High Performance Computing - Exercise 2

Alessandro Minutolo

1 Introduction

The aim of the project is to implement the mandelbrot set in a hybrid MPI-OpenMP code and then test the scaling performance for each MPI and OpenMP. The Mandelbrot set is generated on the complex plane C by iterating the complex funcion $f_c(z)$ 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 M is defined as the set of complex points c for which the aboce sequence is bounded. It may be proved that once on element i 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 M is the following:

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

where I_{max} is a parameter that sets the maximum number of iteration after which you consider the point c to belong to M. Numerically, leveraging the fact that $C \quad R^2$, the Mandelbrot set can be generated in a 2D grid and visualized as in image (Figure 1), where to discriminate whether a pixel belongs to the set or not, the pixel is substituted to the parameter c of the sequence $f_c^{n+1}(0) := z_{n+1} = z_n^2 + c$: if the sequence remains bounded for I_{max} iterations the pixel belongs to the set, otherwise not. Since each point of the set can be computed indipendently from the others, this problem is very parallelizable, there is no need of coordination of the workload among multiple processes.

2 Computational Environment

The tests have benn conducted on 2 EPYC nodes of the ORFEO cluster, which comprehend a total of 256 cores distributed on 4 sockets of 64 cores each.

3 Implementation

The implementation code combines **MPI** (Message Passing Interface) and **OpenMP** (Open Multi-Processing) to parallelize the computation of the Mandelbrot set. Below is a detailed explanation of how these two technologies are utilized together:

3.1 MPI: Distributed Memory Parallelism

The program initializes MPI with `MPI_Init_thread`, requesting a threading support level of `MPI_THREAD_FUNNELED`. This ensures that only the main thread of each process interacts with MPI, while OpenMP threads handle local computations.The total number of processes (`world_size`) and the rank of each process (`world_rank`) are retrieved, these values are used to divide the workload. The Mandelbrot image is divided into rows, which are distributed among the processes. Each process computes a contiguous block of rows based on its rank. The number of rows assigned to each process is calculated to ensure a balanced workload. Any leftover rows are distributed among the first few processes. Each process computes the pixel values for its assigned rows and stores them in a local buffer (`part_buffer`). Once each process has completed its portion, the results are gathered back into the root process using

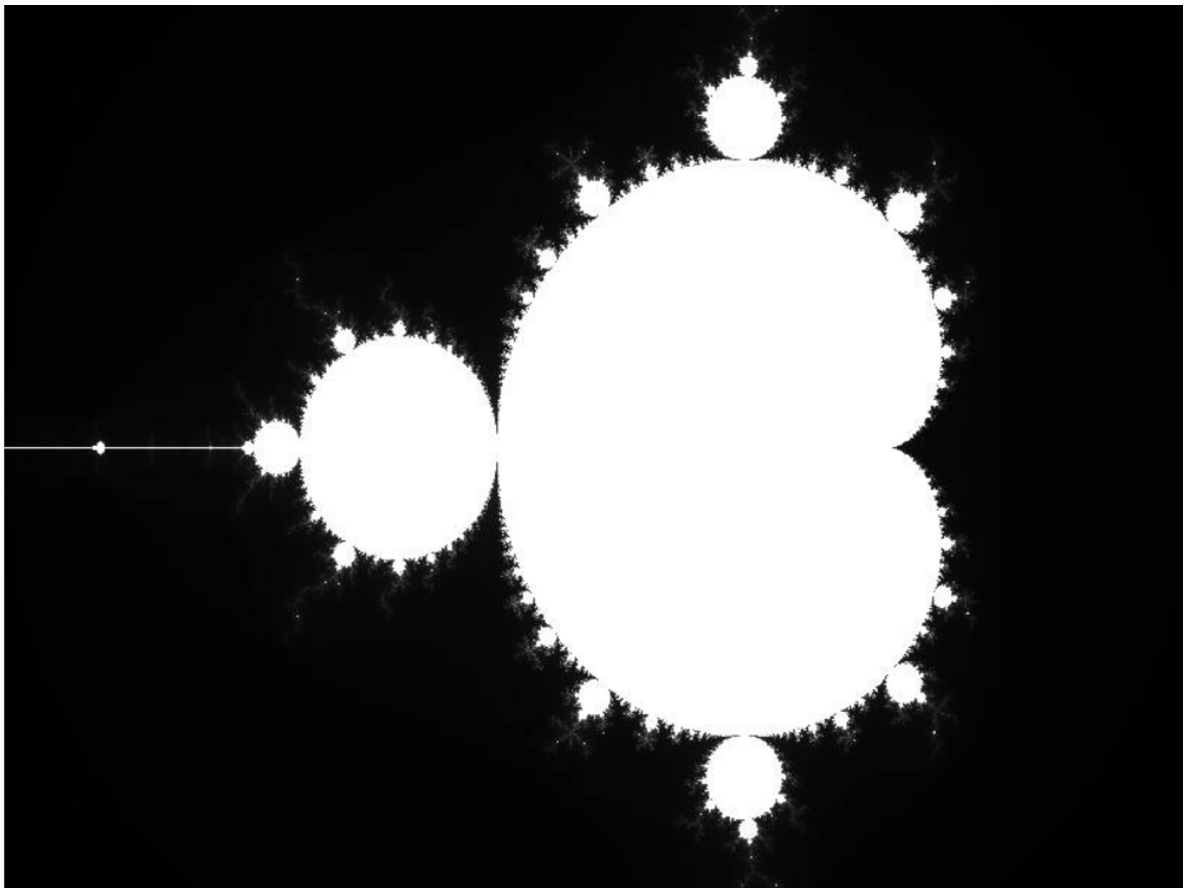

Figure 1: Mandelbrot set

`MPI_Gatherv`. The root process prepares by setting up arrays for `recvcounts` (number of elements contributed by each process) and `displs` (displacement offsets in the final image buffer). This ensures that the gathered data is placed correctly into the final image buffer. The root process writes the final image to a file in the PGM format. A `MPI_Barrier` ensures all processes have completed their computations before gathering results preventing partial data collection.

3.2 OpenMP: Shared Memory Parallelism

The **OpenMP** portion parallelizes the computation within each process. Each process computes its assigned rows using multiple threads. The number of threads is specified via a command-line argument and set with `omp_set_num_threads`. The `#pragma omp parallel for schedule(dynamic)` directive parallelizes the computation of rows. Rows are dynamically assigned to threads to balance the workload, as some rows in the Mandelbrot set may take longer to compute. Each thread works on its part of the local buffer (`part_buffer`), so no synchronization is needed within the OpenMP region. The use of dynamic scheduling in OpenMP ensures efficient utilization of resources, minimizing idle time for threads.

Together, MPI and OpenMP leverage both distributed and shared memory architectures: **MPI** distributes large chunks of the workload (rows of the image) across processes, enabling scalability across the cluster. **OpenMP** divides the workload further within each process, utilizing multiple threads on each node to speed up computation.

4 Scaling

In order to analyze the scalability of the code, two main tests have been conducted:

- **Strong scaling:** it examines how the execution time of a program decreases as the number of processing elements increases, given a fixed problem size
- **Weak scaling:** it evaluates how well a program handles an increasing workload while maintaining a constant workload per processing element.

For each of the two we can identify the OpenMP scaling, hence setting the number of MPI cores to 1, and the MPI scaling, hence setting the number of OpenMP threads to 1.

4.1 Strong Scaling

The primary metric for strong scaling is **speedup**, defined as the ratio of the execution time on a single processing element (T_1) to the execution on P processing elements(T_P):

$$Speedup = \frac{T_1}{T_P}$$

An ideal speedup would be linear, but this is rarely achieved due to overheads.

Efficiency is another critical metric, expressing how effectively the added processing elements are utilized. It is defined as:

$$Efficiency = \frac{Speedup}{P} = \frac{T_1}{P \cdot T_P}$$

As we see in Figure 2, the speedup for the OpenMP strong scaling initially increases nearly linearly as the number of threads increases, demonstrating good parallel performance at lower thread counts. As the number of threads exceeds a certain point (64 threads), the speedup begins to saturate and diverges significantly from the ideal linear scaling (shown as the diagonal line). This behavior indicates that the scaling is performing well within the same socket, when it involves the second one (from thread 64 to 127) the increasing number of threads are not affecting the performance. This phenomenon reflects also the efficiency that drops as soon as the number of threads reach 64. As regards MPI strong scaling, The speedup curve increases as the number of cores grows, but it deviates from the ideal linear scaling much earlier compared to the OMP case. It may mean both that MPI code management is not the best and that the more core we involve the more communication overhead there is. This behavior is also reflected in the efficiency analysis, which we see remaining flat at around 40%. The analysis of

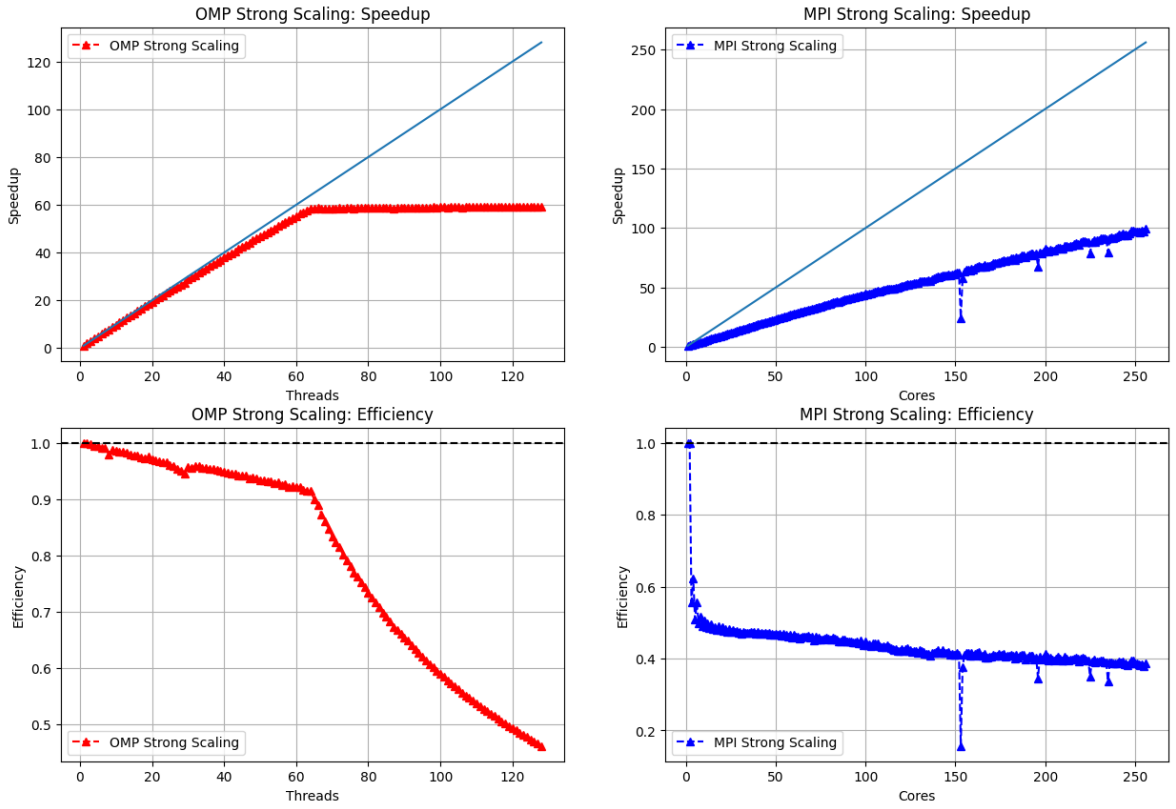

Figure 2: Speedup and efficiency for strong scaling

strong scaling often references Amdahl's Law, which describes the theoretical limit of speedup based on the fraction of the code that is parallelizable(f):

$$Speedup = \frac{1}{(1-f) + \frac{f}{P}}$$

Amdahl's Law highlights the diminishing returns of adding more processing elements due to the presence of serial portions of the program. It is useful for identifying bottlenecks and areas where optimization can maximize parallel performance.

The analysis shows that the parallel fraction of the program is remarkably high for both implementations, indicating that approximately 99% of the workload is parallelizable. This demonstrates the potential improvement of the implementation.

4.2 Weak Scaling

The key metric here is **scaled speedup**, defined as:

$$\text{Scaled Speedup} = P \cdot \frac{T_1}{T_P}$$

Efficiency in weak scaling reflects the program's ability to maintain performance as the workload grows:

$$Efficiency = \frac{\text{Scaled Speedup}}{P} = \frac{T_1}{T_P}$$

as we see in Figure 4, the observed speedup for the OMP implementation follows the ideal linear scaling very closely. This indicates that as the problem size scales proportionally to the number of threads, the workload is efficiently distributed across the available threads. The close match with the ideal scaling suggests that the OMP implementation is well-suited for weak scaling. This behavior

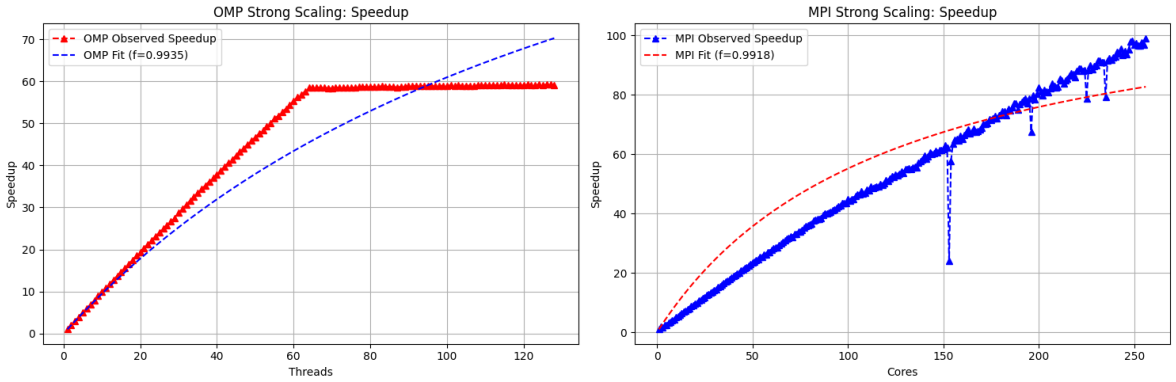

Figure 3: Amdahl's law

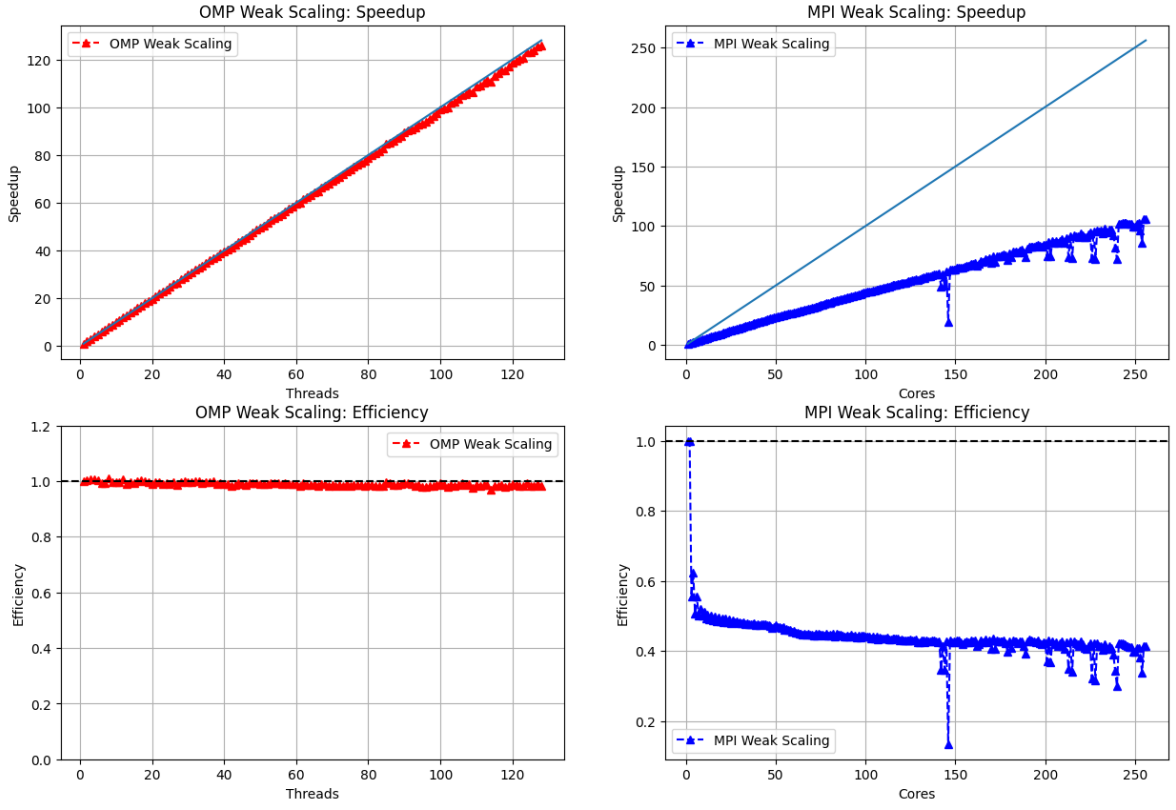

Figure 4: Speedup and efficiency for weak scaling

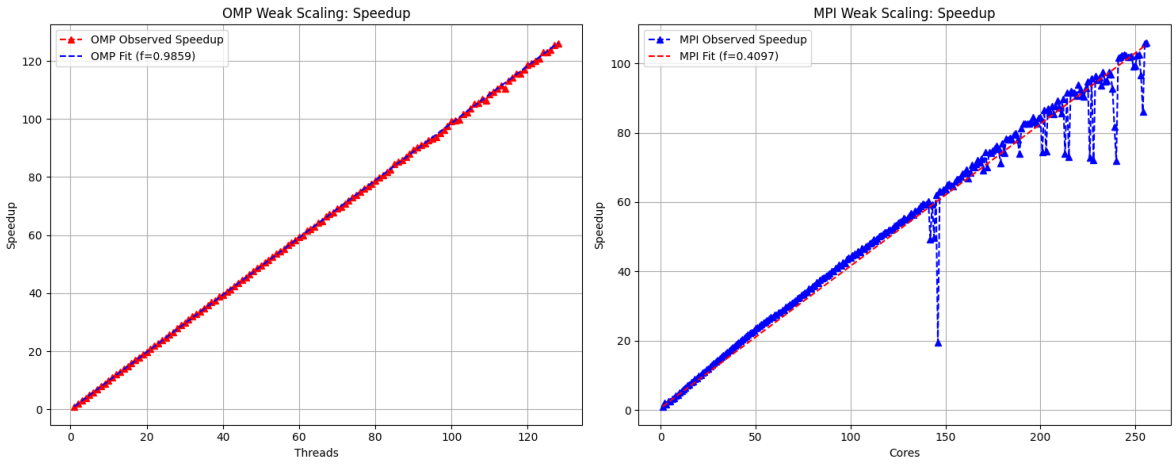

Figure 5: Gustafson's Law

is also reflected by the efficiency which remains very close to 100%. As regards for MPI, speedup initially grows but deviates significantly from the ideal linear scaling. This deviation highlights the increasing cost of communication and synchronization among distributed processes as the number of cores grows. Weak scaling performance is often analyzed in the context of Gustafson's Law, which states that the total workload is scalable with the number of processing elements, allowing for linear speedup by increasing the parallel portion of the program:

$$Speedup = P - (1 - f) \cdot (P - 1)$$

As we can see in Figure 5, for the OMP weak scaling, The value of 98.59% suggests that almost all of the workload is parallelizable, with only a minimal serial portion. This extremely high parallel fraction explains why the observed scaled speedup scales so well with the number of threads. The much lower value of $f \approx 0.41$ for MPI indicates a significant serial portion in the workload or high communication overhead, which limits scalability. Optimizing the MPI implementation to reduce these inefficiencies could lead to substantial performance gains.

5 Conclusion

From the results, we can conclude that the OMP implementation performs very well especially in weak scaling scenarios. The near-ideal speedup and efficiency observed suggest that the current implementation is well-optimized for shared memory architectures. The use of dynamic scheduling appears to play a key role in balancing the workload across threads, allowing the system to handle variations in computational demand effectively. This adaptability likely explains the minimal overhead and excellent scaling behavior observed in the OMP case, even as the number of threads increases significantly. On the other hand, the MPI implementation presents more significant challenges. While communication overhead is an expected limitation in distributed memory environments, a notable bottleneck arises from the static workload distribution strategy currently employed. The division of the workload by rows results in an uneven distribution of computational tasks among processes. This imbalance becomes particularly problematic when working with the Mandelbrot set, as the number of iterations required to compute points varies dramatically across the set. Points that require a higher number of iterations lead to uneven workloads, leaving some processes idle while others continue computing. Addressing this issue would require implementing a more dynamic or adaptive workload distribution strategy, potentially based on prior knowledge of the problem domain or real-time monitoring of process workloads. In summary, while we are satisfied with the performance of the OMP implementation, improvements to the MPI implementation are necessary to achieve better scalability and efficiency. Reducing communication overhead and implementing a more balanced workload distribution strategy would be key steps toward optimizing the MPI implementation for large-scale parallelism.