High Performance Computing 2023 - Excercise 2c

Marco Zampar - SM3800032

September 25, 2024

1 Problem Statement

The aim of this excercise is to carry on the parallel computation in the programming language `C` of the Mandelbrot Set $\mathcal{M}$, defined as the set of complex points $c \in \mathbf{C}$ for which the recursive function (1) is bounded, starting at $z_0 = 0$. It may be proved that once an element z_i of the sequence is more distant than 2 from the origin, the sequence is then unbounded.

$$f_c^{n+1}(0) := z_{n+1} = z_n^2 + c \quad (1)$$

Hence, the simple condition to determine whether a point c is in the set $\mathcal{M}$ is the following

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

where I_{max} is a parameter that sets the maximum number of iteration after which the point c is considered to belong to $\mathcal{M}$ (the accuracy of the calculations increases with I_{max}, and so does the computational cost).

Numerically, leveraging the fact that $\mathbf{C} \sim \mathbf{R}^2$, the Mandelbrot set can be generated in a 2D grid and can be visualized as an 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 (1): if the sequence remains bounded for I_{max} iterations the pixel belongs to the set, otherwise not.

Since the computation of each point in the Mandelbrot set is independent, there is no need of coordination of the workload among multiple processors: it is clear that this problem is intrinsically parallel, promising an efficient and scalable parallelization.

This project's purpose is to develop an hybrid approach for the parallel computation of a portion of the Mandelbrot Set, combining Message Passing Interface (MPI) -distributed memory parallelism- and Open Multi-Processing (OpenMP) -shared memory parallelism-.

It is interesting to note that the Mandelbrot Set is symmetric along the x-axis, a simple proof can be found here, so if the portion of the set we want to compute is symmetric, we can simply compute half of the set and then merge the image with the reverted one! This is to remember that sometimes performance can be gained with some simple tricks.

In the GitHub repository of the project we propose a way to do that, but for the estimation of the computation times we worked with a version of the code without this symmetry trick.

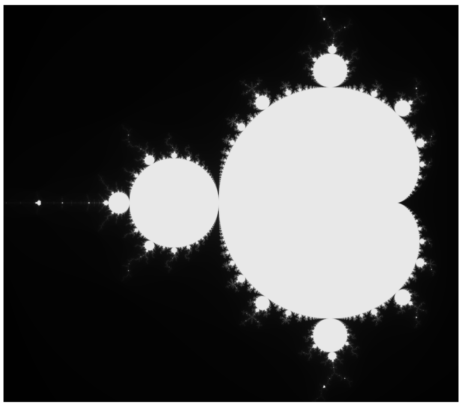

Figure 1: The Mandelbrot Set

2 Computational Resources

The computational resources employed are 2 EPYC nodes of the ORFEO cluster: as far as the estimation of the OpenMP scaling of the code, a socket of an EPYC node was leveraged, which means 64 cores divided in 4 NUMA regions. We could not use the entire node (both 2 sockets with 64 cores each) because an MPI process can run at most on a socket, not on an entire node, see the mpirun documentation.

As far as the estimation of the MPI scaling, 2 entire EPYC nodes `epyc005-006` were employed, which means a total of 256 cores.

More informations about the ORFEO cluster can be found here.

3 Implementation

The first thing to notice in the computation of the Mandelbrot Set is that in many of its portions some rows are computationally more expensive than others: a point that belongs to the set needs I_{max} iterations, while a point far from the set needs few iterations; another interesting aspect is that 2 contiguous rows are similar, also in terms of computational cost.

Eventhough the latter holds also for the columns of the set, we choose to process the rows and not the columns for a simple reason: in the progamming language `C` a matrix is stored row-wise, and since the set is represented and stored in memory as a 2D array, it would be pointless to process its columns because at every step the cache lines containing a row of the matrix will be changed, leading to a great waste of time.

The two considerations above led me to some choiches in the code, available at `mandelbrot.c` of the GitHub repository of the project:

- about OpenMP: in the first `for` loop, to distribute the work between threads I chose a dynamic scheduling, since the work is not perfectly balanced for every row.

- about MPI: I decided to distribute the work between processes based on their rank, this means that rank 0 will get the first, #processes-th, 2*#processes-th rows and so on; the same for other processes: in this way, since 2 contigous rows are similar, the amount of work between processes is balanced, or at least not too unbalanced. Hence, by giving to process 0 the first $\frac{\#rows}{\#processes}$ rows that may be far from the center of the set, we could incur in data starvation, beacuse a process that gets the rows close to the center of the set will have a lot of more work to do.

  Another aspect is dealing with the remainder of $\frac{\#rows}{\#processes}$: this is distributed between the processes such that rank is less or equal to the remainder of $\frac{\#rows}{\#processes}$.

Another interesting aspect of the code is the writing of the image, I tried two different approaches but one was way much better: using MPI IO functions, in particular `MPI_Write_at_all`, to write in parallel to the same file, but this brought 2 main issues: one is about the overhead, the other is the fact that MPI functions can only write binary code to a file, but a PGM file accepts only ASCII characters, in this way I had to create a binary file and then convert it in PGM file, doubling the total IO operations.

Therefore, I opted for a different approach: a `MPI_Gatherv` to collect the matrix in rank 0 and then a serial write to the PGM file.

In any case, I didn't measure the time of the writing of the file because it is serial and would affect the scaling.

About the allocation of the 2d array in memory, I used the `char` data type to store the value of the iteration number of equation (1) for the corresponding complex number, setting I_{max} to 255, in this way the unbalance between points belonging to a point of the set and one outside is not huge.

A final note about the compilation: the flags `-O3 -march=native` were leveraged, to get the most optimization possible.

4 Scaling

The `bash` files to perform the scaling are available at the folder `scaling` of the GitHub repository of the project.

It is common agreement that to get a nice scaling, not only the code has to be well designed and optimized, but also the use of the computational resources has to be wise and accurate.

About the OpenMP scaling, there are two choices I made:

- since we have to run all the threads on a single process, the best choiche was to bind it to a socket (running the application with the flag `--map-by socket --bind-to socket`), which is the largest region available for a process, in this way we got all the 64 cores of the socket available to run the threads..
- the other choice is about the threads affinity: I tried different combinations but the best were `OMP_PLACES=threads OMP_PROC_BIND=close` or `OMP_PLACES=cores OMP_PROC_BIND=close`, but in any case the difference between the other was not remarkable, e.g. setting `OMP_PROC_BIND=spread` there was almost no difference. This is probably due to the fact that each thread processes different rows, then the spread or close binding policy has low influence on the cache locality and cache coherency mechanism.

  I decided to perform the scaling with `OMP_PLACES=cores OMP_PROC_BIND=close`.

About the MPI scaling: we leveraged the binding policy of MPI processes on the cores. Also in this case I tried different allocations of the processes, but no significant difference emerged.

The measurements were repeated 5 times for the OpenMP scaling and 3 times for the MPI scaling.

4.1 MPI and OpenMP Strong Scaling

In order to quantify the strong scaling of a parallel application we have to measure the execution time with an increasing number of computing units - processes for MPI and threads for OpenMP - maintaining a fixed problem size. This is also known as the Ahmdal's law, which is relevant only if the serial fraction of work is independent of the problem size: in this case we can state that this is true, although there might be some serial - but inevitable - overhead due to the creation of the processes/threads.

Scalability is quantified using speedup, defined as the ratio of the time required for one processor to complete the work (approximately equivalent to the serial time, to have a fair compariosn we should use the time taken by the best serial application), compared to the time required by P processors. This relationship is expressed by Equation (1). In an ideal scenario, the speedup would be directly proportional to the number of processors (P), resulting in an efficiency (Equation 2) equal to 1.

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

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

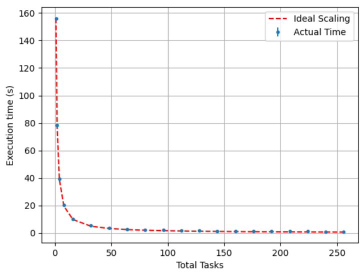

Figure 2: MPI Strong Scaling

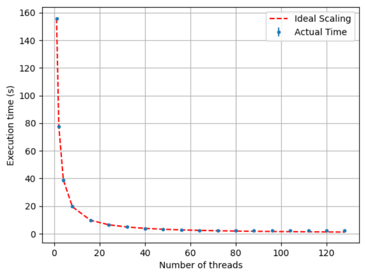

Figure 3: OMP Strong Scaling

In the estimation of the scaling, both MPI and OpenMP programs were launched with `nx = 25000 and ny = 25000`.

We can see that both MPI and OpenMP scale well:

- Concerning MPI scalign, we got a serial execution time of 156.168608 s, and a 256-processes execution time of 0.968919 s, meaning a final speedup of 161.178243 and a reduction of almost 99%.

  We get a decent speedup coefficient (0.65), which improves if we take only half of the point in the regression (0.8), hence we can see that the efficiency after 150 tasks starts to decrease more rapidly, in any case we don't get any plateau.

- Concerning OpenMP scaling, we got a serial execution of 156.742766 s, a 64-threads execution time of 2.440615 s and a 128-threads execution time of 2.445350 s, with a speedup of 64.2 and a reduction of almost 98.5%.

  We don't observe any improvement in working with more than 64 threads probably because of resource contention between threads that run on the same core.

It is nice to see that the efficiency is almost constant at 1 until 64 threads, meaning that we are leveraging well all the available computational resources: this is thanks to the dynamic scheuling of the threads, the independence of the rows which doesn't affect cache coherency and doesn't lead to false sharing, and the fact the I_{max} is 255, a bigger number would worsen the imbalance of the rows and would hamper the parallel scaling.

Another difference between the 2 approaches is that in MPI we have to perform a `MPI_Gatherv`, a collective operation that takes almost for every number of processes 0.4 s, on the contrary in OpenMP we don't have such an operation because we are working in a shared memeory enviroment, this is another reason why the OpenMP scaling is better.

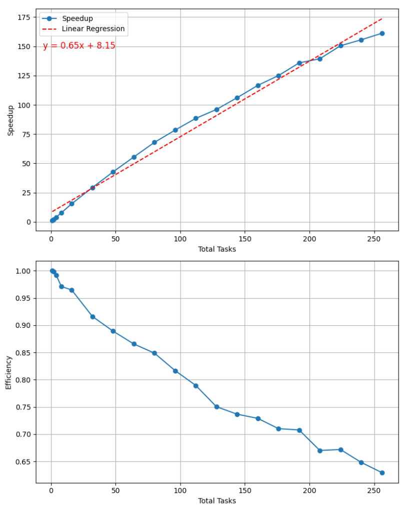

Figure 4: MPI Strong Scaling: Speedup and Efficiency

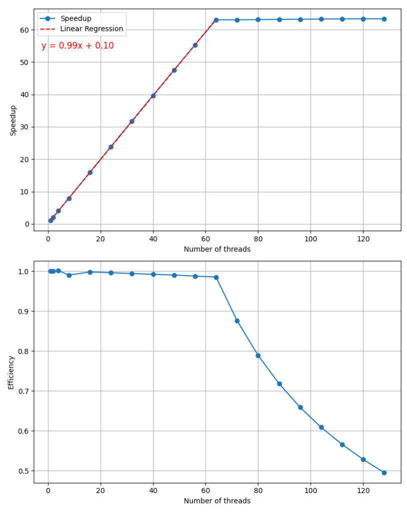

Figure 5: OMP Strong Scaling: Speedup and Efficiency

4.2 MPI and OpenMP 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.

To perform the estimation of weak scaling an initial value of `nx_1=ny_1=2000` was set, with the updating rule for P processors: $nx_P = ny_P = 2000 * \left\lceil \sqrt{P} \right\rceil$.

About MPI scaling, we can see from the plot that the execution time doesn't stay constant but grows linearly, this again can be attributed to the `MPI_Gatherv`, hence from figures 10 and 11 we can see that the computation time scales quite well, staying almost constant for all the 256 tasks, but the gathering time grows linearly and this affects the scaling.

On the contrary, OpenMP scaling behaves weel and similarly to the previous case, until 64 threads the execution time stays constant, but when more threads are added also some overhead is introduced and the execution time starts to grow linearly, leading to a decrease in efficiency.

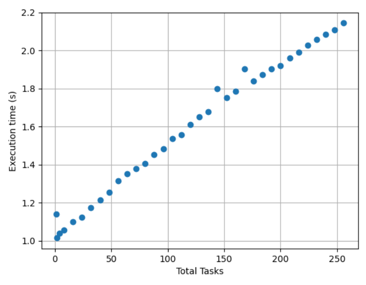

Figure 6: MPI Weak Scaling

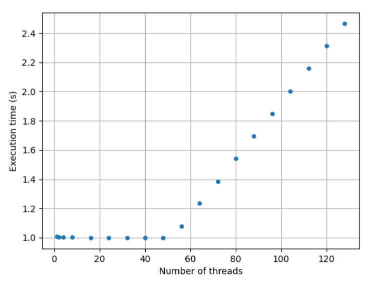

Figure 7: OMP Weak Scaling

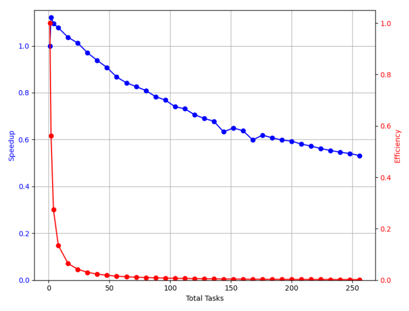

Figure 8: MPI Weak Scaling: Speedup and Efficiency

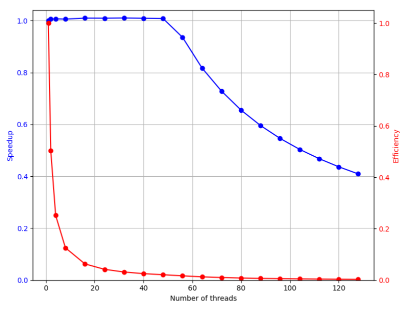

Figure 9: OMP Weak Scaling: Speedup and Efficiency

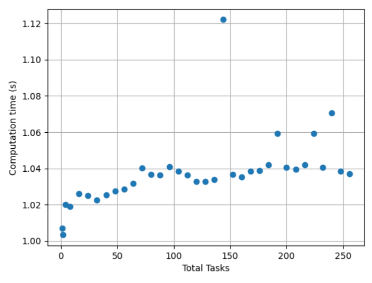

Figure 10: MPI Weak Scaling: Computation Time

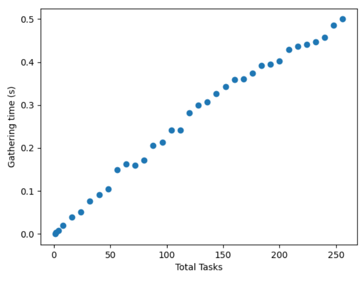

Figure 11: MPI Weak Scaling: Gathering Time

5 Final Considerations

Looking at the plots it may seem that the code works much better with OpenMP than MPI, but we have to note that, while the OpenMP scaling can't handle more than 64 threads: running more threads on a single CPU doens't bring any improvement, propably because of resouce contention, all the cores run a thread and probably in this case SMT is not efficient; on the contrary eventough its scaling is not perfect - due to some communication overhead - MPI can handle better a greater amount of processors, leading to a lower execution time percentual reduction in the strong scaling and to a constant scaling of the computation time.

We can explain this behaviour with the differences between shared and distributed memory: the shared approach cannot scale to large amount of processing units but with a low amount of processors is probably better than the distributed approach, but on a large scale a distributed approach is necessary and fundamental.

6 Hybrid Scaling

We also performed and Hybrid Scaling with both MPI processes and OpenMP threads: we leveraged an entire EPYC node, running from 1 to 8 processes and from 1 to 16 threads, mapping the processes with `--map-by numa`.

About the strong scaling: fixing `nx` and `ny` to 25000 we got an execution time of ~ 1.61 s (using 8 processes with 16 threads each), and a reduction of almost 99%.

About the weak scaling, the trend was similar to the MPI scaling: by adding processes the time grows linearly, but fixing the number of processes and adding threads time stays almost constant.

References

[1] Proof of the Symmetry of the Mandelbrot Set available at:https://www.quora.com/Is-the-Mandelbrot-set-completely-symmetrical-about-the-real-axis

[2] Source Code: GitHub Repository available at: https://github.com/mzampar/High_Performance_Computing_2023

[3] Open MPI mpirun Documentation available at:

https://docs.open-mpi.org/en/v5.0.x/man-openmpi/man1/mpirun.1.htmlthe-bind-to-option

[4] Computational Resources, ORFEO Documentation available at: https://orfeo-doc.areasciencepark.it/HPC/computational-resources/