High Performance Computing
Exercise 2

Luis Fernando Palacios Flores *

University of Trieste

1 Introduction

The Mandelbrot set is the collection of complex numbers c for which the function defined by equation (1) remains bounded when iterated, starting at $z_0 = 0$. The bounding condition is $|z_n| \leq 2$, for $n \geq 0$. Computationally, the Mandelbrot set is generated in a 2D grid and can be visualized as an image as shown in Figure 1. Each pixel in an $r \times s$ image is formed by computing equation (1) up to a maximum number of iterations $I_{\max}$. If the sequence remains bounded within the maximum number of iterations, the corresponding complex number belongs to the Mandelbrot set; otherwise, it does not.

$$z_{n+1} = z_n^2 + c \tag{1}$$

Since the computation of each point in the Mandelbrot set is independent, it is easy to distribute the workload among multiple processors or cores without the need for coordination. Therefore, this problem is embarrassingly parallel, allowing for efficient and scalable parallelization.

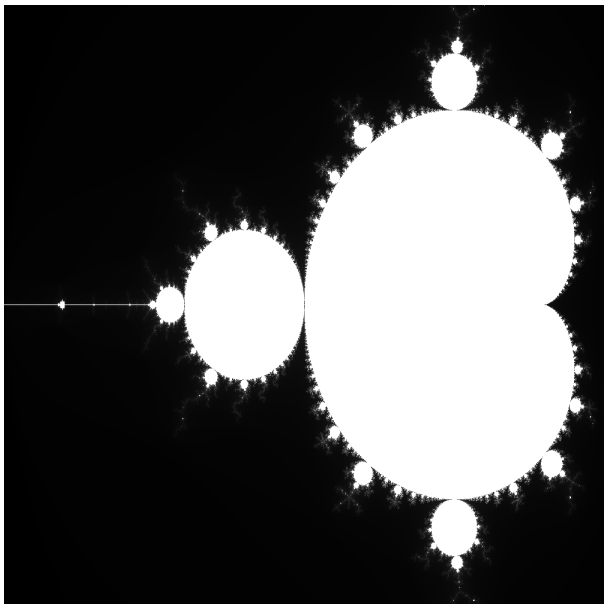

Figure 1: Mandelbrot set in the $[-2, 0.5] \times [-1, 1]$ region of the complex plane computed in parallel with MPI and OpenMP with a maximum number of iterations of 65536.

*luisfernando.palaciosflores@studenti.units.it

This project explores the integration of Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) for parallel computing in the Mandelbrot set calculation. The goal is to efficiently compute the Mandelbrot set within a specified region of the complex plane, leveraging MPI for distributed memory parallelism and OpenMP for shared memory parallelism.

Additionally, the implementation's speed-up is tested to provide a reliable estimate for the optimal match between job size and the required amount of resources for a specific task.

2 Methodology

2.1 Distributed Memory (MPI) parallelization

To parallelize the computation of the Mandelbrot set using MPI, a partitioning scheme along the y-axis was adopted. The 2D grid is divided into as many blocks as the number of processors over which the work is distributed. Each block (j) contains $r \times s_j$ pixels with $s_j \leq s$ and $j \leq N$, for N processors. Although the partitioning scheme aims to distribute the workload evenly among processors, Figure 1 illustrates some load imbalance, particularly for processors handling the extreme blocks of the grid.

In this project, the code to compute the Mandelbrot set was implemented using C. The region of the complex plane where the computations are done is $[-2, 0.5] \times [-1, 1]$. Since this is an embarrassingly parallel problem each processor performs computations in an array of the corresponding size. At the end of the computations, the distributed results of each block are collected by the root process using the `MPI_Gatherv` collective operation.

It is noteworthy to mention that, for properly collecting the partial results, each process computed both the number of elements that would be sent and the respective displacement of elements in the receive array buffer of the root process. This calculation was carried out with two additional `MPI_Gatherv` calls by the root process for each of these arrays.

2.2 Shared Memory (OpenMP) parallelization

To manage the computations of each processor, OpenMP was employed to parallelize the loop that determines whether a given complex number within the designated subset or block of the complex plane region belongs to the Mandelbrot set or not. The threads spawned by the processes iterate equation (1) for each complex number in the corresponding block and write the results in the respective array of their parent process. Due to the independent nature of computations for each complex number, a straightforward Round-Robin fashion distribution for the threads was chosen.

2.3 Strong Scaling

Measuring strong scaling involves timing a parallel application with an increasing number of computing units —processors for MPI and threads for OpenMP— while maintaining a constant problem size [1]. The objective is to assess how effectively a parallel application can leverage additional resources to expedite the solution of a fixed-size problem. Scalability is quantified using speedup, defined as the ratio of the time required for one processor (approximately equivalent to the sequential time) to execute the application, compared to the time required for P processors. This

relationship is expressed by equation (2). In an ideal scenario, the speedup would be directly proportional to the number of processors (P), resulting in an efficiency equal to 1, where efficiency is the ratio of speedup to the number of processors.

$$S(P) = \frac{t(1)}{t(P)} \tag{2}$$

In this project, the parallel application for computing the Mandelbrot set was executed on the ORFEO cluster, utilizing two EPYC nodes (epyc003 and epyc004). For MPI strong scaling, the problem size remained constant at 1000 rows and 1000 columns per image. All the cores in the nodes were utilized, and each process was mapped to a core using the mapping policy `--map-by core`, and only one OpenMP thread was employed.

On the other hand, for OpenMP scaling, the problem size was also fixed at 1000 rows and columns, with only one MPI process utilized. This configuration was chosen to comply with the instructions provided in the project assignment. Due to the use of only one MPI process, only the cores in one socket (of node epyc003) were employed. The MPI process was mapped to one `socket`, and the binding policy was set to `socket`. As per the available documentation [2–4], it is not possible to bind a process to an entire node. Hence, for OpenMP scaling, only one socket was utilized. The thread affinity was configured to `threads` for places and `close` for the binding policy.

2.4 Weak Scaling

To perform weak scaling, both the number of processors and the problem size are increased at the same rate so that the workload per processor remains constant. Following Gustafson's law, weak scaling measures how the execution time of a parallel application remains roughly constant as the number of processing elements increases in proportion to the problem size. The objective here is to evaluate the parallel application's efficiency in handling larger problems by seamlessly adding resources proportional to the problem size.

Here, the problem size of the parallel application is given by the product of the number of rows and columns ($r \times s$). The workload per processor was fixed to be constant so that the problem size is proportional to the number of processors used for the computations. Without loss of generality, the number of rows and columns was set to be the same so that their calculations were easier.

Using the `short int` data type to store the value of the iteration number of equation (1) for the corresponding complex number, the workload in both MPI and OpenMP scaling was maintained at 2MB per processor/thread. The mapping and binding policies mentioned above were retained, with the only difference being a reduced number of computing units.

3 Results and discussion

In Figure 1 the result of the execution of the parallel application is shown, successfully computing the Mandelbrot set using MPI and OpenMP. The program was faster with the increasing number of processors and threads as shown in Figure 2. For strong scaling the parallel application was more than 100 times faster increasing the number of processes and about 30 times faster by adding more threads, in the best scenario. The data was well described by a linear model in terms of the

inverse of the number of workers, with more than 90% explainability for both MPI and OpenMP scaling.

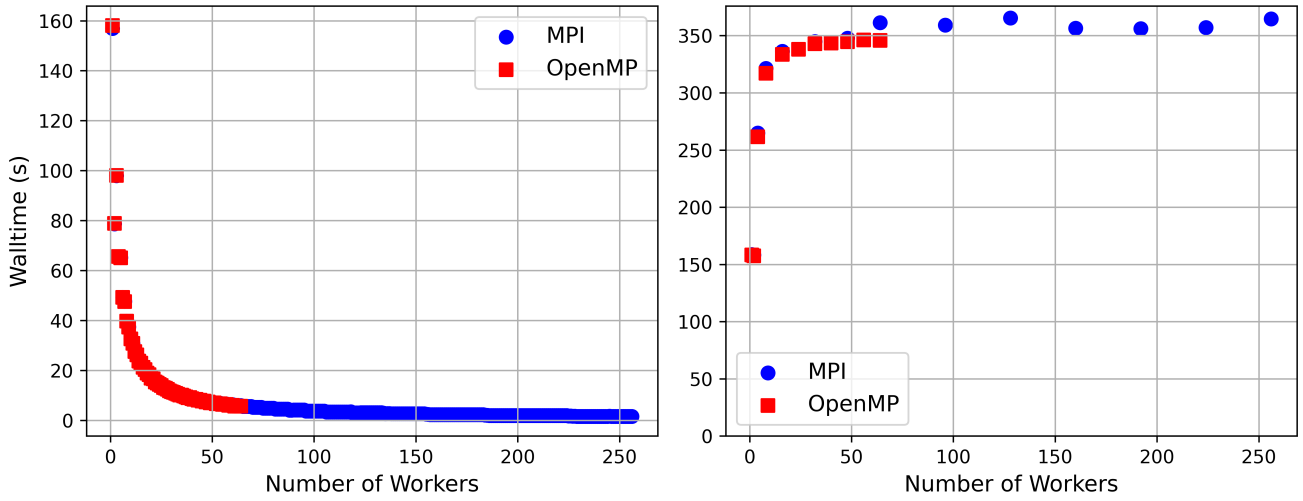

Figure 2: Walltime vs the number of processors (MPI scaling) and threads (OpenMP scaling). Left: Strong Scaling; Right: Weak Scaling.

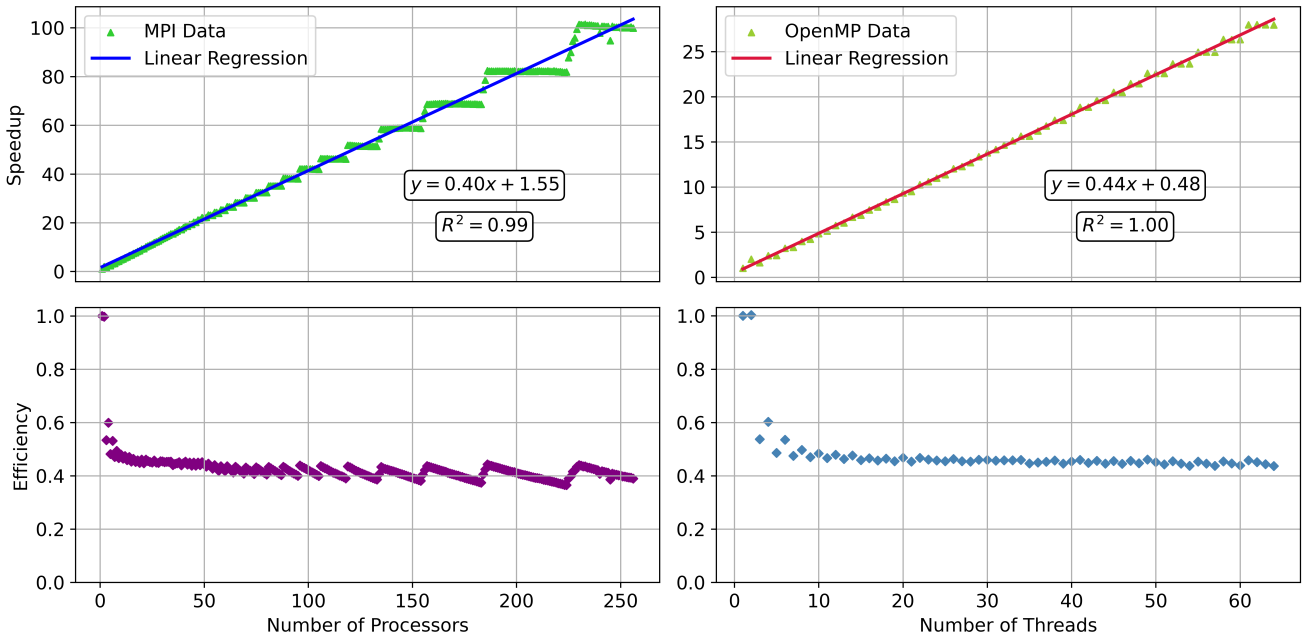

Figure 3: Speedup and efficiency vs the number of processors/threads in strong scaling. Left: MPI data; Right: OpenMP data.

In the case of weak scaling, Figure 2 illustrates that the execution time stabilizes around 360 seconds for MPI scaling and 344 for OpenMP scaling. The outcomes of strong scaling align with expectations, as the parallel execution time decreases inversely proportional to the number of

workers. However, in the case of weak scaling, where the parallel execution time is anticipated to remain approximately constant with an increase in the number of workers, the observed trend deviates from the expected pattern since there is an initial increase in the execution time instead of a decrease.

The strong scaling analysis revealed a linear behavior in speedup with the number of computing units, as evident in Figure 3. Both the MPI and the OpenMP data exhibited approximately a 60% and 56% deviation, respectively, from the ideal scenario, excluding constant terms. Figure 2 showcased similar performance for the parallel application using MPI and OpenMP alone. This similarity suggests that OpenMP's loop parallelization mirrors the partitioning strategy applied in this specific case.

For processes distributed in more than one socket, there are some plateaus in speedup and fluctuating efficiency for the MPI data. Amdahl's law predicts a stagnation of the speedup due to the serial portion of the parallel application but not different plateaus. Among the possible causes are the load imbalance discussed in Section 2.1, overhead in the creation and management of the arrays that store the information of each process, and cache-related issues. The benefits of parallel computation are diminished by these effects, leading to local stagnations in the walltime. In the case of shared memory, the speedup does not exhibit prolonged stagnation values. The efficiency of the MPI data fluctuates around 42% and 45% for the OpenMP data. This suggests that for optimal efficiency and effective use of resources, using only one socket or a reduced number of cores (especially above 10) might be sufficient. Enhancing efficiency may involve devising a better partitioning scheme for the complex plane region among processes and optimizing computations within their respective blocks, promising overall performance improvements for the parallel application.

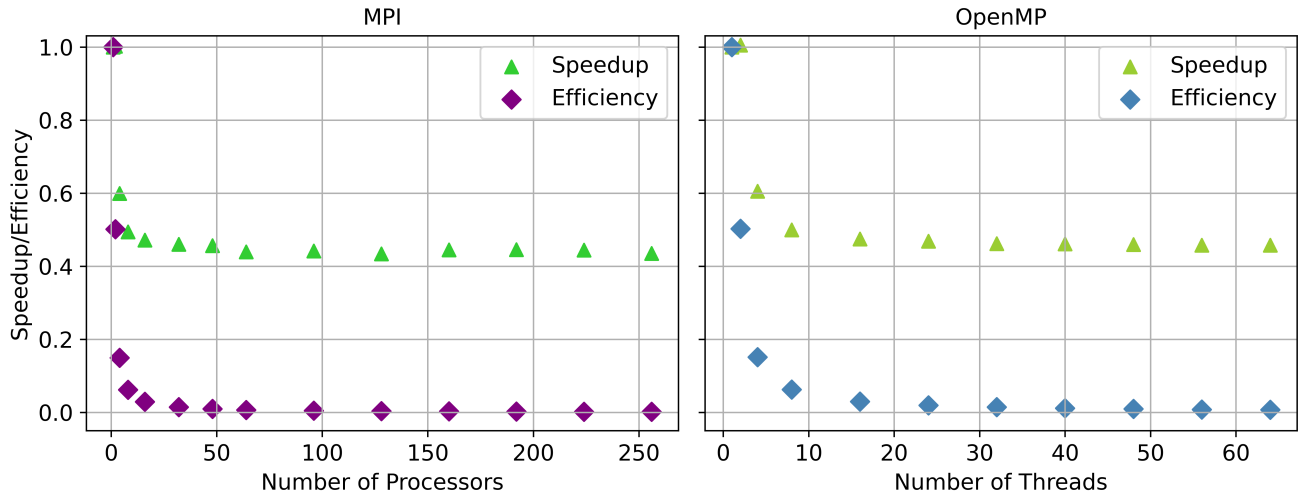

Figure 4: Speedup and efficiency vs the number of processors/threads in weak scaling. Left: MPI data; Right: OpenMP data.

As aforementioned, in the investigation of weak scaling for the parallel application, an unexpected observation emerged as the speedup exhibited a decline with the increasing problem size and the number of processes/threads, despite maintaining a constant workload per computing unit. Contrary to expectations set by Gustafson's law, the speedup failed to demonstrate a linear increase with the growing number of computing units, as illustrated in Figure 4. Instead, it rapidly

stabilized around a constant value, resulting in a speedup plateau at approximately 0.4. Moreover, the efficiency of weak scaling experienced a rapid drop to zero, with values dipping below 1% for more than 50 computing units in both cases. The observed deviation from Gustafson's law prompts further refinement of the parallel application's implementation. One potential avenue for improvement is the modification of the workload distribution, as discussed earlier. The expected load imbalance among the computing units might be leading to some units finishing earlier and idling while waiting for others to complete, contributing to the observed behavior.

Conducting hybrid scaling for shared memory with two EPYC nodes and 4 processors generally resulted in a smaller walltime compared to the pure MPI or OpenMP scaling shown in Figure 2. The mapping and binding policy remained consistent with the methodology described earlier. In hybrid strong scaling, lower walltime values were observed for all the number of threads across the two nodes. However, when the threads were distributed in more than one socket there was a fluctuation around the minimum time of around 1 second. As for the walltime in weak scaling, the trend mirrored Figure 4, albeit with lower values for threads in one socket. Surprisingly, for a larger number of threads, the walltime exhibited a linear behavior with the number of threads, deviating from the original constant trend associated with the increasing problem size and the number of computing threads. This deviation could be attributed to factors such as memory overhead and load imbalance, making it inefficient and more costly to spawn additional threads.

4 Conclusions

The computation of the Mandelbrot set using a parallel hybrid approach using MPI and OpenMP proved successful. In the strong scaling test, both the MPI and OpenMP data demonstrated an efficiency of over 40%, with the speedup well described linearly to the number of processes in distributed memory and threads in shared memory. This analysis suggested that for optimal efficiency and resource utilization, employing only one socket or a reduced number of cores, particularly above 10, might suffice for the described implementation. However, the study of weak scaling revealed an unexpected deviation from Gustafson's law. The speedup decreased, stabilizing at 0.4, while efficiency rapidly dropped to zero. The observed inefficiencies in strong scaling and the surprising weak scaling results could be attributed to the anticipated load imbalance inherent in the implemented methodology for parallelizing Mandelbrot set calculations. Further enhancements, including the development of a more effective workload partitioning scheme, are imperative to ensure improved performance of the parallel application and align the scaling results with expectations.

References

[1] Frank Nielsen. *Introduction to HPC with MPI for Data Science*. Springer, 2016.

[2] Binding/pinning - hpc wiki. https://hpc-wiki.info/hpc/Binding/Pinning, 2024. Accessed: 2024-03-05.

[3] mpirun(1) man page (version 3.0.6). https://www.open-mpi.org/doc/v3.0/man1/mpirun.1.php, 2024. Accessed: 2024-03-05.

[4] 17.1.2. mpirun / mpiexec — open mpi 5.0.x documentation. https://docs.open-mpi.org/en/v5.0.x/man-openmpi/man1/mpirun.1.html#label-schizo-ompi-bind-to, 2024. Accessed: 2024-03-05.