High Performance Computing assignment
Exercise 2C

Piero Zappi

April 2024

1 Introduction

The Mandelbrot set $\mathcal{M}$ is a two-dimensional set which is generated on the complex plane $\mathbb{C}$ by iterating the complex function $f_c(z) = z^2 + c$; graphically, it is represented as a famous fractal shape, which exhibits self similarity. Specifically, the Mandelbrot set is defined as the set of the complex numbers c for which the function $f_c(z)$ does not diverge when iterated starting from the complex value $z = 0$; in other words, it comprises all the complex numbers c such that the sequence $z_0 = 0, z_1 = f_c(z_0), z_2 = f_c(z_1), \ldots$, is bounded in absolute value. It is possible to prove that once an element i of the sequence is farther than 2 from the origin the series is then unbounded; this property can be used to generate the set $\mathcal{M}$: given a portion[1] of the complex plane, the function $f_c(z)$ is iterated for each complex number c belonging to the portion, until the sequence diverges or a maximum number of iterations I_{max} is reached. Formally, the condition to determine whether a point c belongs to the set $\mathcal{M}$ is the following:

$$|z_n = f_c^n(0)| < 2 \quad \text{or} \quad n > I_{max} \tag{1}$$

The problem of generating the Mandelbrot set $\mathcal{M}$ is well suited for being computed efficiently in parallel, since each point c can be calculated independently of each other. This report introduces a distributed memory parallel implementation of an algorithm to compute $\mathcal{M}$; in particular, a `C` hybrid code has been developed, using both the *Message Passing Interface* (MPI) and the *OpenMP* (OMP) libraries to combine their functionalities for efficient parallel computation. Furthermore, this report will provide an in-depth analysis of the strong and weak scalings of the code, offering valuable insights into its efficiency and performance.

2 Implementation

This section of the report will briefly introduce the implementation of the algorithm; the entire code is available on the author's GitHub repository [1].

2.1 Algorithm

The developed code computes the Mandelbrot set $\mathcal{M}$ and produces as output an image file of the set, using the `.pgm` format. Given a region of the complex plane, the image of $\mathcal{M}$ is computed by iterating the function $f_c(z)$ for each point c of the region until the condition

[1] The Mandelbrot set $\mathcal{M}$ approximately occupies the circular region centered on $(-0.75, 0)$, with a radius of ≈ 2

(1) is met. Each point c corresponds to a pixel of the final image: if the sequence does not diverge, then the point c belongs to the Mandelbrot set and we therefore assign the value of 0 to its corresponding pixel; on the other hand, if the series diverges, we give to the pixel the value n of the iteration for which $|z_n(c)| > 2$, saturating to I_{max}. As previously anticipated, I_{max} is a parameter of the code that sets the maximum number of iterations after which a point is considered to belong to $\mathcal{M}$; both the accuracy of the computations and the computational cost increase with I_{max}.
The implemented code defines a matrix `M` of integers (`short int`), whose entries `[j][i]` store the final image's pixels' values, ranging from 0 to I_{max}. Once the matrix `M` has been computed, it can be easily converted to a `.pgm` file by passing it as input to the dedicated `write_pgm_image()` function.

2.2 Parallelization

As we already stated, each pixel of the matrix `M` can be computed independently from the others; the MPI and the OMP libraries have then been used and combined to exploit the highly parallelizable nature of this problem and provide an efficient parallel computation.

2.2.1 MPI

The main task of computing the matrix `M` is divided among MPI processes by assigning to each process a `local_rows` amount of rows to compute. In order to reduce the problem of load imbalance, the rows are distributed among the MPI tasks in a round robin fashion, with the first processes getting one more row than the others if the number of rows is not divisible by the number of processes. In fact, the Mandelbrot set's inner points are computationally more demanding than the outer ones, with the frontier being the most complex region to be resolved: by evenly subdividing the complex plane among the MPI tasks, assigning to each process a block of contiguous rows of the matrix `M`, we would have encountered severe imbalance problems.
The code has been implemented such that each process allocates only the memory required for the local part of the matrix `M` that it has to compute (denoted as `local_M`), ensuring to employ no more than the strictly necessary resources in order to improve the memory efficiency. Once all the MPI tasks finish their main computation, the results have to be gathered from all the processes to the master process, identified by its `rank=0`, which allocates the memory required to store the entire matrix `M`. To manage this, the `MPI_Gatherv()` collective operation is used, since the `local_rows` computed by each process can be different: in fact, some processes can occasionally be assigned one more row than others. Finally, the master process reorders correctly the rows of the whole matrix `M` before calling the `write_pgm_image()` function to produce the Mandelbrot set's image.

2.2.2 OMP

The computational work assigned to each MPI task is further subdivided among OMP threads, with each thread tasked with computing one of the `local_rows` rows assigned to the process at a time. To manage this, as it's possible to notice by looking at code (1), a parallel region is introduced using the `#pragma omp parallel for` directive to handle two nested loops. We employed the `dynamic` scheduling in order to minimize the load imbalance among the threads; in fact, the dynamic work assignment ensures that each thread receives a new row to compute as soon as it finishes the previous one, effectively balancing the computational load among the threads spawned by a process. This approach is particularly

effective when the computational load of each iteration of the loop is unpredictable as in this case; by assigning each row to the first OMP thread that requests a workload, the workload distribution remains balanced among the threads, enhancing the overall efficiency. The `mandelbrot()` function is called inside the inner loop to compute, for each point c of the considered region of the complex plane, the value of its associated pixel in the final image of $\mathcal{M}$; the results are then stored in the matrix `local_M` allocated by each process. The matrix `local_M` is accessed in a row-wise fashion in order to reduce the cache misses as much as possible. Additionally, a "touch-by-all" policy is employed to minimize the remote accesses: the memory required for the `local_M` matrix is allocated using `malloc` and therefore the actual mapping into the physical memory occurs only when the data is accessed by the OMP threads, allowing each thread to have its data placed in the most suitable memory location for efficient processing.
A further optimization involves pre-computing the quantities `y` and `index`, since they stay constant for all the pixels belonging to the same row of the matrix and consequently remain unchanged throughout all the iterations of the inner loop.

```
1
2  // Compute the local part of the matrix M on each process
3  #pragma omp parallel for schedule(dynamic)
4  for (int j = 0; j < local_rows; ++j)
5  {
6    const double y = y_L + (start_row + j * size) * dy;
7    const int index = j * n_x;
8    for (int i = 0; i < n_x; ++i)
9    {
10     double complex c = x_L + i * dx + y * I;
11     local_M[index + i] = mandelbrot(c, I_max);
12   }
13 }
```

Code 1: main parallel region to compute the values of the pixels of the Mandelbrot set's image. The quantities `x_L` and `y_L` are respectively the coordinates of the real and the imaginary part of the bottom left corner of the considered portion of the complex plane; `dy` and `dx` are the steps, while `n_x` is the number of columns of the matrix `M`. Finally, `start_row` corresponds to the specific process' `rank` and `size` is the number of processes.

We provide also the definition of the `mandelbrot` function which is called within the inner loop of the parallel region. As it is a small and frequently used function, the keywords `static inline` provided the compiler with hints for optimization, which led to significant performance improvements.

```
1
2  static inline int mandelbrot(const double complex c, const int max_iter)
3  {
4    double complex z = 0.0;
5    int k = 0;
6    while (creal(z)*creal(z) + cimag(z)*cimag(z) < 4.0 && k < max_iter)
7    {
8      z = z*z + c;
9      k++;
10   }
11   return k;
12 }
```

Code 2: definition of the `mandelbrot` function

We show a sample image of the Mandelbrot set $\mathcal{M}$ produced with the implemented algorithm.

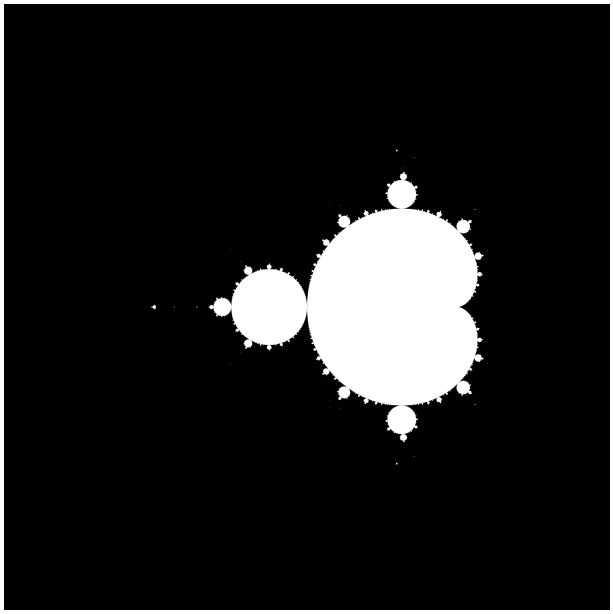

Figure 1: Mandelbrot set's image made of 5000×5000 pixels; this image has been produced setting $I_{max} = 65535$ and considering the square region of the complex plane centered at $(-0.75, 0)$, with a side length of 4.

3 Experimental setup

The implemented program has been tested to determine and assess the strong and weak scalings of the code; all the measurements have been conducted on the EPYC nodes of the ORFEO cluster [2]. Each EPYC node is equipped with two AMD EPYC 7H12 CPUs, which are based on the "Rome" architecture and composed of 64 cores each organized in 4 NUMA regions, each having 64 GiB of memory.
The program has been compiled employing the highest optimization level `-O3` along with the `-march=native` flag to maximize the performance of the code; this combination ensured that the code was highly optimized and tailored specifically for the CPU architecture on which it had to run.
In order to improve the reliability of the results, each time measurement has been collected six times and their mean has been computed, along with the standard deviation. Throughout these measurements, the program was executed setting $I_{max} = 65535$ and selecting the square region of the complex plane centered at $(-0.75, 0)$, with a side length of 4.
To streamline the analysis, the data collection process has been automated by using some `bash` scripts, which are available on the author's GitHub repository [1].

4 Strong scaling

The strong scalability of the algorithm has been studied focusing both on the OMP and the MPI scalings.

4.1 OMP

The analysis has been conducted by running the implemented program with a single MPI task and gradually increasing the number of OMP threads from 2 to 64, while keeping constant the total amount of workload. At each execution of the program a Mandelbrot set's image made of 1000×1000 pixels has been computed in parallel. In addition, we fixed `OMP_PLACES=cores` and `OMP_PROC_BIND=close` based on tests indicating this as the optimal configuration; the `close` binding policy provided slightly better performances compared to the `spread` one, although the difference was not significant. Figure 2 shows the plot[2] of the average execution time against the number of threads.
The speed-up and the efficiency have been computed for each employed number of threads. The speed-up Sp is defined as the ratio between the serial run-time of the code T_S, which we obtained by executing the code with a single MPI task and only one OMP thread, and the parallel run-time T_P:

$$Sp = \frac{T_S}{T_P} \tag{2}$$

Figure 3 contains the plot of the speed-up against the number of OMP threads.
The parallel efficiency is given by the following relation:

$$Eff = \frac{Sp}{p}$$

where p is the number of computing units (MPI tasks or OMP threads). The plot of the efficiency values against the number of threads is shown in figure 4.

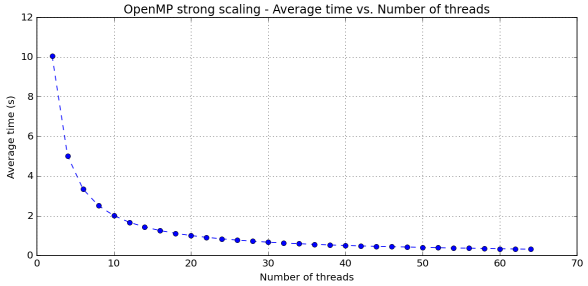

Figure 2: OMP strong scaling, average time against the number of threads

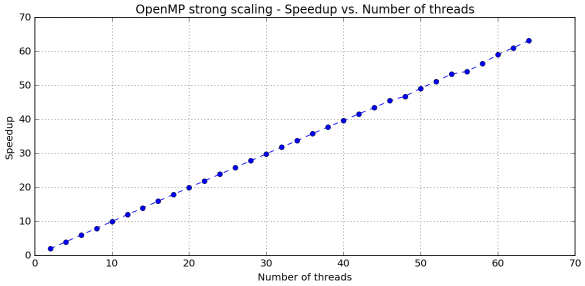

Figure 3: OMP strong scaling, speed-up values against the number of threads

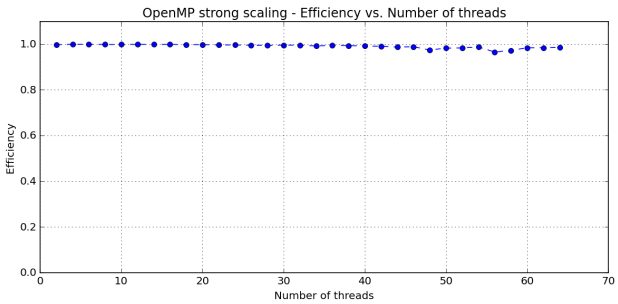

Figure 4: OMP strong scaling, efficiency values against the number of threads

[2]The error bars on this and the following plots are so small that they are not visible

By looking at figure 2, we can notice the substantial performance improvement achieved through the parallel implementation with respect to the serial execution time of the code, which has been measured at around 20 s. Moreover, we can observe that the average execution time consistently decreases as the number of OMP threads increases, proving the effectiveness of the parallel computation. Furthermore, the plot in figure 3 provides evidence of a nearly perfect linear speed-up, with a slope approximately equal to 1. This observation highlights the scalability and the efficiency of the parallelization strategy deployed. Additionally, as depicted in figure 4, the computed parallel efficiency closely approaches the ideal value of 1. Overall, the OMP strong scaling analysis indicates that the implemented program scales almost perfectly with the employed number of threads. The code demonstrates remarkable performance gains, with negligible parallel overhead.

4.2 MPI

The study has been conducted by running the code with a single OMP thread per MPI task and gradually increasing the number of MPI tasks from 4 to 254, while maintaining a constant workload. A total of two nodes of the EPYC partition of the ORFEO cluster have then been used. Also in this case, a Mandelbrot set's image made of 1000×1000 pixels has been computed in parallel for each execution of the code. Furthermore, we selected for the allocation of the MPI processes the `--map-by core` option of `mpirun`, in order to minimize the time needed for the communication between tasks. Figure 5 provides the plot of the average total execution time, which includes the time required for the MPI communications, against the number of processes. Also in this case, the speed-up and the efficiency have been computed for each employed number of MPI tasks; figure 6 contains the plot of the speed-up against the number of processes, while the plot of the efficiency values against the number of processes is shown in figure 7.

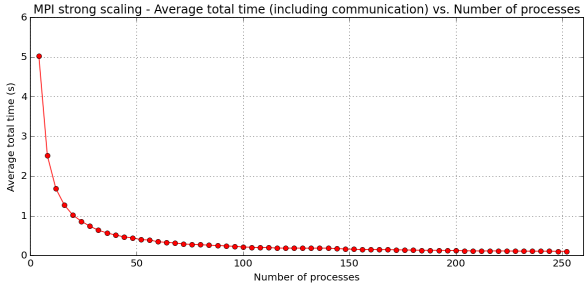

Figure 5: MPI strong scaling, average total time against the number of tasks

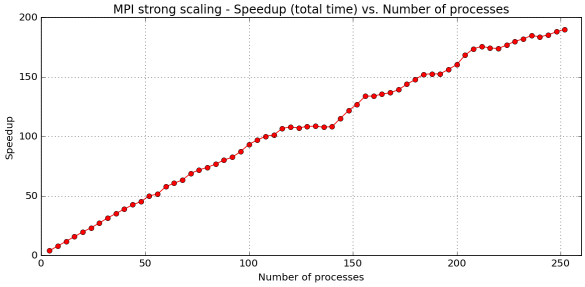

Figure 6: MPI strong scaling, speed-up values against the number of tasks

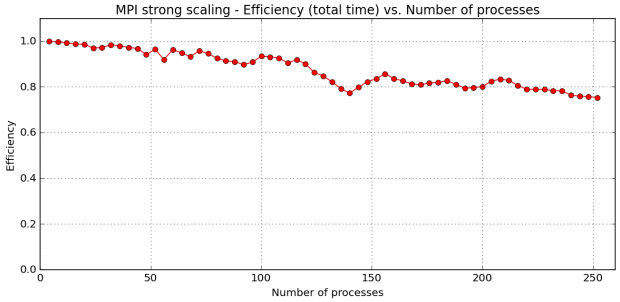

Figure 7: MPI strong scaling, efficiency values against the number of tasks

These plots provide evidence of the significant performance gains achieved through the parallel implementation, mirroring the previous observations from the OMP strong scaling analysis. Notably, we can notice that the average total execution time decreases with the number of MPI tasks, highlighting the effectiveness of the parallel computation. However, by examining figures 6 and 7, we can detect the effects of the parallel overhead, stemming from the processes' creation, and the communication costs, both of which tend to increase with the number of employed MPI tasks. In fact, up to 128 processes, which is the number of cores of an EPYC node of the ORFEO cluster, the computed speed-up is almost perfectly linear, closely reaching the ideal slope value of 1. Nevertheless, beyond this threshold, a slight departure from the ideal trend becomes increasingly evident. This deviation coincides with the allocation of a new node, amplifying communication times as inter-node communication generally takes longer than the one between processes allocated within the same node. Correspondingly, the parallel efficiency, while still being commendable, experiences a notable decline at around the 128 processes mark; the consistent downward trend we can observe, unlike in the OMP strong scaling analysis, is likely attributable not only to the growth of the communication costs, but also to the parallel overhead: generally, spawning threads within a process is much less costly than creating entirely new processes. Nonetheless, despite these observations, the overall performance gains achieved through the implemented parallelization remain significant, although they diminish beyond a certain threshold.

5 Weak scaling

The weak scalability of the code has been studied focusing both on the OMP and the MPI scalings.

5.1 OMP

To assess the OMP weak scalability, the code has been executed using a single MPI process and gradually increasing the number of threads from 2 to 64, all while maintaining fixed the workload assigned to each thread. Specifically, during these measurements each thread was tasked with computing 100×1000 pixels of the Mandelbrot set's image. Similarly to the methodology employed in the OMP strong scaling analysis, we fixed `OMP_PLACES=cores` and `OMP_PROC_BIND=close`. The plot of the average execution time against the number of threads is shown in figure 8.

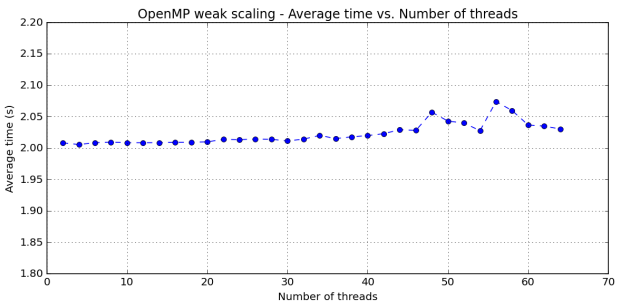

Figure 8: OMP weak scaling, average time against the number of threads

The plot in figure 8 provides evidence of the fact that the average execution time remains relatively constant as the number of employed threads increases, indicating a good weak scalability of the developed implementation. The consistent execution time suggests minimal parallel overhead in managing additional threads, highlighting the effective scaling with increased computational resources.

5.2 MPI

The MPI weak scalability analysis has been conducted by running the program with a single OMP thread per MPI task and gradually increasing the number of MPI tasks from 4 to 254, all while maintaining fixed the workload assigned to each process. Also in this case, each MPI process was tasked with computing 100×1000 pixels of the Mandelbrot set's image; furthermore, the `--map-by core` option has been selected for the allocation of the tasks. Figure 9 provides the plot of the average total execution time, which again includes the communication time, against the number of processes.

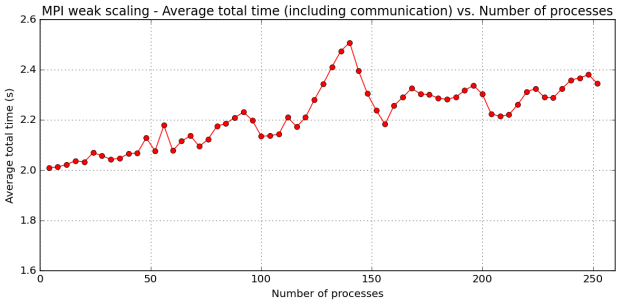

Figure 9: MPI weak scaling, average total time against the number of tasks

By looking at figure 9, we can notice that the code showcases a good weak scalability, as it manages to handle additional processes with minimal impact on the execution time. In fact, the average total execution time remains relatively stable, experiencing only a marginal increment with respect to the number of MPI tasks.

6 Conclusions

The analysis of the strong and weak scalings of the code highlighted the significant performance gains and efficiency achieved through the developed parallel implementation, confirming the suitability of the problem of computing the Mandelbrot set for being efficiently computed in parallel.

7 References

1. Piero Zappi, `High_Performance_Computing_Assignment` GitHub repository. URL: `https://github.com/PieroZ01/High_Performance_Computing_Assignment/tree/master`

2. AREA Science park, ORFEO documentation, 2023. URL: `https://orfeo-doc.areasciencepark.it`