Exercise 2 HPC

Edoardo Zappia

May 2024

1 Introduction

The Mandelbrot set, a renowned fractal, is created through iterations of a simple function on complex numbers. It constitutes the collection of complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z = 0$. It's demonstrable that if an element z_i of the series exceeds a distance of 2 from the origin, the series will diverge. This fundamental property is instrumental in generating the Mandelbrot set: for each complex number c, the function $f_c(z)$ is iterated until either the series diverges or a maximum number of iterations is reached. Mathematically, the condition to verify is:

$$|z_c| \leq 2 \quad \text{or} \quad n \geq I_{\max}$$

This condition ensures that the generated Mandelbrot set remains bounded within a predefined threshold. Computational generation of the Mandelbrot set entails evaluating this condition for a multitude of complex numbers across a 2D grid, resulting in a visual representation akin to an image. Each pixel in the resultant image corresponds to a complex number, and its color is determined by whether the corresponding series remains bounded within the maximum number of iterations $I_{\max}$. The Mandelbrot set's computation, owing to its inherent independence across points, lends itself naturally to parallelization. This characteristic renders the problem "embarrassingly parallel," facilitating efficient and scalable parallelization without necessitating extensive coordination among processors or cores.

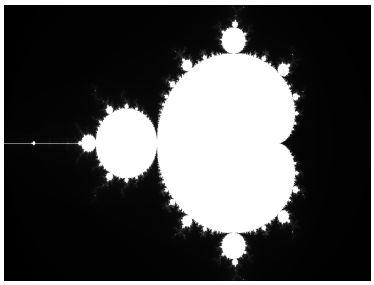


Figure 1: Mandelbrot set $[-2, 1] \times [-1, 1]$

2 Parallelization

The problem lends itself perfectly to parallelization. To compute the points of the set, both distributed memory parallelization (MPI) and shared memory parallelization (OMP) have been leveraged.

2.1 Parallelization implementation

The code implements a program for parallel generation of the Mandelbrot set using both MPI (Message Passing Interface) and OpenMP. Here's how the implementation of both is integrated:

The program begins by initializing MPI to enable communication and coordination among multiple processes working together. A threading level is specified through MPI_Init_thread, ensuring that only the main thread handles MPI calls, while secondary threads are single-threaded with respect to MPI. This ensures MPI functions in a multi-threaded environment.

Subsequently, the program obtains information about the total number of processes and the ID of each process within the MPI_COMM_WORLD communicator. Each process then calculates which parts of the Mandelbrot image it should generate based on the total number of rows and its own ID.

To ensure fair workload distribution, the total number of Mandelbrot image rows is evenly divided among the various processes. The last process may also receive any remaining rows if the total number of rows is not perfectly divisible by the number of processes.

Using OpenMP, each process generates its parts of the Mandelbrot image in parallel. The number of OpenMP threads is set using omp_set_num_threads to specify how many threads will be used to execute work in parallel within each MPI process. The #pragma omp parallel for directive automatically distributes the workload among the available threads on each process, allowing them to work in parallel. This approach maximizes the utilization of available computing resources on each node and accelerates image generation.

After generating the image, processes use MPI_Gatherv to gather and combine their image parts on a single process. Finally, the root process writes the final image to a file in PGM format.

In summary, the integration of MPI and OpenMP allows for full utilization of available parallel computing resources, improving the overall performance of Mandelbrot set generation and enabling rapid exploration of a vast input space. The fair division of work among the various processes ensures efficient utilization of available resources and maximizes the scalability of the program on parallel architectures.

2.2 Strong Scaling

Strong scaling is a concept in the context of computational parallelization, referring to the ability of an algorithm or application to maintain performance proportional to the number of added computational resources while keeping the

problem size constant. In simpler terms, it measures how much faster a program becomes when adding more computational resources to solve the same problem.

Mathematically, scalability of strong scaling can be expressed as:

$$S(n) = \frac{T(1)}{T(n)}$$

where:

- $S(n)$ represents the speedup, which is the ratio of the execution time on a single processor to the execution time on n processors.
- $T(1)$ is the execution time on a single processor.
- $T(n)$ is the execution time on n processors.

An ideal strong scaling would achieve linear speedup, where doubling the number of processors halves the execution time. Strong scaling is a key concept in the design and optimization of parallel applications, as it provides a measure of the scalability of an algorithm and can help determine the optimal number of computational resources to use to solve a given problem in the shortest possible time.

In our implementation for strong scaling with MPI processes, we maintained the image size, hence the number of rows and columns, respectively, at 1600 and 2400, while fixing a single OMP thread. Conversely, in the implementation of strong scaling with OMP threads, we kept the image size, hence the number of rows and columns, respectively, at 600 and 800, while fixing a single MPI process.

2.3 Weak Scaling

Weak scaling is a concept in the context of computational parallelization that evaluates the ability of an algorithm or application to maintain a constant amount of work per processor as the problem size and the number of processors are increased proportionally. In other words, weak scaling measures how well an algorithm can handle an increase in problem size by proportionally distributing the workload among the added processors.

Mathematically, scalability of weak scaling can be expressed as:

$$E(n) = \frac{T(1)}{n \times T(n)}$$

where:

- $E(n)$ represents efficiency, which is the ratio of the execution time on n processors to the execution time on a single processor.
- $T(n)$ is the execution time on n processors.
- $T(1)$ is the execution time on a single processor.

An ideal weak scaling would have constant efficiency as the number of processors varies. In other words, if the problem is multiplied by n and the number of processors is also multiplied by n, the execution time remains constant.

Weak scaling is important for evaluating the ability of a parallel system to effectively scale with increasing problem sizes and numbers of processors, helping to determine if a parallel solution is suitable for handling larger workloads.

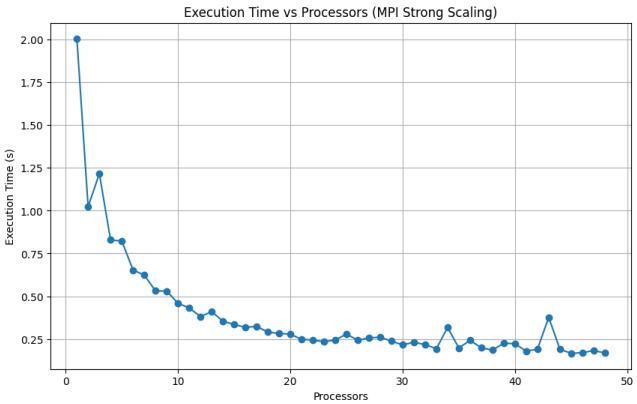
In our implementation for weak scaling with MPI processes, we varied the size of the image, thus the number of rows and columns, respectively, starting from 1000 by 1000 up to 48000 by 1000. Similarly, in the implementation of weak scaling with OMP threads, we varied the size of the image, hence the number of rows and columns, respectively, starting from 1000 by 1000 up to 48000 by 1000.

2.4 Hardware Specifications

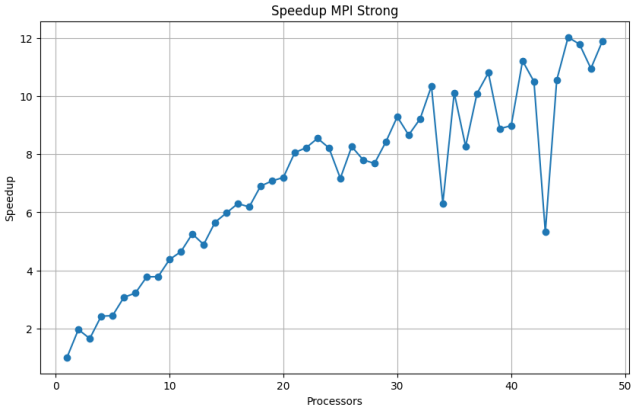
In this project we used two THIN nodes of the ORFEO cluster in the strong and weak scaling analysis with MPI and one node for the OMP scaling analysis. The THIN partition comprises 12 Intel nodes: two equipped with Xeon Gold 6154 and 10 equipped with Xeon Gold 6126 CPUs. The primary distinction in the hardware lies in the available random access memory (RAM). Each node contains 24 cores, resulting in a total of 48 cores used.

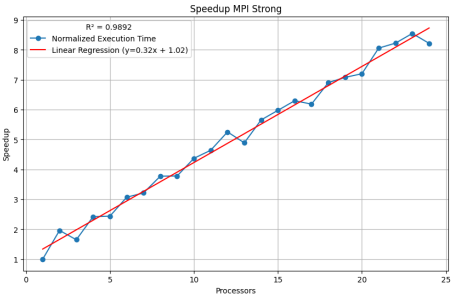
3 Results

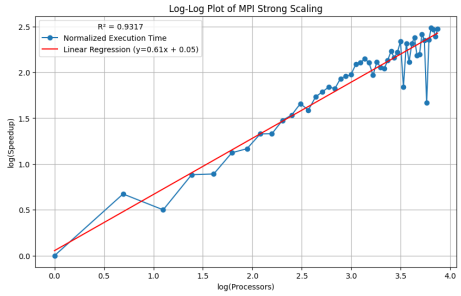
3.1 MPI Strong Scaling



As evident from the graph, the execution time decreases as the number of processors increases. Additionally, it is noticeable that after an initial phase of sharp decline, there is a stabilization, and the decrease almost halts.



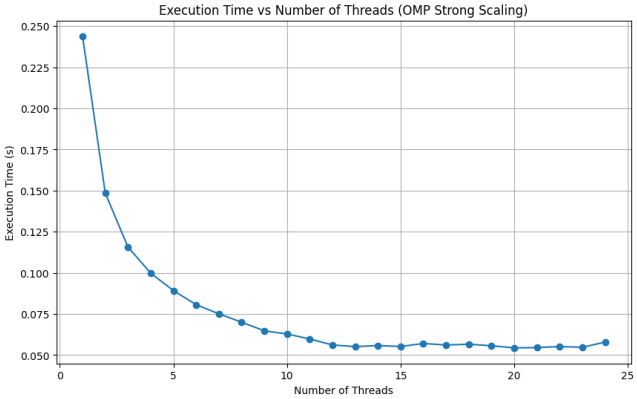




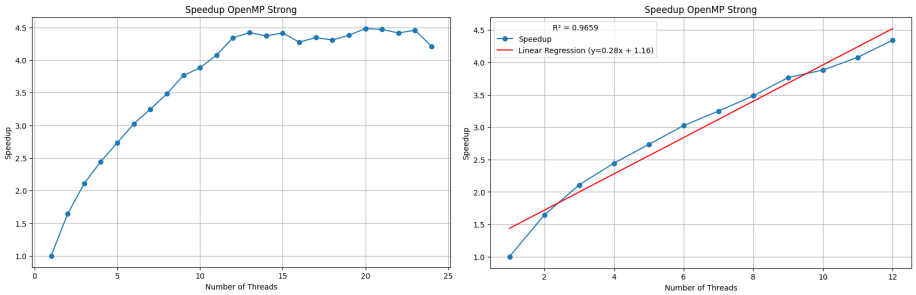
To further analyze the results, speedup and efficiency were calculated. The speedup exhibits a behavior somewhat different from the ideal one, as it shows a logarithmic curvature.

The speedup graph indicates a logarithmic decrease upon exhausting the cores of the first node (24 cores), which may be attributed to transitioning from one THIN node to another. When considering the speedup solely utilizing one node, it is evident that the trend is linear, and the developed model indicates a discrepancy of approximately 68% from the ideal case. However, to analyze the total data, a logarithmic model has been developed, as the linear model fails to fully account for the data. The logarithmic model, with an R^2 value of 0.93, provides a better description of the data.

3.2 OMP Strong Scaling

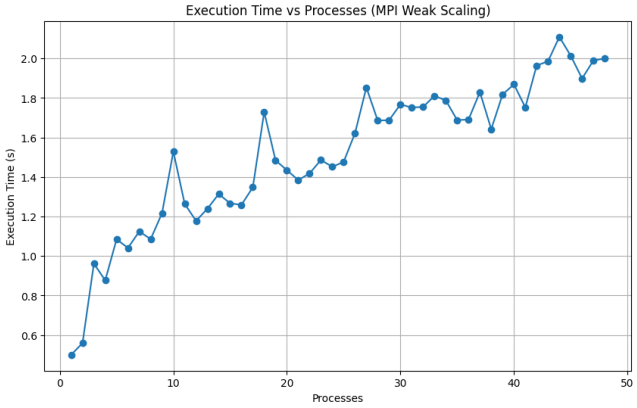


As evident from the graph, the execution time decreases as the number of threads increases.

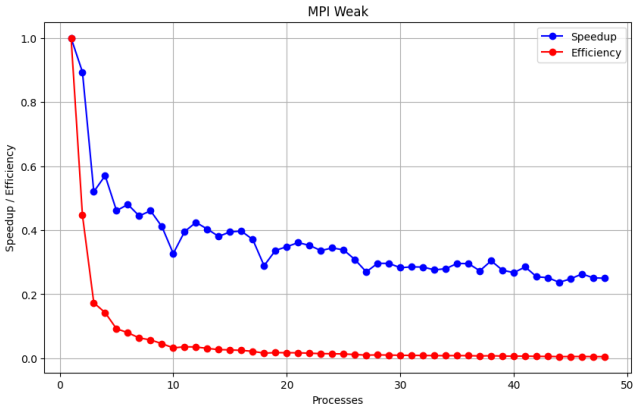


For the strong scaling analysis with OMP, a single THIN node was employed. In this scenario as well, a similar behavior to what was observed with strong scaling using MPI was noted, albeit with a dip occurring when transitioning between sockets (12 threads). When considering the speedup with the initial 12 threads, a linear trend is observed, and a linear model was also developed accordingly. The results indicate that this speedup deviates by approximately 72% from the ideal model.

3.3 MPI Weak Scaling

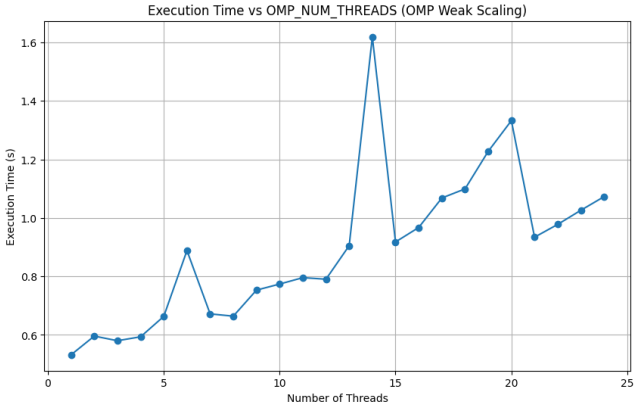


As evident from the graph, the execution time increases linearly with the growth of both the number of threads and the problem size. In the ideal scenario, the execution time should remain constant.

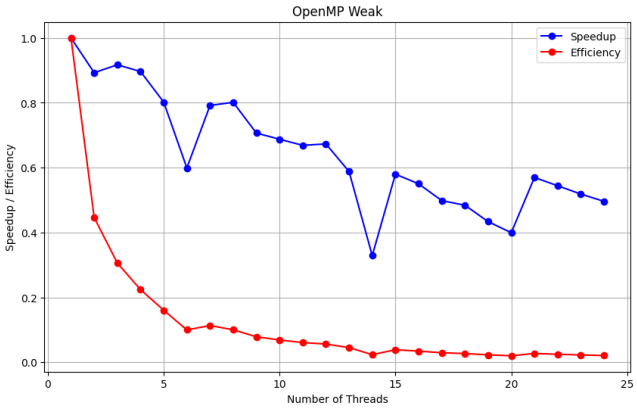


According to Gustafson's law, the speedup should increase linearly with the number of processes and the problem size. However, in our experiment, it decreases until stabilizing around 0.25, forming a plateau. Furthermore, efficiency decreases drastically from the outset, stabilizing below 1%. This unexpected behavior may be attributed to communication overhead or poor workload distribution. It is possible that some units complete their tasks before others and remain idle while waiting for the others to finish.

3.4 OMP Weak Scaling



Once again, we observe a similar trend to the previous one. However, this time we notice a "jump" when transitioning from one socket to another. Overall, a linear trend is observed.



Unlike the case with MPI, the weak scaling speedup with OMP exhibits a linear decline, potentially stabilizing around 0.5. Conversely, the efficiency drops below 1%, mirroring the previous case.

4 Conclusion

In conclusion, our investigation into parallel Mandelbrot set generation using MPI and OpenMP highlights the complexities of communication patterns, hardware configurations, and algorithm choices within OpenMPI. While our experiments revealed promising trends, such as linear scaling in some cases, unex-

pected behaviors, like efficiency drops, underscore the need for further optimization and refinement in parallel implementations.