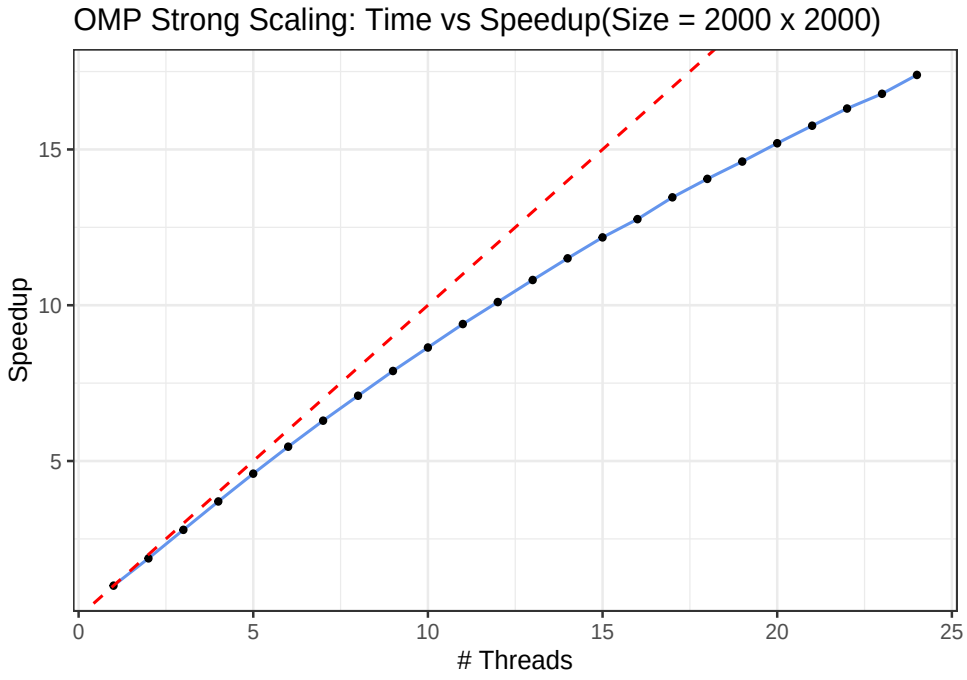

Figure 5: Strong scalability: Speedup vs OMP Threads, the red dashed line represent the ideal speedup.

4.4 OpenMP Weak Scaling

To evaluate the weak scalability of the program i changed the height of the image proportionally to the number of processes

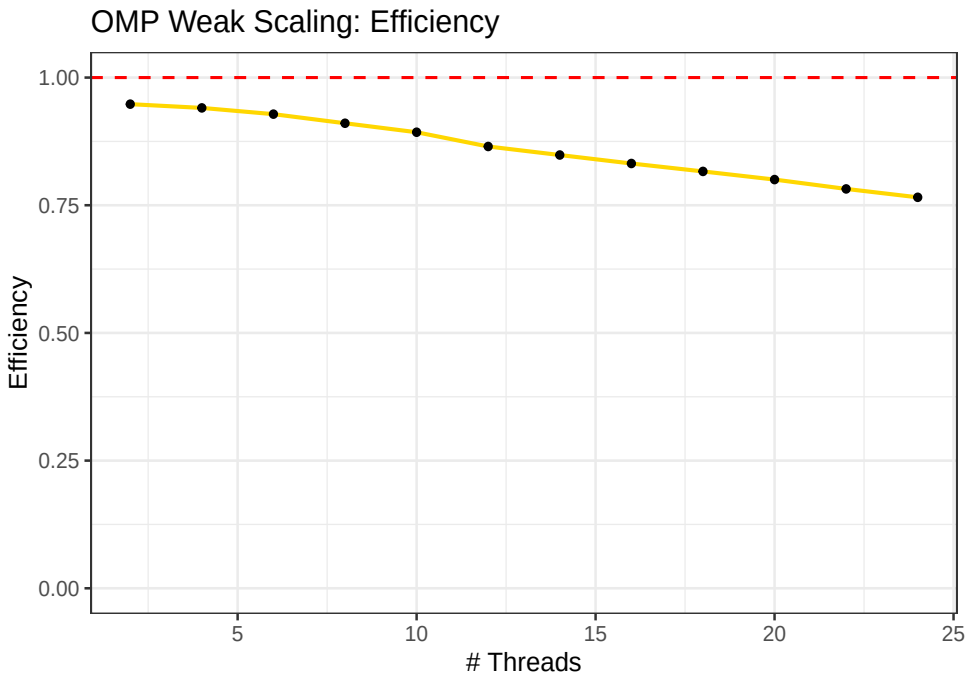

Figure 6: Weak scalability: Efficiency vs OMP Threads, the red dashed line represent the ideal efficiency.

As before the speed-up (Figure 5) of the parallelized program shows a logarithmic behavior which is more pronounced than the MPI version of the program, indicating potential overheads or inefficiencies in handling inter-process communication or synchronization.

For weak scaling (Figure 6), the analysis shows a decreasing trend in efficiency as more processes are added. However, the system maintains an efficiency level above 75 %, which suggests that the program scales reasonably well when the problem size increases proportionally with the number of processors.

References

[1] OpenMP Documentation https://www.openmp.org/resources/refguides/

[2] AreasSciencePark . Orfeo-doc 2023. https://orfeo-doc.areasciencepark.it

[3] Open MPI Documentation https://www.open-mpi.org/doc/

[4] Scalability: strong and weak scaling https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/

[5] Marco Barrasso, HPC Repository https://github.com/marcobarrasso1/HPC_Project

A Appendix 1

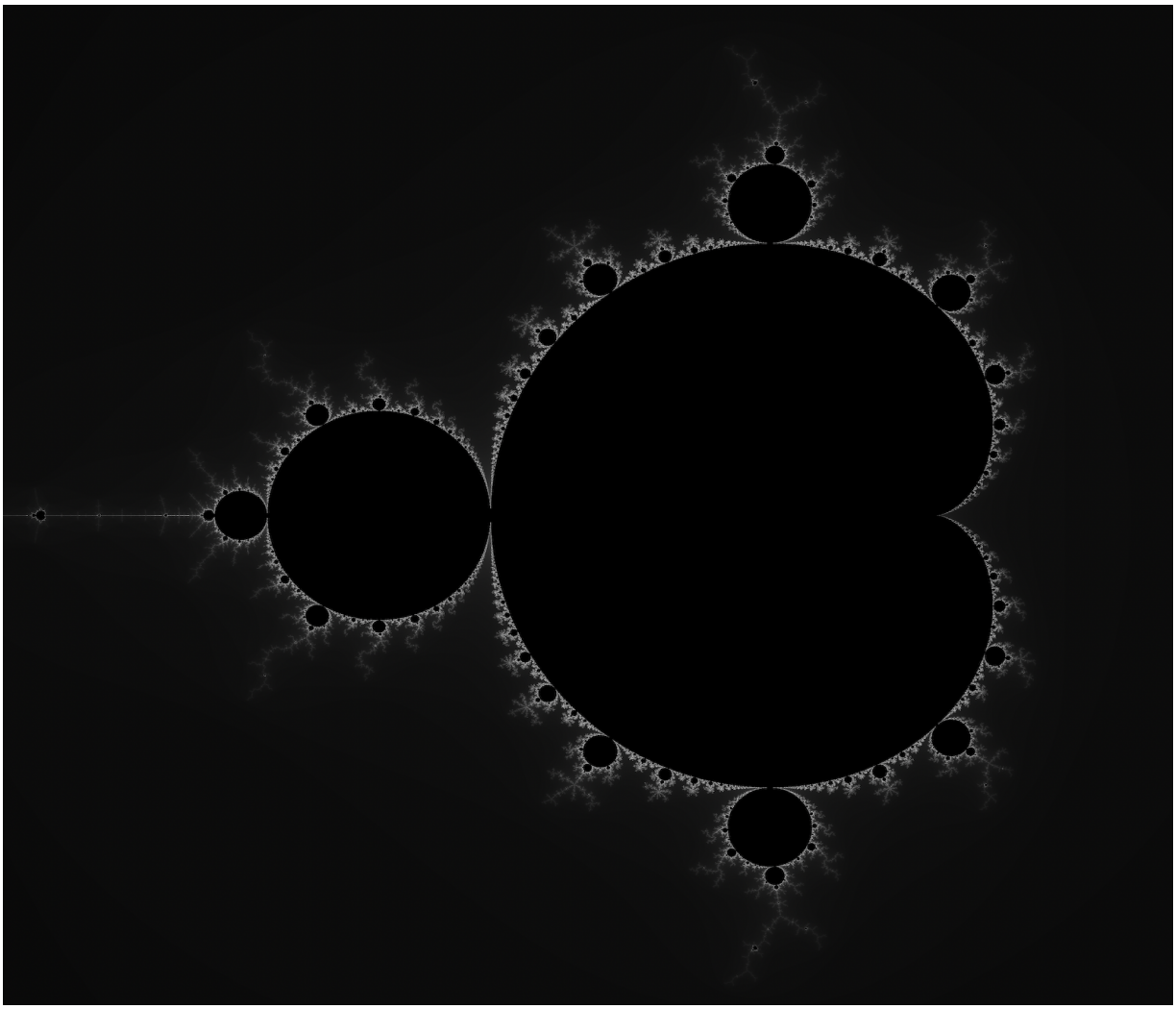

Figure 7: Mandelbrot image produced with 5000×5000, $x_L = -2, y_L = -1.5, x_R = 1.5, y_R = 1.5, I_{max} = 65535$

.