The Mandelbrot set
Hybrid MPI/OpenMP implementation

Alessandro Della Siega

May 2024

1 Introduction

The goal of this assignment is to compute and visualize the Mandelbrot set.

The properties of the Mandelbrot set make this problem an ideal candidate for parallel computing, as it is embarrassingly parallel. This characteristic allows each point in the complex plane to be computed independently of the others, making it a perfect candidate for a hybrid implementation using both the Message Passing Interface (MPI) and Open Multi-Processing (OpenMP).

To assess the performance of our implementation, we will evaluate both strong and weak scaling. Strong scaling involves measuring how the solution time varies with a fixed total problem size as we increase the number of processors. Weak scaling, on the other hand, involves measuring how the solution time changes as we increase both the number of processors and the total problem size proportionally.

For this evaluation, we will consider the following scenarios.

1. **MPI scaling**: In this scenario, we will fix the number of OpenMP threads to one per MPI process and increase the number of MPI processes. This approach allows us to evaluate how well the computation scales when distributing tasks across multiple processes, each running on separate processors or nodes.

2. **OpenMP scaling**: Here, we will fix the number of MPI processes to one and increase the number of OpenMP threads. This scenario will help us understand how the computation scales when leveraging multiple threads within a single process, taking advantage of multi-core processors.

By conducting these scaling experiments, we aim to gain insights into the efficiency and performance of our hybrid MPI and OpenMP implementation on the ORFEO cluster. This analysis will not only help us optimize the computation of the Mandelbrot set but also provide valuable information on how to effectively utilize high-performance computing resources for similar parallelizable problems.

1.1 Architecture of computational resources

In this project, we will implement a hybrid MPI and OpenMP solution to compute the Mandelbrot set. By leveraging the combined strengths of MPI

and OpenMP, we can efficiently distribute the computation workload across multiple nodes and cores within the ORFEO cluster, specifically utilizing the EPYC partition. This partition comprises 8 nodes, each equipped with two AMD EPYC 7H12 CPUs. Each CPU contains two sockets with 64 cores each, providing substantial computational power. The nodes are interconnected via an Infinity Fabric network with a theoretical bandwidth of 96 Gb/s, facilitating high-speed data transfer between nodes. Each processor is organized into 8 Core Complex Dies (CCDs), with each CCD containing 2 Core Complexes (CCXs), and each CCX comprising 4 cores with 16 MB of L3 cache.

2 Implementation

2.1 Encoding

The encoding of the problem is straightforward. An image of the Mandelbrot set is a two-dimensional array of pixels. Each pixel corresponds to a point in the complex plane. However, the image is stored as a one-dimensional array of pixels. Each pixel is stored as a `unsigned char`, its maximum value can be 255. This means that the maximum iteration that can be reached on a point of the Mandelbrot is 255. A scale of gray is used to represent the points of the Mandelbrot set. For each pixel we will assing a value based onn wether the corresponding point belongs to the mandelbrot set:

- if the point belongs to the Mandelbrot set, the pixel is assigned the value 0 and the pixel will be black;
- if the point does not belong to the Mandelbrot set, the pixel is assigned the iteration count at which the point exited the circle of radius 2. The pixel will be assigned a value between 1 and 255 and the pixel will be colored according to the gray scale.

2.2 C implementation

Let us present a brief summary of C implementation:

- `main.c`, it contains the main function in which MPI is initialized, the amount of work is distributed among the MPI processes, and the computation is performed. In the end, the computation is gathered and the image is saved. The main function also takes care of the timing of the computation;
- `mandelbrot.c`, in this file we define two functions. The first is the iterative quadratic map that characterizes the points of the Mandelbrot set. The latter is the function whose task is to evaluate, within an OpenMP parallel region, the Mandelbrot set on a given array of pixels;
- `image_utils.c`, in this file we define the function that saves the image in the desired PGM format;
- `timing_utils.c`, here the function that saves the timing results on a `.csv` file.

2.3 MPI parallelization

In order to distribute the work among the MPI processes we decided to adopt a sequential fashion.

Suppose to have P processes and $N = n_x \times n_y$ pixels to compute. The root process divides the one-dimensional array in M blocks of approximately N/M pixels each. Obviously, the last block may contain less pixels if M is not divisible by N since the remainder is distributed starting from the first process, with respect to the rank. The first process is assigned the first block of pixels, the second process the second block, and so on. The last process is assigned the last block of pixels. The gathering of the results is done by the root process, which calls `MPI_Gatherv` to collect the results from all the other processes and saves the image.

2.4 OpenMP parallelization

The OpenMP parallelization is performed in `compute_mandelbrot_set` in the file `mandelbrot.c`. The parallel region is defined at the beginning of the function and the parallel for directive is used to parallelize the loop that iterates over the pixels of the image, i.e. the elements of the one-dimensional array. The scheduling policy is set to `dynamic`: OpenMP assigns one iteration to each thread. When the thread finishes, it will be assigned the next iteration that has not been executed yet.

2.5 Experimental setup

For what concerns the timing of the computation, we will use the `MPI_Wtime` function to measure the time spent in the computation of the Mandelbrot set. In particular, For each run, we start the timer after the MPI initialization and stop it right after the gathering of the results. In our measurments we do not include the time required to save the image on a file since we are not interested in the I/O performance.

Now, for what concerns the scaling analysis, we will consider the following setup. For both the MPI and OpenMP scaling, we will consider the strong scaling and the weak scaling. The strong scaling consists in fixing the dimension n_x, n_y of the Mandelbrot image and increasing the number P/T of processes/threads. While, the weak scaling consists in fixing the amount

of work per process/thread and increasing the number of processes/threads. Let us define the following setup:

- **MPI strong scaling**: we fix $T = 1$,

$$n_x = 4096$$

$$n_y = 4096$$

  and $P = 1, 2, 4, 8, 16, 32, 64, 80, 96, 112, 128$.

- **MPI weak scaling**: we fix $T = 1$,

$$n_x = 1024 \times \texttt{round}\{\sqrt{P}\}$$

$$n_y = 1024 \times \texttt{round}\{\sqrt{P}\}$$

  for $P = 1, 2, 4, 8, 16, 32, 64, 80, 96, 112, 128$;

- **OpenMP strong scaling**: we fix $P = 1$,

$$n_x = 4096$$

$$n_y = 4096$$

  for $T = 2, 4, 6, 8, ...62, 64$;

- **OpenMP weak scaling**: we fix $P = 1$

$$n_x = 1024 \times \texttt{round}\{\sqrt{T}\}$$

$$n_y = 1024 \times \texttt{round}\{\sqrt{T}\}$$

  for $T = 2, 4, 6, 8, ...62, 64$.

.

For the MPI scaling, we set `--map-by core` since we want that, after that a node is selected, each process spawns in its cores.

For the OpenMP scaling, we used the default option `--map-by socket` since we are using a single process and a single socket which has 64 cores. Lastly,we set `OMP_PLACES=cores` to minimize resource contention: assigning threads to distinct cores helps to minimize contention for CPU resources, ensuring that each thread has dedicated compute power

3 Results

In this section we will discuss the results obtained from the scaling analysis.

3.1 MPI scaling

The results obrained The MPI strong scaling seems to scale well up to 128 processes as shown in Figure 1. This is confired by the speedup which is perfectly linear. The discrepancy between the ideal and the actual strong scaling is low. The nature of the Mandelbrot set computation allow us to compare our results to the ideal scaling. By ideal scaling we mean the Ahmdal's law in which we assume that the entire work is parallelizable and there is no communication overhead.

The MPI weak scaling, after 10 processes reaches a plateau. This means that the speedup is constant as the number of processes increases as shown in Figure 4. This is reasonable since the amount of work per process is constant and also the time is constant as the number of processes increases.

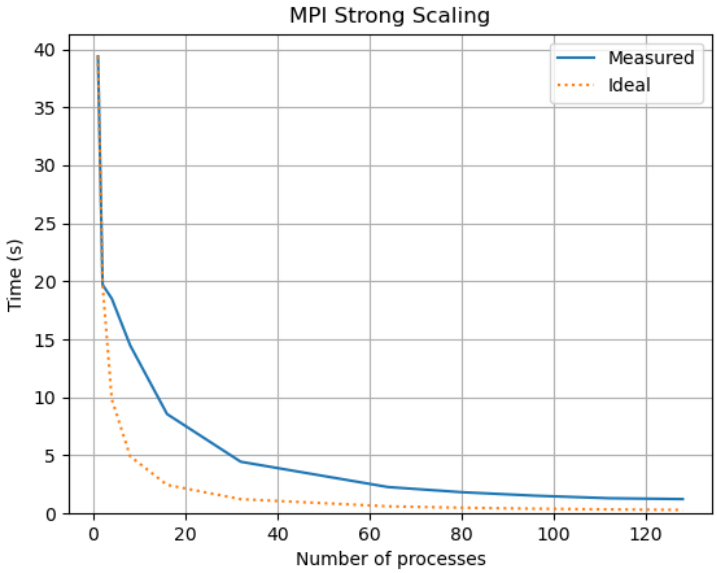

Figure 1: MPI strong scaling

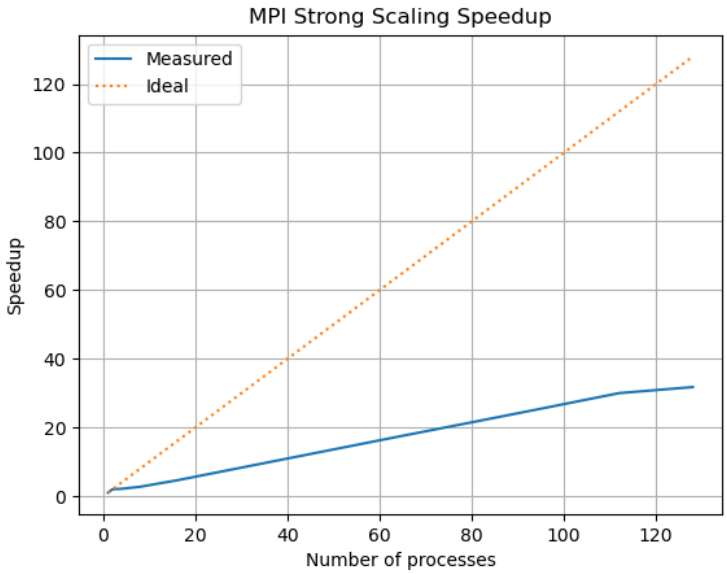

Figure 2: MPI strong scaling speedup

3.2 OpenMP scaling

The OpenMP strong scaling seems to scale with some more overhead than the MPI strong scaling. However, there is a substantial discrepancy between the ideal and the actual strong scaling as shown in Figure 5. After 20 threads the speedup decreases and becomes constant. This means that the overhead is significantly larger than the computation time.

For what concerns the OpenMP weak scaling, the speedup becames constant as the number of threads increases as shown in Figure 8. We can notice that in this case the scaling assumes a step-like shape. This is due to the fact that we decided to round the square root of the number of threads to the nearest integer to make the amount of work per thread appproximately harder.

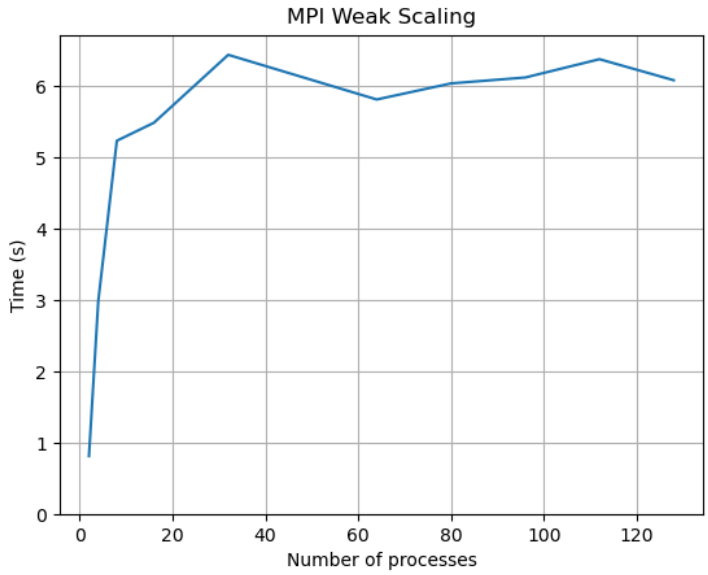

Figure 3: MPI weak scaling

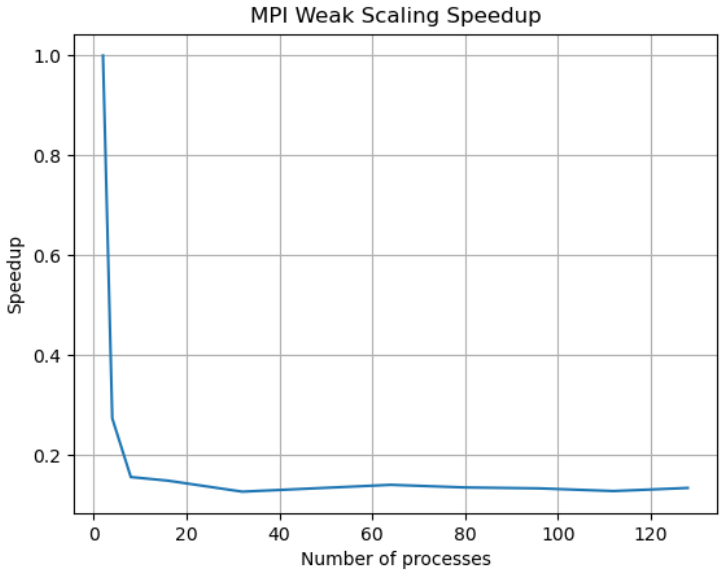

Figure 4: MPI weak scaling speedup

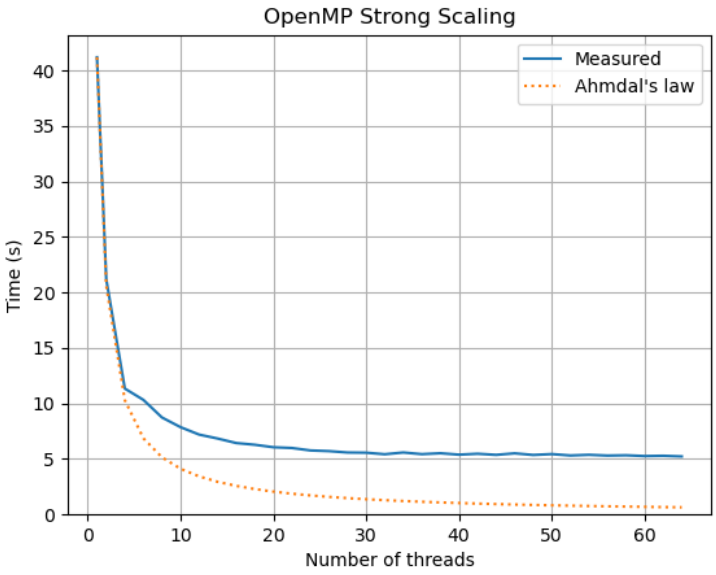

Figure 5: OpenMP strong scaling

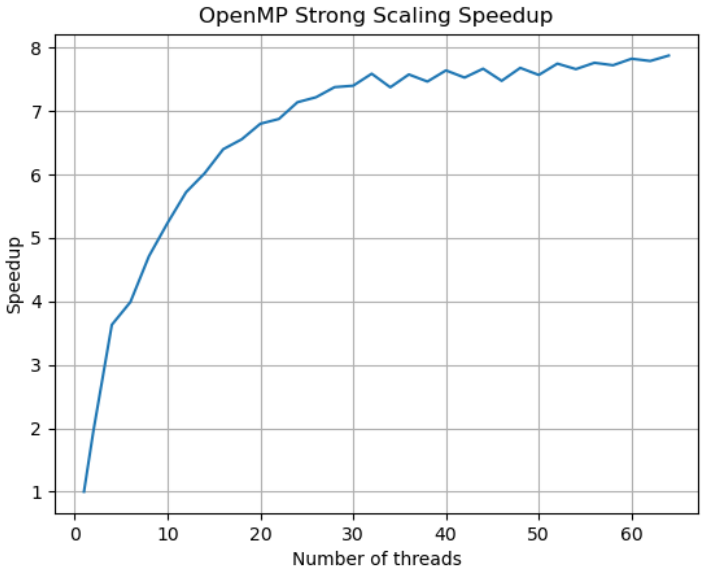

Figure 6: OpenMP strong scaling speedup

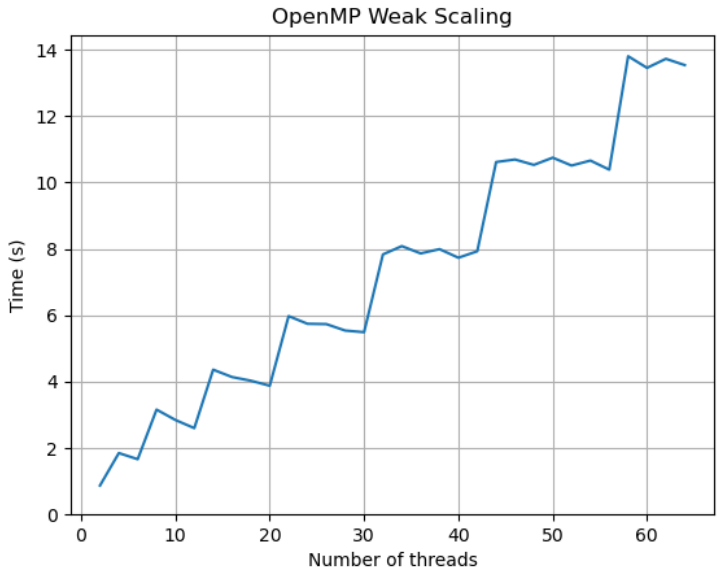

Figure 7: OpenMP weak scaling

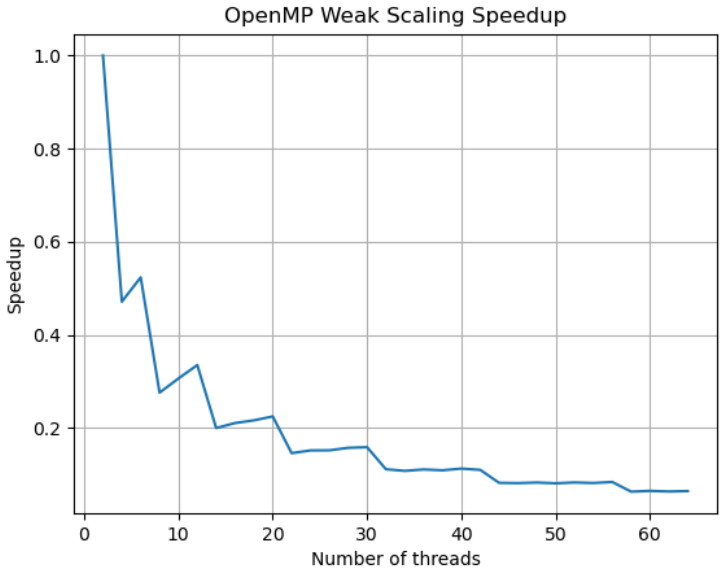

Figure 8: OpenMP weak scaling speedup

4 Conclusion

Within this assignment I explored a parallel approach for computing the Mandelbrot set, developing an hybrid MPI+OpenMP code. The designed solution seems to scale as expected and the results are consistent with the theoretical expectations. However there are some important considerations to be made:

- The OpenMP strong scaling shows a substantial discrepancy between the ideal and the actual strong scaling. This can be explained by the following reasons:

  1. the workload is unbalanced among the threads. Some regions of the Mandelbrot set require more iterations to be computed than others. This is due to the fact that the Mandelbrot set is a fractal and the convergence rate is different for each point;
  2. the overhead of the OpenMP parallelization is larger than the computation time. This is due to the fact that the OpenMP parallelization is not efficient for small workloads. The overhead of the parallelization is larger than the computation time. Probably, this implemenation of considered not large enough images to be computed in parallel.
  3. the OpenMP parallelization performance could suffer from false sharing. In fact, we used a global array of pixels to store the results of the computation. False sharing occurs when two or more threads write to different variables that reside on the same cache line. This can lead to cache coherence traffic between the cores, which can degrade the performance.

- The MPI scaling seems to scale as expected. In particular, the use of collective operations, such as `MPI_Gatherv`, allows to gather the results in a more efficient way than using point-to-point communication.

Some possible improvements to the code could be:

- subdividing the image in a more balanced way among the threads. This could be done by using a more sophisticated partitioning algorithm. For example, we could use a space-filling curve to divide the image in a more balanced way. However, this could make the implemenation very complex;

- exploiting some symmetry properties of the Mandelbrot set. The Mandelbrot set is symmetric with respect to the real axis. This means that we could compute only the upper half of the image and then mirror the results to obtain the lower half. This could reduce the amount of work to be done;