Efficent generation of Mandelbrot set image sequences using MPI and OpenMP

Luigi Riz, Raffaele Pojer
{luigi.riz, raffaele.pojer}@studenti.unitn.it

Summary

The Mandelbrot set is a famous set of complex number that became popular due to his particular aesthetic. Its main characteristic is the infinitely complicated boundary that, with magnification, reveals finer and recursive details, thus making it belong to the set of fractal curves. Image 1 shows the set in a colored environment.
In this work we show how it is possible to create efficiently such images, exploiting the power of parallelism and multi-threading. All the computations have been performed using the HPC@UniTrento.

I. Introduction

The Mandelbrot set is the set of the complex numbers for which the function:

$$f_c(z) = z^2 + c \tag{1}$$

does not diverge to infinity when iterated from $z = 0$. These set images can be generated by sampling complex numbers c in the space region and checking if the sequence $f_c(0), f_c(f_c(0)), ...$ goes to infinity. If so, the complex number is outside the set and is coloured as black, otherwise the number of iterations until convergence is stored, and according to its value, a color will be assigned to the point. More formally, the Mandelbrot set can be expressed as the set of values c in the complex plane for which the orbit of the critical point $z = 0$ under iteration of the quadratic map $z_{n+1} = z_n^2 + c$ remains bounded. For example, for $c = 1$ the iteration sequence is 0, 1, 2, 5, 26, ... which tends to infinity. So, 1 is not an element of the Mandelbrot set.
A very important note to state, is that each pixel in an image is independent of the other ones, so its value does not depend on one of its neighbour points.
Exploring this set is an opportunity for researchers to implement always better and more efficient algorithms, leading to interesting results and more and more defined images.
Using the computational power of the cluster of the University of Trento, we aim at generating a sequence of deeper and deeper zooms in a given coordinate in the complex plane, in an ideal trip towards the infinitesimally small details of the boundaries of the Mandelbrot set (as in Figure 2). According to the available resources, the algorithm will automatically distribute the zooms and the parts of the images to the cores, exploiting parallelism and multi-threading.
Figure 1 shows the Mandelbrot set in a coloured environment.

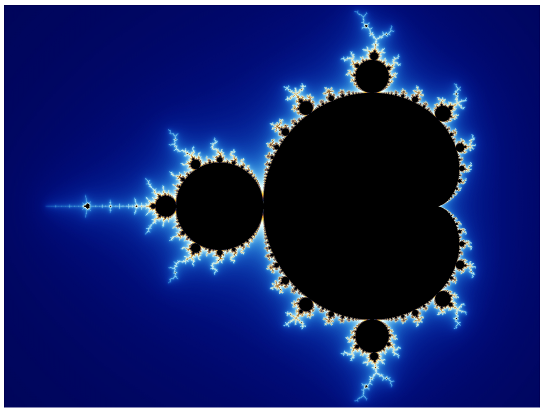

Fig. 1: Mandelbrot set image

II. Related works

During our research we found some articles to tackle the problem. In particular, the paper by Gamage *et al.*[1] uses the Message Passing Interface (MPI) to divide the image into partitions and assign each of them to a certain core. In particular, this paper tries three different partition schemes, each of them having a master-slave architecture. The three partition schemes are:

1) Naive approach: Row Based Partition Scheme
2) First come First Served: Row Based Partition Scheme
3) Alternating: Row Based Partition Scheme

In point **1** the image rows are distributed equally to the number of processors, meaning that each processor calculates the region of the image on which it will work. When the individual parts of the images are computed, the slave process sends the local buffer to the master process (via an `MPI_Send` - `MPI_Recv` pair). So, the master process collects all the partitions together and finally, it saves the generated image into the output file.
In point **2** the rows are partitioned mimicking the behaviour of an operating system scheduler. The master checks whether there are free cores and then, according to the node availability, it assigns a row at a time. If a processor finishes the job, the master collects the computed row and it can assign to the free node the next row to compute. When all rows are gathered, the master merges all of them and produces the output image.
In point **3** the authors suggest dividing the image row by row. Each processor works on all the rows for which

$$row_{index} \mod n_{processes} = process_{id}$$

So, this results in assigning the rows in an alternating fashion. As in previous methods, the master gathers all the parts computed by the slave processors and it finally generates the complete image.
The results show that this last approach produces the best performances, especially when the number of tasks is greater than 16, while the worst case is represented by the naive base approach. Differently from our work, the paper by Gamage *et al.*[1] focuses only on the generation of a single image, while in our project we also implement a technique to handle the zoom into a pre-specified image point, involving also the use of threads with OpenMP.

III. Problem analysis

As a first thing, we tried to generate the Mandelbrot set using a serial implementation. The starting algorithm was taken from [2], which has later been reshaped in a way to adapt it to the parallel paradigm.
The code in [2], summarized in Algorithm 1, simply iterates over all the pixels in the image, representing the complex plane, and counts the number of iterations the point takes to converge. If the number of iterations exceeds `IterationMax`, the color for the specific pixel is set to black, since the point is considered as diverging. In the other case, the color is set depending on the number of iterations the point takes to converge.
As shown in Algorithm 1, two for loops iterate over each pixel of the image, that is then mapped to specific coordinates in the complex plane $\mathbb{C}$. Then, another for loop iterates until convergence or until `IterationMax` is reached, considering equation 1 as reference function.

Algorithm 1 Serial Mandelbrot

```
Initialize the output .ppm file
for $iY = 0, 1, \dots iYMax$ do
    for $iX = 0, 1, \dots iXMax$ do
        Retrieve coordinates $Cx$ and $Cy$ from $iX$ and $iY$
        Initialize $Zx$, $Zy$, $Zx2$ and $Zy2$ to 0
        for $Iteration = 0, 1, \dots IterationMax$ do
            Update $Zx$, $Zy$, $Zx2$ and $Zy2$ according to formula 1
            if $(Zx2 + Zy2) < EscapeRadius$ then
                /* Convergence reached */
                Exit from this loop
            end if
        end for
        Calculate the color of the pixel [$iX$, $iY$] according to $Iteration$
        Write the RGB value in the output file
    end for
end for
```

We adapted the initial algorithm from [2] in a way to plot coloured images in place of the black-and-white ones it originally generated. To do so, the colour is determined using a set of logarithmic functions, that create smoother images, reducing the differences between near pixels.
A peculiarity of this algorithm, is that the RGB value of each pixel is not stored in memory, but is directly written on the output `.ppm` file in a purely sequential way. When transposing the algorithm into the parallel paradigm, we had to think a lot about this piece of code, since the access to the file has to be sequential and exclusively managed by only a process.
We started working on this implementation and we tried to adapt it, having in mind these four goals:

- Improve the performances using MPI (keeping as reference the article by Gamage *et al.* [1]);
- Exploit multi-threading working with OpenMP;
- Generate a sequence of images representing the zooms towards specific coordinates;
- Give the user the possibility to have control on the generation, by defining some optional command-line parameters such us the magnification factor, the ending coordinates, the number of threads used and others.

IV. Problem solution

We can divide our algorithm's execution path into two sub-paths. In the first, we have the case in which there are less processors than images to generate. In this case, each processor will have one or more frames to complete by itself in a sequential way. In the second case, we have more resources than frames to generate, so the processes have to coordinate themselves when working on the same frame.
We subdivide this section in two parts, one dedicated to the communication between processors (MPI), and the other part is about the threads implementation (OpenMP).

A. *MPI*

The main work in this project was about the communication between the processors. The most challenging part was related to the second sub-problem of our algorithm, that is when we have more processes than images to generate, so we have to distribute the resources during computation. Given the number of processors available, we divided it with the number of zooms to obtain the number of processors per each image. In order to handle all the different processors for each frame, we changed the default communicator in MPI (`MPI_COMM_WORLD`), that allows the communication between all the processors at once, and we substituted it with multiple communicators, one for each task. The logic is the same as the default one, but in this case each processor will have a local rank that will be used for the `MPI_SCATTER` and `MPI_REDUCE` operations. The `MPI_Comm_split` function creates new communicators by splitting the default communicator into new groups. Each frame is then processed in parallel with the others frames. As written before, we divided the images rows-wide and assigned each row alternately to each available processor in the group.
All groups have a ”frame master”, who is responsible to collect all the different rows computed by the others processors for the current frame. If the division between the rows and the number of process is not even, the remaining rows will be

finished by the master process. We noticed that the numbers of the rows that are computed by the master process does not influence the computation time, since the reminder of the division typically gives small numbers. All the frames are then written in a unique folder with the corresponding number of zoom levels as name.

B. OpenMP

In order to increase the performances of our algorithm, we decided to use also OpenMP, in a way to exploit multi-threading and speed up the computations. In particular, as a first step we analysed our parallel application, trying to spot pieces of code for which the application of parallel directives was possible. In fact, our goal was to use OpenMP for a fine-grain parallelization, since the coarse-grain one was already solved using MPI. Once the candidate lines of codes have been found, we carried on an analysis on data dependencies, trying to spot and remove them in a way to have the application work with the desired behaviour. Finally, once the dependencies have been removed, the OpenMP directives have been applied. The improvements in terms of performances are reported in Section V.

1) Possible places of application for OpenMP: Analysing our parallel application, we found two possible applications of the multi-threading library. The first one is to use the parallel directive to split among different threads the rows of the image a single process has to work with (see Listing 1 in the additional material), in the case of a single process working on one or more zooms. The second one is to adopt OpenMP to parallelize the computation of the set of pixels each process has to cope with, in the case of multiple processes composing the same image (see Listing 2). In both the code fragments, the application is computing the values of the iterations for each pixel and, as already sad, this calculation is independent from pixel to pixel. So we thought that these pieces of code were valid places of application for OpenMP.

2) Analysis and removal of data dependencies: Once we found the two candidate chunks of code to be parallelized, we performed the analysis of data dependencies, that was rather easy since we had only few variables and the arrays were used just in a write-only way. The results of the analysis are summarized in Table I in the additional material.
Looking at the table, it is easy to see that the only measure needed to remove data dependencies is the declaration of the variables `Cx` and `Cy` as private. In this way, the different threads work on their private copies of the variables and when they access them, they find the values they are supposed to find. There is no need to use a `lastprivate` directive, since the two variables are used just to store temporarily the complex coordinates the application is working on and then they can be removed from memory.
It is interesting to note that, although the introduction of OpenMP in our project has been very straightforward and rather easy, the improvements in performance we had are notable, as next section will underline.

V. Benchmarks

A. Setting

The last step of our project consisted in a benchmarking phase, in which we tried to analyze the performances of our application in terms of time. In particular we set up three different experiments: in the first one our code had to generate $400px \times 400px$ images using $IterationMax = 1000$, whereas in the other two cases we asked for $4000px \times 4000px$ plots with $IterationMax = 6000$. In the first two experiments we varied the amount of processes sharing the amount of work, using a single-thread solution. In the last trial, instead, we kept fixed the number of cores ($n = 16$) and modified the number of threads used. In all the experiences we tested the application on a growing number of frames, in a way to test our solution on problems with increasing complexity. Table II summarizes all the parameters used during this phase. To have a more stable and accurate time measure each trial has been performed 5 times. The results have been subsequently averaged and the mean time has been finally used to measure speed-up and efficiency.
It is worth underlining that the evaluation has been performed considering the overall time of computation, including the time needed to save the file, which is a well-known bottleneck that we knew would affect our performances.

B. Results

All the data we collected is available in an Excel file in our GitHub folder, so here we will just report and comment the graphs we extrapolated from all the data we had.

1) "light" experiment: This experiment was rather easy to solve, since the generation of 64 frames by a single process took less than 20 seconds. As a consequence, the use of parallelization brought some benefits, but the overhead inter-process communication introduced caused bad results in terms of efficiency. The speedup (Figure 3) and the efficiency (Figure 4) graphs highlight this behaviour. In fact, looking at the first one, we can see that the speedup is near to the theoretical linear speedup until 8 processes are used; when more processes are used the speedup is still growing, but less consistently, until we use 128 processes, when a drop in performances is observable. The efficiency graph suggests another aspect to be considered: the highest values of efficiency for each curve are usually reached when the number of processes is equal to the number of frames. After this peak, the efficiency curve stays nearly constant when using 2 or 4 processes per frame, then efficiency drops again, probably due to the communication and I/O overhead.

2) "heavy" experiment: The second experiment was far more complex to solve for our experiment. In fact, the worst case scenario (32 frames by a single process) took more than an hour to complete. In this case, the introduction of parallelization produced better results in terms of performances. Perhaps also in this case speedup and efficiency are influenced by the I/O overhead, that we think is an un-removable bottleneck of the application. The obtained results

are summarized in Figure 5 and Figure 6. Looking at the first plot, it is interesting to state that, even when introducing many processes (*e.g.* 128), there is no drop in the speedup, that remains strictly increasing. However, the plotted curves are still far from the theoretical linear speedup.
The second graph, apart from showing an outlier when generating 1 frame with 2 processes (in that case we have an efficiency greater than 1), shows a behaviour similar to the one depicted in Figure 4. In fact, also in this case the efficiency for each curve decreases, until a new local maximum is found when each frame is generated by a single process. After that, the curve remains stable for the next few points, after which it starts again to decrease.

3) "heavy threads" experiment: In the last experiments we wanted to explore the impact multi-threading has when solving our problem. The results, summarized in Figure 7 and Figure 8, suggest that the use of multi-threading is effective to improve the performances. In fact, the speedup plot shows that all the curves have a strictly increasing behaviour, even if we still have to underline that they are far from the theoretical curve.
The efficiency graph, instead, shows that overall the efficiency values are higher, suggesting that the use of multi-threading introduces less overhead than the use of multi-processing.

VI. Conclusion and future works

The process of developing this application helped us understanding the power and the complexity of High Performance Computing, in which many different aspects have to be considered when solving a given problem. In particular, we think that some other analyses and improvements could be introduced in our project. For example, it should be interesting to explore the performances of our application, leaving out the time needed to write the output files on the disk. In this way, without the influence of the non-parallelizable bottleneck of I/O, we could better understand the goodness of our parallelization approach. From a very rough benchmark, in which we considered only the time needed to produce the image in memory without any file saving, we saw that the speedup curve is closer to the theoretical linear speedup one. As a consequence also the efficiency values are closer to 1.
Another possible field of improvement could be the exploration of new functions to map the number of iterations of each pixel to its final RGB values. In fact, with the adopted solution solution, some images (in particular the more zoomed ones) tend to have very uniform colors, resulting in a loss of details of the boundaries of the Mandelbrot set (as *e.g.* in the last images in Figure 2).

References

[1] B. M. S. V. Gamage and V. M. Baskaran, "Efficient generation of mandelbrot set using message passing interface," *CoRR*, vol. abs/2007.00745, 2020.

[2] Rosettacode, "Mandelbrot set." https://rosettacode.org/wiki/Mandelbrot_set#C, 2021. [Online; accessed 13-December 2021].

[3] Wikipedia, "Mandelbrot set — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Mandelbrot%20set&oldid=1059560683, 2021. [Online; accessed 12-December-2021].

ADDITIONAL MATERIAL

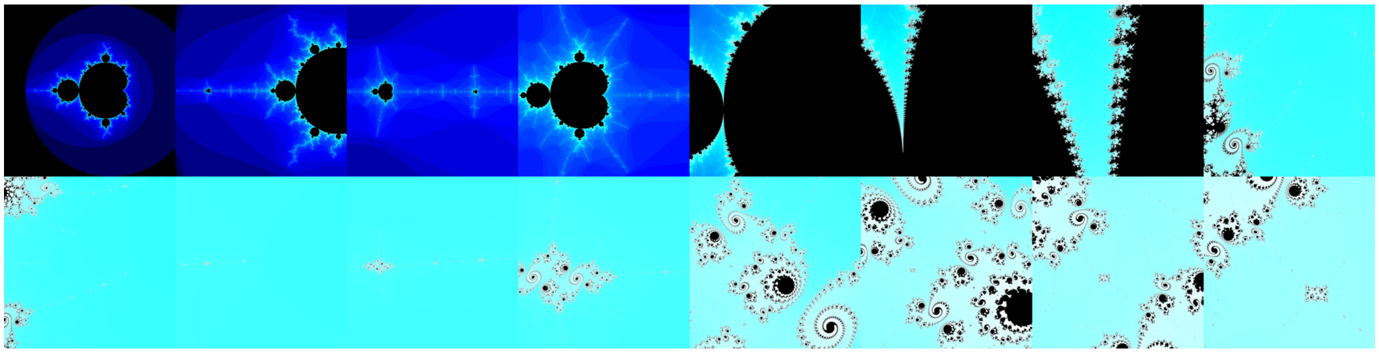
Fig. 2: 16 zooms into the set

Listing 1: First place of application for OpenMP

```
157         #ifdef _OPENMP
158         #pragma omp parallel for num_threads(arguments.threads) private(Cy, Cx)
159         #endif
160         for (int iY = 0; iY < iYmax; iY++)
161         {
162             Cy = CyMin_cur + iY * PixelHeight;
163             if (fabs(Cy) < PixelHeight / 2)
164                 Cy = 0.0; // Main antenna
165             for (int iX = 0; iX < iXmax; iX++)
166             {
167                 Cx = CxMin_cur + iX * PixelWidth;
168                 iterations[iX][iY] = mandelbrotIterations(Cx, Cy, IterationMax, ER2);
169             }
170         }
```

Listing 2: Second place of application for OpenMP

```
234     #ifdef _OPENMP
235     #pragma omp parallel for num_threads(arguments.threads) private(Cy, Cx)
236     #endif
237     for (int i = 0; i < iXmax * rows_per_proc; i++)
238     {
239         Cy = CyMin_cur + (i / iXmax + frame_rank * rows_per_proc) * PixelHeight;
240         if (fabs(Cy) < PixelHeight / 2)
241             Cy = 0.0; // Main antenna
242         Cx = CxMin_cur + (i % iXmax) * PixelWidth;
243         iterations[i] = mandelbrotIterations(Cx, Cy, IterationMax, ER2);
244     }
```

Memory location	Earlier Statement			Later Statement			Loop carried?	Kind of dataflow
	Line	Iteration	Access	Line	Iteration	Access		
Cy	162	i	Write	164	i	Write	No	Output
Cy	162	i	Write	168	i	Read	No	Flow
Cx	167	i	Write	168	i	Read	No	Flow
Cy	239	i	Write	241	i	Write	No	Output
Cy	239	i	Write	243	i	Read	No	Flow
Cx	242	i	Write	243	i	Read	No	Flow

TABLE I: Data dependencies in Listings 1 and 2

Experiment	Resolution	# Iterations	# Frames		# Processes		# Threads	
			From	To	From	To	From	To
light	$400px \times 400px$	1000	1	64	1	128	Fixed to 1	
heavy	$4000px \times 4000px$	6000	1	32	1	128	Fixed to 1	
heavy threads	$4000px \times 4000px$	6000	1	32	Fixed to 16		1	128

TABLE II: Parameters used for the different experiments during benchmarking

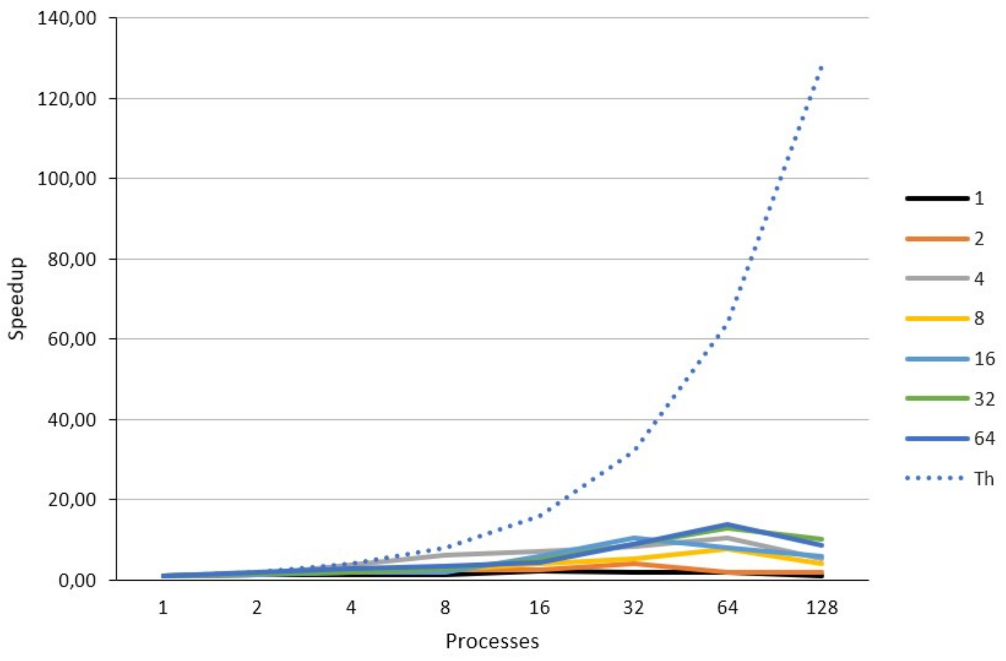

Fig. 3: Speedup recorded for the “light” experiment. The label of each line represents the number of generated frames. The dashed line represents the theoretical speedup.

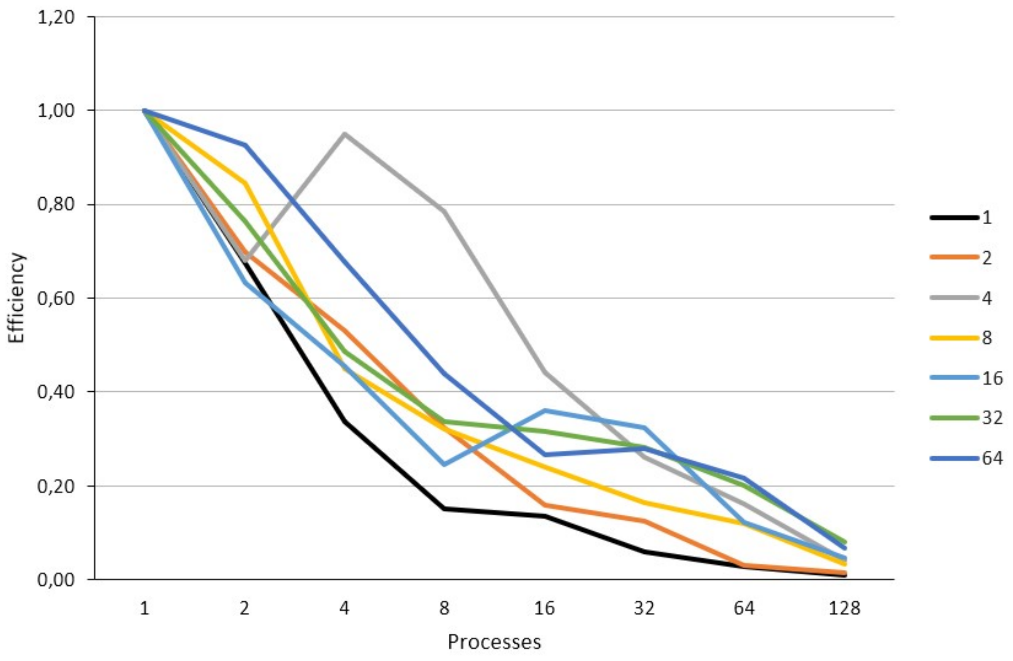

Fig. 4: Efficiency recorded for the “light” experiment. The label of each line represents the number of generated frames.

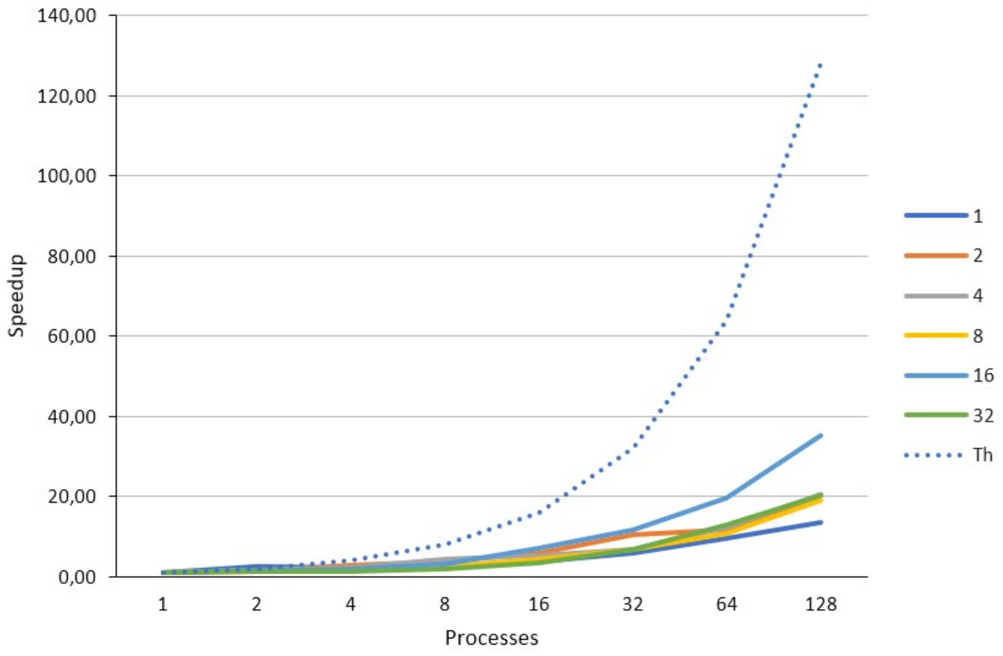

Fig. 5: Speedup recorded for the “heavy” experiment. The label of each line represents the number of generated frames. The dashed line represents the theoretical speedup.

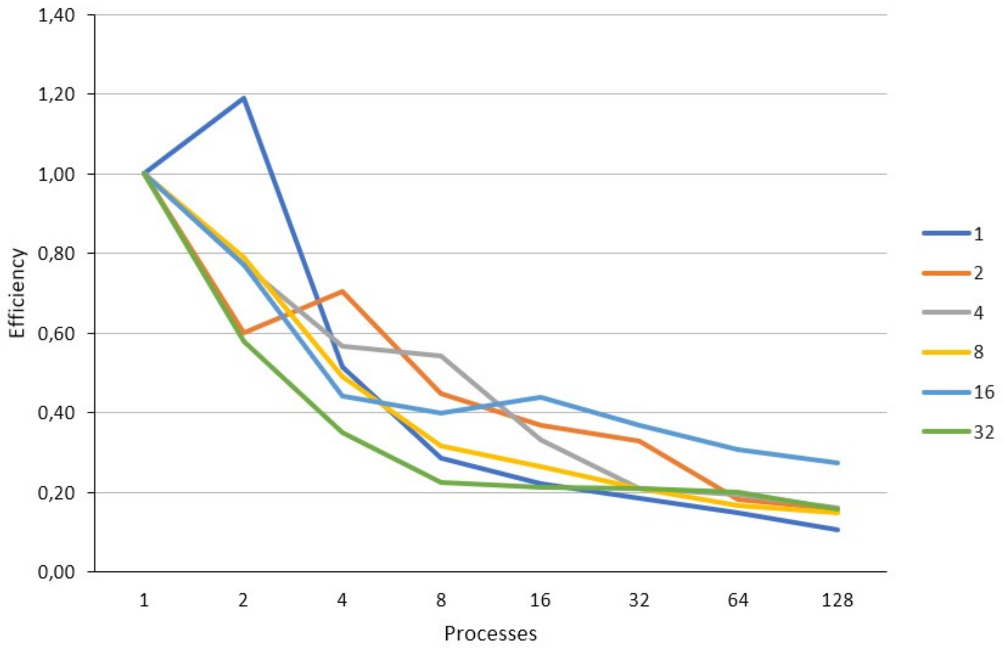

Fig. 6: Efficiency recorded for the “heavy” experiment. The label of each line represents the number of generated frames.

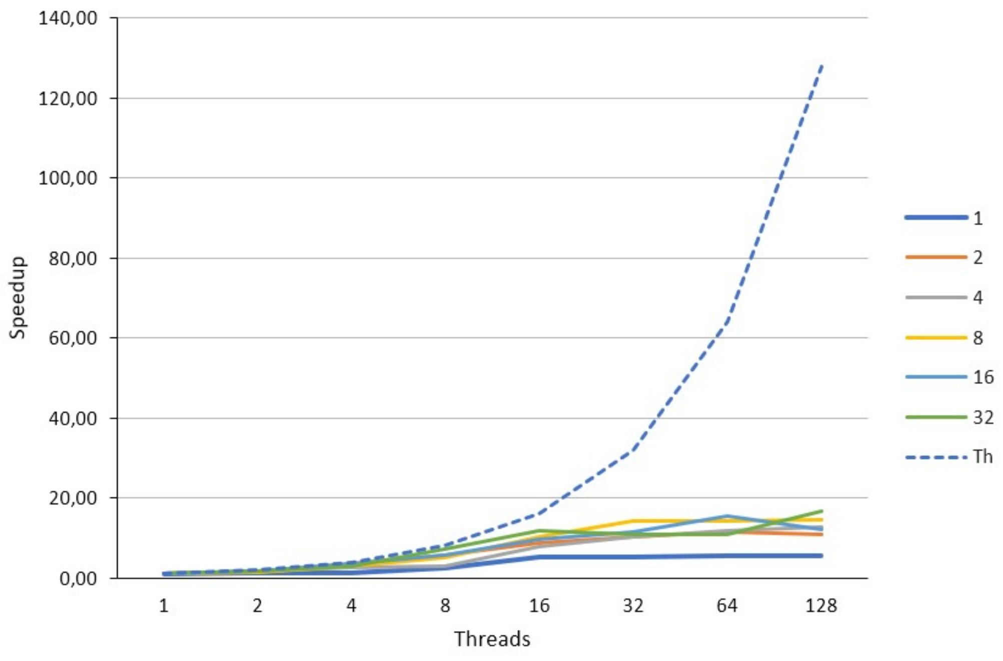

Fig. 7: Speedup recorded for the "heavy threads" experiment. The label of each line represents the number of generated frames. The dashed line represents the theoretical speedup.

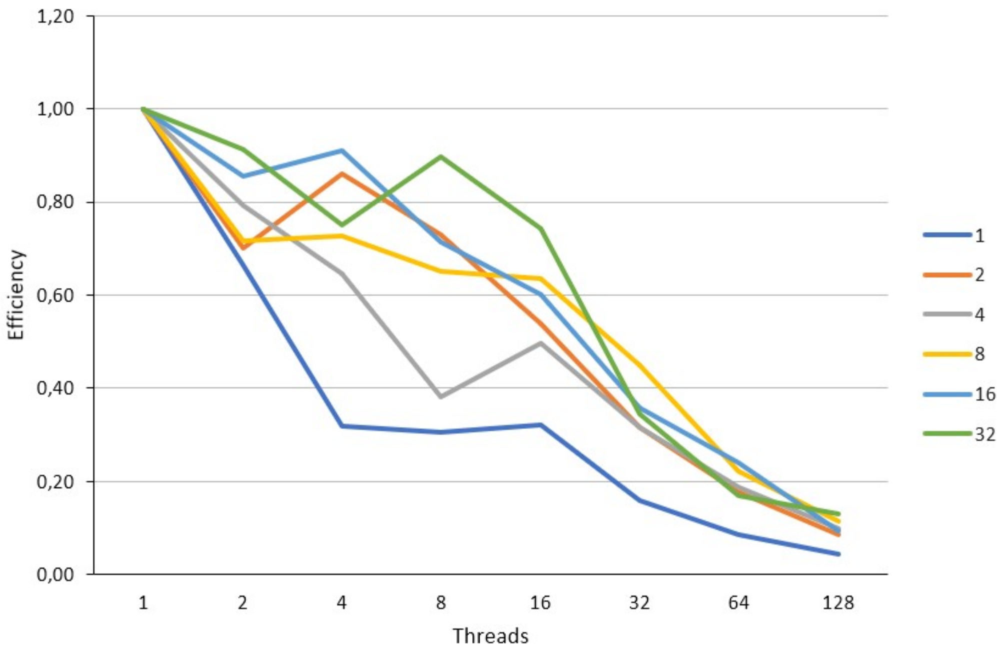

Fig. 8: Efficiency recorded for the "heavy threads" experiment. The label of each line represents the number of generated frames.