MSc in Data Science and Artificial Intelligence

High Performance Computing A.A. 2023/2024

Mandelbrot Set Generator Algorithm

A parallel implementation based on OpenMP

Prepared by:

Gabriel Masella
University of Studies of Trieste, Italy
Escuela Politécnica Superior, Universidad de Alicante, Spain

February 17, 2025

Contents

1 Introduction

The Mandelbrot set is a classic example of a parallel problem in high-performance computing (HPC). Its generation involves iterating the complex function $f_c(z) = z^2 + c$ for each point $c = x + iy$ in a specified region of the complex plane. An iteration is considered part of the Mandelbrot set if the sequence

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

remains bounded.
Typically, this is determined by checking whether $|z_n = f_c^n(0)| < 2$ or $n > I_{max}$.
This straightforward yet computationally demanding test makes the Mandelbrot set an ideal candidate for parallel implementations. In our approach, each pixel of the output image is associated with a complex point, and its corresponding value is computed independently based on the iteration count until divergence or the maximum number of iterations is reached. For this project, an OpenMP-only implementation was developed where the computational domain is discretized into a 2D matrix. The pixel values are stored using either an 8-bit or a 16-bit integer representation, depending on the maximum number of iterations. The code accepts key parameter values via command-line arguments, including the image dimensions, complex region boundaries, and maximum iteration count. Thus, it enables flexibility in the resolution and region of the computed Mandelbrot set. To address the inherent imbalance in load, the implementation employs dynamic scheduling and loop collapse techniques, recognizing that points within the Mandelbrot set generally necessitate more iterations than those residing on its boundaries.
In order to evaluate the performance and scalability of the implementation, a series of experiments were conducted on the Orfeo cluster at the Area Science Park of Trieste, utilizing EPYC nodes with Rome architecture.
The experiments involved both strong and weak scaling tests. In the strong scaling tests, the problem size was kept constant while the number of OpenMP threads was varied. On the other hand, the weak scaling tests involved increasing the problem size proportionally with the number of threads, ensuring that the computational work per thread remained constant. The experimental results yielded insights into the performance characteristics and efficiency of the employed parallelization strategy.

2 Proposed Implementation

The implementation of the Mandelbrot set generator was conceived with two primary goals in mind: computational efficiency and flexibility. The program is initiated with the input of various command-line parameters that define critical aspects of the output image, including the desired resolution (n_d and n_y), the region of the complex plane to be investigated (defined by the coordinates of the lower-left and upper-right corners), and the maximum number of iterations (I_{max}). This latter parameter is employed to determine the boundedness of the sequence. These parameters allow the user to adjust both the level of detail and the computational workload, which is crucial when dealing with high-resolution images or when a high iteration count is required to capture the intricate details of the fractal.
Once the parameters are parsed, the code calculates the corresponding increments in the complex plane for each pixel, effectively mapping the discrete grid of the image to a continuous section of the complex plane. For each pixel, the complex number c is determined, and the iterative process begins at $z = 0$. The core functionality lies in iteratively computing $z_{n+1} = z_n^2 + c$ until the magnitude of z exceeds 2 or until the iteration count reaches I_{max}. The outcome of this iterative process is stored in a matrix, with each entry representing the number of iterations performed before divergence—or zero if the point is part of the Mandelbrot set.
The implementation of this process is characterized by its strategic selection of data types for storing iteration counts. Specifically, an 8-bit unsigned char is employed when I_{max} is 255 or less, and a 16-bit unsigned short int is used when I_{max} is greater than 255. This approach is indicative of a balanced strategy that seeks to optimize memory usage while meeting the requirement for higher precision.
Parallelism is achieved through the use of OpenMP, a choice that aligns perfectly with the inherently parallel nature of the Mandelbrot computation. The nested loops over image rows and columns are transformed into a single parallel loop using the collapse clause, while dynamic scheduling is employed to counteract the load imbalance inherent in the problem—since points within the Mandelbrot set tend

to require many more iterations compared to those outside.
This parallel approach ensures that all available processing cores on modern multi-core architectures, such as the EPYC nodes with Rome architecture at the Orfeo cluster, are utilized efficiently. After the computation, the generated image matrix is optionally written to a PGM file using a dedicated utility function, enabling immediate visual inspection of the Mandelbrot set. The design and implementation demonstrate a balanced strategy that maximizes performance through parallelization while maintaining flexibility and efficiency in memory usage and output generation.

3 Experimental Results

3.1 Setup

The experimental configuration was designed around two distinct scaling tests—strong and weak scaling—to assess the performance of the OpenMP-based Mandelbrot implementation on a modern high-performance computing (HPC) system.
For the strong scaling tests, a fixed problem size was employed; the image dimensions were maintained at 9000×5845 pixels, and the region of the complex plane was set to $[-2.0, 1.0] \times [-1.0, 1.0]$. The accompanying SLURM batch script compiled the code and then iterated over a range of OpenMP thread counts ($4, 8, 16, 32, 64,$ and 128). Each iteration resulted in the generation of a CSV entry, which documented the number of threads, the fixed problem parameters, and the associated computation time. This method provided insight into the relationship between computation time and the number of threads, demonstrating a decrease in time as the number of threads increased for a constant workload.
In contrast, the weak scaling tests were designed to maintain a constant workload per thread. In this case, the problem size was scaled proportionally with the number of OpenMP threads, and the image dimensions were set to 180×117 pixels multiplied by the thread count while maintaining the same region and maximum iteration values. The SLURM script for weak scaling executed multiple runs with varying thread counts, and the resulting performance data were stored in a CSV file. This experimental configuration enabled the assessment of the implementation's capacity to manage an escalating total workload while ensuring that each thread received an equivalent share of work. Consequently, this provided a comprehensive representation of the algorithm's scalability under a weak scaling regime. Both experiments were conducted on the Orfeo cluster at the Area Science Park of Trieste, employing EPYC nodes with Rome architecture.

3.2 Strong Scaling

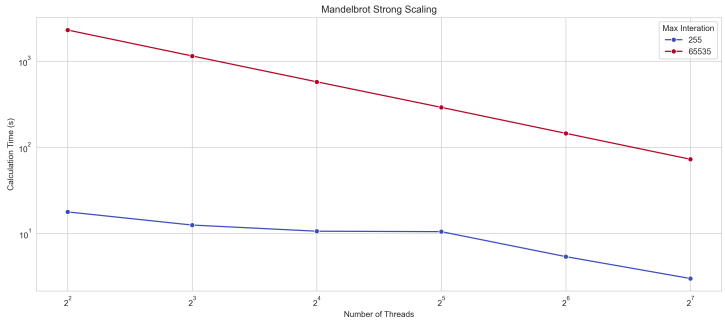

Figure 1: Time of Execution for increasing number of processors involved

In the strong scaling experiments, the problem size (9000×5845 pixels) and the region of the complex plane ($[-2.0, 1.0] \times [-1.0, 1.0]$) were held fixed while the number of OpenMP threads increased from 4 up to 128. The experimental results indicate a consistent trend: as the number of OpenMP threads increases, the calculation time decreases for both $I_{max} = 255$ and $I_{max} = 65535$. This behavior suggests that the implementation effectively utilizes parallelism, particularly at lower thread counts, where the speedup is more pronounced. However, as the number of threads increases to higher levels

(e.g., 64 and 128), the rate of time reduction slows down.This leveling off is typical in strong scaling tests due to factors such as memory bandwidth limitations, synchronization overhead, and the inherently imbalanced nature of Mandelbrot computations (some regions of the fractal require more iterations than others). The increased iteration count naturally leads to longer overall times, but the downward trend remains comparable, suggesting that the parallelization strategy maintains its effectiveness even under significantly higher iteration demands.

3.3 Weak Scaling

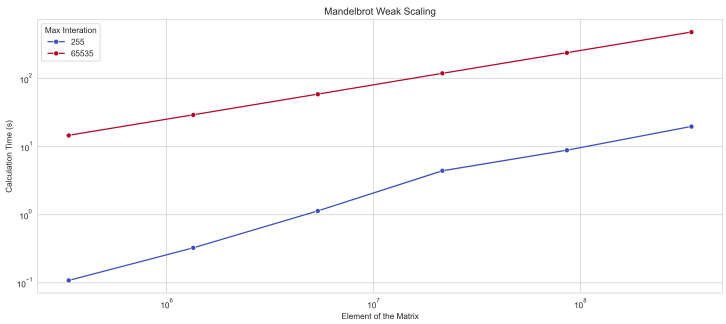

Figure 2

For the weak scaling tests, the problem size was scaled in proportion to the number of OpenMP threads, ensuring that each thread retained roughly the same amount of work. Under these conditions, if the parallel implementation scaled perfectly, the total execution time would remain constant as the total problem size increased. However, the results obtained from the execution of these tests demonstrate a substantial increase in computation time as the total number of elements increases. This phenomenon can be attributed to the additional overhead associated with larger memory footprints and potential scheduling inefficiencies. For both $I_{max} = 255$ and $I_{max} = 65535$ values, the computation times deviate from the ideal, flat line, with the higher iteration count consistently requiring more time. Despite this growth, the relatively steady slope indicates that the overhead does not grow disproportionately; the implementation can still handle larger overall workloads with more threads, albeit with some performance penalty. Overall, the weak scaling behavior underscores that while the approach remains viable for increasing problem sizes, practical factors—like memory bandwidth, synchronization overhead, and subtle load imbalances—can still limit perfectly constant execution times.

4 Conclusion

In summary, this project demonstrated the effective use of OpenMP for computing the Mandelbrot set on modern multi-core systems. By mapping a region of the complex plane to a 2D-pixel matrix and iteratively computing $f_c(z) = z^2 + c$, the implementation achieved robust parallel performance through dynamic scheduling and loop collapse. The experiments, conducted on EPYC Rome nodes at the Orfeo cluster, revealed that while strong scaling shows clear speedups at moderate thread counts, overheads such as memory bandwidth and synchronization limit performance gains at higher concurrency. Weak scaling tests, with a constant workload per thread, highlighted similar challenges as problem size increases, though the approach remains viable. Overall, the work underscores the balance between computational efficiency and scalability in parallel fractal generation, providing valuable insights into both the benefits and limitations of the adopted strategies.