PROJECT PROPOSAL PHPC-2017

Pure-MPI and hybrid OpenMP/MPI parallel implementations of a Mandelbrot set generator

Principal investigator (PI)	Aymeric Galan
Institution	EPFL
Address	Lausanne
Involved researchers	–
Date of submission	December 11, 2017
Expected end of project	June 30, 2017
Target machine	Bellatrix cluster (EPFL)
Proposed acronym	PARMANDEL

Abstract

The Mandelbrot set is certainly the most famous illustration of fractals, but one does imagine that it is surprisingly generated by a very simple equation. In this project, we implement a high-performance version of the algorithm, in pure C++. A detailed comparison is provided between our three versions : a serial version, a parallelized version using MPI, and an hybrid version using OpenMP and MPI paradigms.

1 Presentation of the problem

1.1 Mathematical formulation

The Mandelbrot set is defined in the complex plane. It corresponds to the set of complex numbers c such that the recursive equation given by [1] :

$$z_{n+1} = z_n^2 + c\,, \text{ with } z_0 = 0 \tag{1}$$

remains bounded, that is the modulus of z_n stays below some threshold value $|z_{\text{th}}|$. It has been demonstrated that any complex number whose modulus is higher than 2, does not belong to the Mandelbrot set (the equation above diverges) [2]. Hence one usually sets $|z_{\text{th}}| = 2$. In order to generate an image of the Mandelbrot set, one represents it as the complex plane itself. The position on the image gives real x and imaginary y part of the complex number $c = x + i\,y$. In this work, we choose for simplicity to only consider a square image, of size $N \times N$, and we fix the limits to be $[-2, 1] \times [-1.5i, 1.5i]$. The iteration given by Eq. (1) is performed on each individual "pixel" c, that determines if it belongs to the set or not. Consequently, a maximum number of iterations has to be defined to avoid infinite calculations. The last step of this algorithm is to assign a value to each pixel, that represents how fast the divergence is. Every pixel that belongs to the set has the value 0. On the opposite, every pixel for which the iterative process tends to infinity has either a value equal to the number of iteration needed to reach the condition $|z_n| > |z_{\text{th}}|$, either the maximum number of iterations $n_{\max}$.

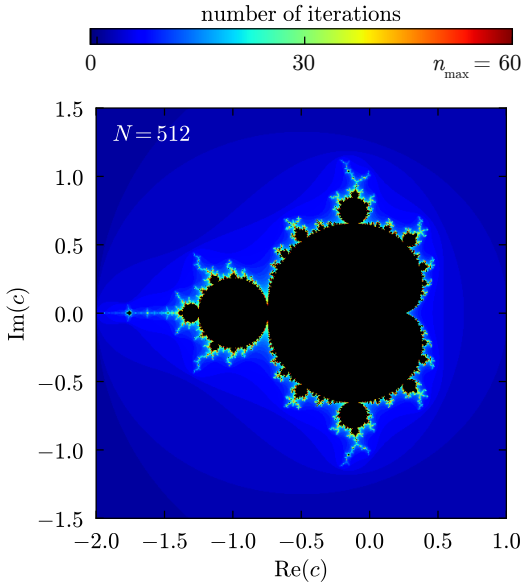

Figure 1: Example of output of a Mandelbrot set computation, for an image size $N = 512$ and a maximum number of iteration per point $n_{\max} = 60$. A simple program was written in Python for this illustration.

In order to illustrate the result of the algorithm, Fig. 1 shows the resulting image, with parameters $N = 512$ and $n_{\max} = 60$. It is generated by a simple Python program written for the sake of illustration.

1.2 Preliminary comments

The key point of this problem is that computation of each pixel is totally independent of other pixels of the image. Hence one calls this problem to be embarrassingly parallel. This encourages a lot the implementation of a parallel version of the algorithm, in place of a standard serial version. However, we must pay attention to the distribution of computing tasks. Indeed, it can be easily seen that the number of iterations needed to reach the divergence limit is absolutely not the same across the complex plane. At a point located far from the origin, z_n will diverges with a much higher rate than for a point close the edge of the Mandelbrot set. Moreover, a point that appears to belong to the set causes the iteration to last the longest possible, until $n_{\max}$. In order to parallelize efficiently the algorithm, one has to make sure that every parallel process has almost the same amount of work than other processes. Namely, we have to take great care of load balancing.

2 Implementation

2.1 General organization

We implement the algorithm as follow. Since it is written in plain C++, we choose to make use of the oriented-object capabilities of the language. A new class `MandelbrotSet` is defined, whose main parameters are the dimensions of the problem, a buffer that contains a representation (1D array) of the image, and an object `Dumper` used to generate ASCII or binary image. The `MandelbrotSet` class has one public method, `run()`, whose role is to compute the set and output an image, depending on input parameters (size of the problem, maximum number of iterations, *etc*) and compiler options (I/O toggle, use of `std::complex`, verbose mode, *etc*). Concerning the actual computation of the set, in order to visit all the pixels of the image, we use two nested `for` loops. Once the real and imaginary parts of the current point are calculated, a third `for` loop proceeds to the iterative process defined by Eq. (1). In order to write the image, we use a standard ASCII output for serial and OpenMP versions, and a binary output as soon as MPI I/O is involved. In this work, we first concentrate on the parallelization of the main computation, namely the coverage of the whole grid and pixel values assigning. In a second time, we focus on the parallelization of the I/O, which is the main bottleneck of the parallelizations paradigms employed here.

In this work, the main quantity extracted from the code is the run time, denoted by T, in seconds, which is defined as the time taken by the program to finalize the computation, and if set, to write an image. In order to obtain it whatever the version of the code (serial or parallel, OpenMP, or MPI), we implement a `Timer` class with simple methods to start, stop and get time. We then create two subclasses that inherit from `Timer`, one for non-MPI code using standard C++ library `chrono`, another for MPI code using `MPI_Barrier()` and `MPI_Wtime()` to get precise timing over the entire execution.

2.2 Implementations of parallelism

The OpenMP paradigm is used here to accelerate the computation of rows in the image. It is implemen-tatied as a `#pragma` right before the first `for` loop. More precisely, we use `#pragma omp parallel for schedule(dynamic)`. The first three statements are self-explanatory. We add the `schedule(dynamic)` clause since the amount of work at each pixel of a row is not constant at all, as explained in previous section. It lets OpenMP to organize its work in a dynamical way to face this local load balancing issue.

The MPI side of the implementation is done as follows. We choose to implement it as a master/workers algorithm, namely there is typically one rank (the master) that assigns parts of the image to each other ranks (the workers). In order to take into account the load balancing constraint, the worker divide the image in n_{row}. This new parameter can be passed as a third value to run the program, after the image size N and the maximum number of iterations per pixel $n_{\max}$. When the method `run()` is called, the master rank first sends to each worker a row to compute, using the blocking method `MPI_Send()`. When a worker finishes its computation after receiving the message (with `MPI_Recv()`), it sends a message to

the worker using same methods, then waits until it receives either a new row to compute, or a special tag meaning that it can quit.

Concerning the I/O, it is also parallelized using MPI. If the image output is toggled on, as soon as a worker finishes to computes its row, it writes it down in the binary image file. It accesses and writes to the file using special MPI methods (`MPI_Open()`, `MPI_Write_at`, *etc*...), trough a MPI communicator. In order to avoid issues during access to the file when multiple workers finish their work while some others do not, we split the main communicator into one communicator for each workers, and one for the master. This way, each worker can freely open, write, and close the file without worrying about the state of other workers. We would like to precise that this way of proceeding could not be the best. First, because it can create a lot of communicators, while a common communicator for all workers sound more intuitive. Secondly, because it is a kind of *short circuit* to the built-in MPI-I/O ensemble of methods, which is supposed to rely on communicators, maybe to avoid possible issues. In this work, as a first version of the code, we stick to this naive approach. In the section dedicated to suggestions of improvements, we give additional informations about this part of the code.

The hybrid version simply puts together both OpenMP and MPI implementations. The only difference resides on the initialization of the MPI environment, by using the method `MPI_Init_thread()`. For the required level of thread support, we use `MPI_THREAD_FUNNELED`, since the multi-threaded environment does not call any MPI method during the computation of each single pixel.

2.3 Optimization

It is crucial to optimize as much as possible the implementation of the algorithm before trying to use any parallelization paradigm. As a first step in the optimization process, we can remove useless computations in the algorithm, such as square roots, by squaring any complex modulus. We can also make sure to reuse variables that are already in the cache when possible. For instance, in Eq. (1) one has to compute the square of a complex number, that involved to compute the square of both real and imaginary parts of that number. Further in the algorithm, the modulus of that complex number is required to check for divergence, which required again squared real and imaginary parts. Using same variables again can have a significant impact when the number of iterations become large. Regarding operations involving to double the value of a number, one can simply add the number to itself instead. This way, a multiplication is replaced by an addition, which can have an impact on the computing time when dealing with high precision floating-point numbers, when multiplication is sometimes more affected by numbers precision than addition. Besides these optimizations on arithmetic operations, we first debug the code with `gdb`, the, optimize it using profiling tools such as `valgrind` and `gprof`.

3 Theoretical considerations and predictions

3.1 Qualitative aspects about complexity

We expect the computation time to depend mainly on the size of the image N, as well as on the maximum number of iterations allowed to compute a single pixel $n_{\max}$. The worst-case complexity of the algorithm can be estimated quite easily. As mentioned in the section dedicated to implementation, the program consists on three nested `for` loops. First two loops iterate on the pixels of the image, in x and y directions, with indices going as usual from 0 to $N-1$ with unity step in each loop. At this point, it is said to have a complexity $\mathcal{O}(N^2)$. The third one, located inside these loops, iterates from 1 to $n_{\max}$ with unity step. But in reality, it goes to $n_{\max}$ only for a point that belong to the Mandelbrot set. Hence the complexity of the whole algorithm is $\mathcal{O}(N^2\, n_{\max})$. Moreover, for a small value of $n_{\max}$ compared to the image size, we expect the algorithm to scale as $\mathcal{O}(N^2)$. Inversely, if $n_{\max}$ is sufficiently large so that N^2 is negligible, we imagine the complexity to tend towards $\mathcal{O}(n_{\max})$. In this work, we often fix $n_{\max}$ to 100 or so, thus the scaling of the algorithm should stays close to $\mathcal{O}(N^2)$.

3.2 Performance expectations

Besides complexity aspects, that are a quite abstract at this stage, one can make predictions about how much the serial version of the algorithm would be improved if it is parallelized in any way. Two cases

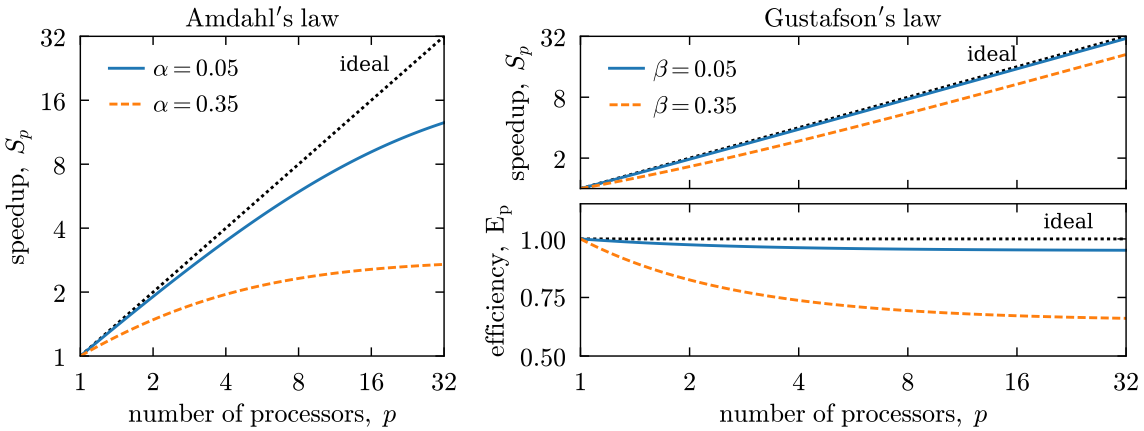

Figure 2: Performance predictions concerning strong and weak scaling, for a code that is 5% or 35% non-parallelizable. The code implemented for the work should behave like the first value when not using image output, and more like the second value with a non-parallelized I/O.

are usually considered in order to observe the speedup due to parallelization as a function of the number of processes or threads : fixing the size of the problem and increase the number of processes/threads (strong scaling), or increase both size and number of processes/threads (weak scaling). The relevant quantity in such tests is the speedup S_p, defined as the ratio between the run time of the serial over the run time of the parallelized code, that is $S_p \equiv T_1/T_p$. Note that in the case of a mutli-threaded code (OpenMP or hybrid), p can be replaced by the number of threads per processors, denoted by t in this work.

Strong scaling and Amdahl's law A theoretical prediction on the performance increase by a parallelized program over a serial one can be given by Amdahl's law [3]. Historically, it is the first of the two point of view presented in this work. It states that increasing the number of processes while keeping fixed the size of the problem leads to a bounded speedup at some point, depending on how much the code can be parallelized. Mathematically, it is defined by the following formula for the speedup S_p for a code running on p processors/threads :

$$S_p = \frac{1}{\alpha - \dfrac{1-\alpha}{p}} \quad , \tag{2}$$

with α the percentage of non-parallelizable code. A high α leads to a high upper limit of the speedup. One calls the situation described by Amdahl's law to be a strong scaling test, when N is fixed and only p varies.

Weak scaling and Gustafson's law A second estimation of S_p is provided by Gustafson's law [4]. Based on the idea that Amdahl's law does not reflect real-world applications, it considers the case of increasing at the same time the number of processes and the size of the problem in same proportions. In such a case, one would expect that the execution time remains constant. This law is defined as follows :

$$S_p = p - \beta(N)\,(p-1)\ . \tag{3}$$

This time, in the portion of non-parallelizable code $\beta(N)$, one takes into account the fact that the system to be computed is bigger. In fact, it can be thought as the percentage of execution time that is spent on the serial part of the code, for a given data set size. In contrast with previous situation, Gustafson's law is related to the so-called weak scaling. The speedup S_p, in the case of weak scaling, can be studied along with the parallel efficiency, defined by $E_p = S_p\,/\,p$, with p the number of processors/threads.

As discussed in above sections, the algorithm that generates the Mandelbrot set is, at first approximation, entirely defined by three nested loops (if one does not consider any I/O-related code). Hence the great majority of the code, that is close to 100%, is parallelizable. In order to give a more precise estimation of α, we use the profiling tool `gprof` to measure the relative execution time of the part that computes the Mandelbrot set, in the serial version code. It appears that even for large images ($N = 10\,000$), the percentage of the total run time used to compute the set does not go below 95%. However, as soon as we activate image output, this percentage drastically decreases to 65%. Applying these numbers to both scaling laws, with $\{\alpha, \beta\} = 0.05$ without I/O and $\{\alpha, \beta\} = 0.35$ with I/O, one can provide an estimation of how well the code will scale once parallelized. For this purpose, we plot Amdahl and Gustafson's laws on Figure 2, for both values of α and β. We see that Amdahl's law gives much more pessimistic predictions than Gustafson's law. Indeed, event for $\alpha = 0.05$, the speedup starts to decrease significantly at 8 processors, whereas no such speedup loss is observed in the case of Gustafson's point of view. In the following, we expose results in order to provide a comparison between these predictions and the real scaling of our code.

4 Results

In this section, we present results from various scaling tests, applied on different versions of the code : serial, pure-MPI, and hybrid OpenMP/MPI. The pure OpenMP version is not tested, since this work is more focused on the MPI load balancing and an hypothetical improvement brought by multi-threads. These results are all obtained from runs on the Bellatrix cluster, at EPFL [5]. Each data point correspond to the mean value computed over 10 distinct runs of the same executable. This ensure the accuracy of the results. Additionally, error bars are verified to be smaller than the size of markers on plots. Except when explicitly mentioned, the maximum number of iterations per pixel, $n_{\max}$, is fixed to 100 for simplicity (visually, it gives satisfying results on the final image). The discussion of these results is given in the next section.

4.1 Pure-MPI version

Figure 3 shows the strong scaling test of the pure-MPI version, that is the speedup at fixed image size N when increasing the number of processors p. No output is considered here. We choose to fix N to 10 000, which is a quite large image, to avoid issues that could arise on small grids. We also provide a plot of the run time, as well as comparison curves corresponding to ideal case and Amdahl's prediction for $\alpha = 0.05$. Multiple values of the number of rows for the grid division (load balancing) are tested. The worst case is given by $n_{\text{row}} = p_{\text{w}}$, which is the minimum number of rows possible, equal to the number of worker processors. On the opposite side, the presumably best case is also shown, when the number of rows is equal to the total height of the image, corresponding to N rows of 1 pixel in height.

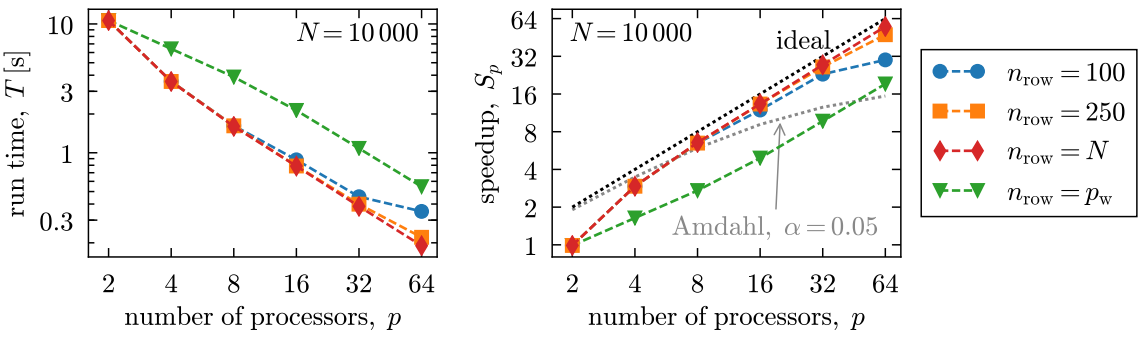

Figure 3: Strong scaling test for pure-MPI version of the code, without I/O, for an image size $N = 10\,000$. Different values for the load-balanced row division are shown, along with the maximum division $n_{\text{row}} = N$, and the minimum division $n_{\text{row}} = p_{\text{w}}$.

Figure 4 exposes weak scaling test of the same code. It also shows results from different values of n_{row}. At this point, it is worth noting how one normalized the speedup to compute the parallel efficiency. The definition E_p states that it is equal to S_p divided by the number of processors p. But in the case of the master/workers paradigm, there are $p_{\text{w}} \equiv p - 1$ processors that *truly* compute the set. For that reason, there is an ambiguity on the denominator to use to compute E_p. At high p values, there is almost no difference, since $p \approx p_{\text{w}}$. But for a small numbers of processors, it does impact on the efficiency. For this reason, we plot both cases on Fig. 4, in order to illustrate this difference. Note also that when doubling the number of processors, we do not double N, since it is defined by the height (and width) of the image. We double the total number of pixels, which is equivalent to multiply N by a factor $\sqrt{2}$ while p is multiplied by 2.

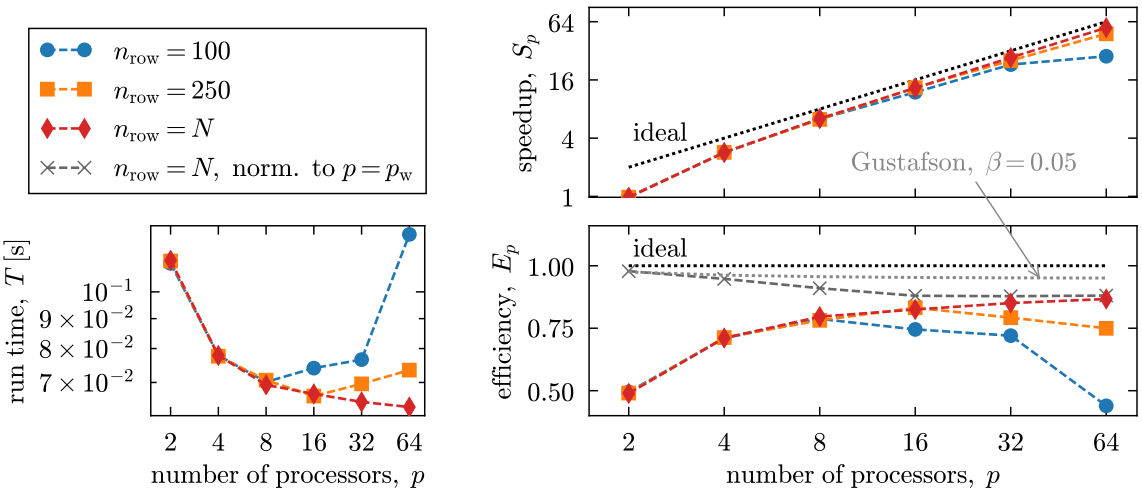

Figure 4: Weak scaling test for pure-MPI version of the code, without I/O, for an image size going from $N = 1024$ to $N = 5793$. Different values for the load-balanced row division are shown, along with the maximum division $n_{\text{row}} = N$, and the minimum division $n_{\text{row}} = p_{\text{w}}$. For the best-case curve on the efficiency panel, we plot the same curve, but normalized to the number of worker processors instead of the total number of processors.

4.2 Hybrid OpenMP/MPI version

In this section we expose results obtained with the OpenMP/MPI hybrid version of the code. Compared to the pure-MPI code, there is one extra degree of freedom, because one can vary the number of threads per processor as well as the number of processors. In order to deal with this situation, and not to study every possible configuration, we first proceed to a simple strong scaling test, at fixed number of processor, while making the number of threads t to vary from 1 to 16. We can not go higher than 16, because of the hardware configuration. Indeed, on the Bellatrix cluster, each node contains two sockets, and each socket contains 8 CPUs. Since it appears that one single thread can run on a single CPU, one could only run 16 threads on a node, and assign only one MPI task to that node. Moreover, for optimal results, one should pay attention to memory access. Each socket has easy access its RAM, and another access to the neighboring socket. Hence, one has to make sure, if it is possible, that all threads of a single task run on the same socket, in order to avoid at maximum the latency caused by memory access on another socket. All these restrictions can be asked to the SLURM scheduler on the cluster, thanks to multiple `#SBATCH` directives at the top of the batch script [6].

Consequently, for the first scaling test, we fix the number of worker processors to 1, in order to find the maximum number of threads per processor that still leads to interesting speedup gain. The efficiency of this OpenMP implementation should depends on the "size" of the parallelized loop, since it is usually not very justified to use this paradigm to boost computations on small datasets. For this purpose, we plot results of this scaling test for 3 different problem sizes, from $N = 1\,000$ to $N = 10\,000$, namely medium image to a quite large one. Figure 5 shows this initial strong scaling test with respect to the

number of threads. We clearly see that for medium sized images, going up to 16 threads for one processor does not seem to be bring a good improvement. Thus we restrict the following tests to a maximum of 8 threads per MPI task.

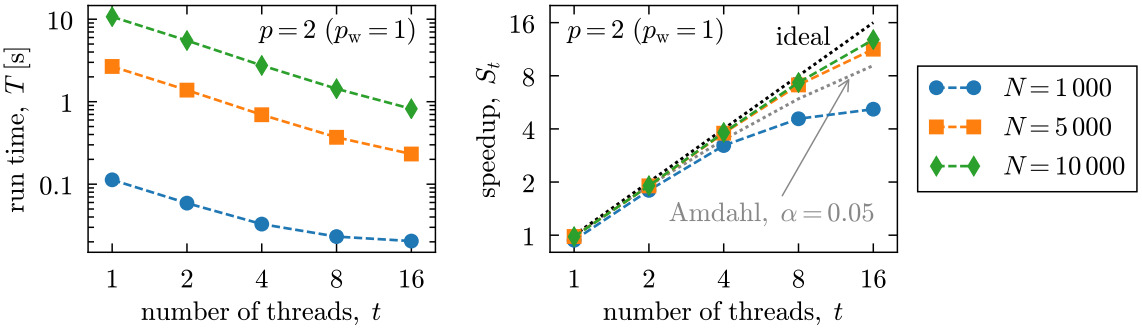

Figure 5: Strong scaling test for hybrid version of the code, without I/O, and varying the number of threads per processor while keeping the number of worker processors to the minimum. Results for different image sizes are plotted. All tests are run with $n_{\text{row}} = N$ (best load balancing).

Figure 6 shows results from the more "standard" strong scaling, with varying number of processors. We plot run time and speedup curves for 4 and 8 threads per processors. In order to see whether or not the hybrid version is superior to the pure-MPI version, we also bring back the best result from Fig. 3, which happens to be the case when $n_{\text{row}} = N$, that is the load balancing is optimal.

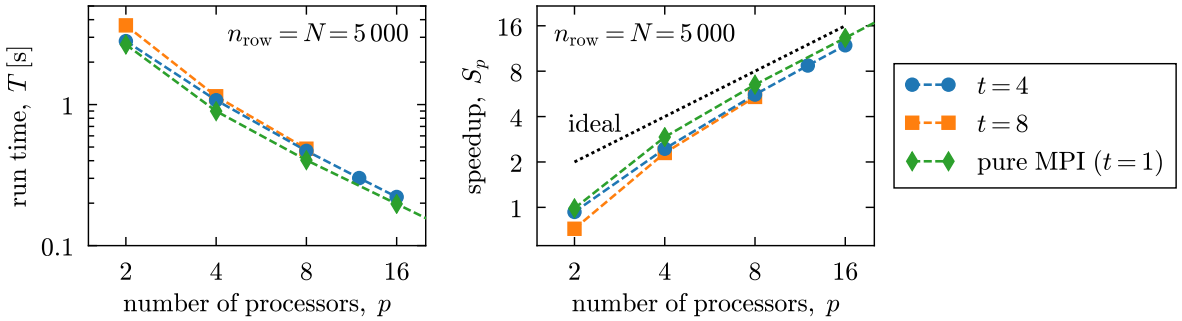

Figure 6: Strong scaling test for hybrid version of the code, without I/O, for 4 and 8 threads per processor. All tests are run with $n_{\text{row}} = N$ (best load balancing).

Results of weak scaling test of the hybrid version are exposed on Fig. 7. In a similar way to the strong scaling, we show results for 4 and 8 threads per processor, as well as the best pure-MPI case.

4.3 Influence of I/O

All results above concern the computation of the Mandelbrot set, without any output taken into consideration. We would like to show how much the addition of the image output affect the total run time of the program. On Fig. 8, we plot run times of serial, pure-MPI running on 16 or 64 processors, and hybrid code running on 16 processors with 4 threads per processor. Each version is tested with and without image output, so every curve can be compared with its same version with I/O, as well as with other versions. For MPI and hybrid version, the I/O is parallelized using the MPI paradigm, so each worker processor write its own part of the image as soon as it completes its computation. Note that each parallelized version in this section assumed the setting $n_{\text{row}} = N$ (maximal load balancing allowed by our algorithm).

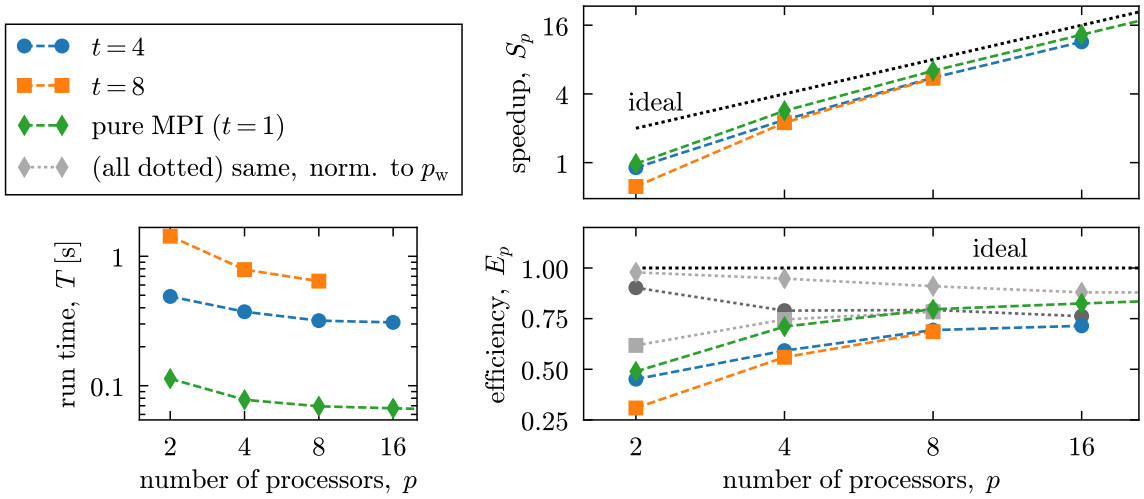

Figure 7: Weak scaling test for hybrid version of the code, without I/O, for 4 and 8 threads per processor, and an image size going from $N = 1024$ to $N = 4096$. All tests are run with $n_{row} = N$ (best load balancing). Every colored dotted line have a corresponding gray and dotted one on the efficiency panel, representing the same speedup but normalized to the number of worker processors, instead of the total number of processors.

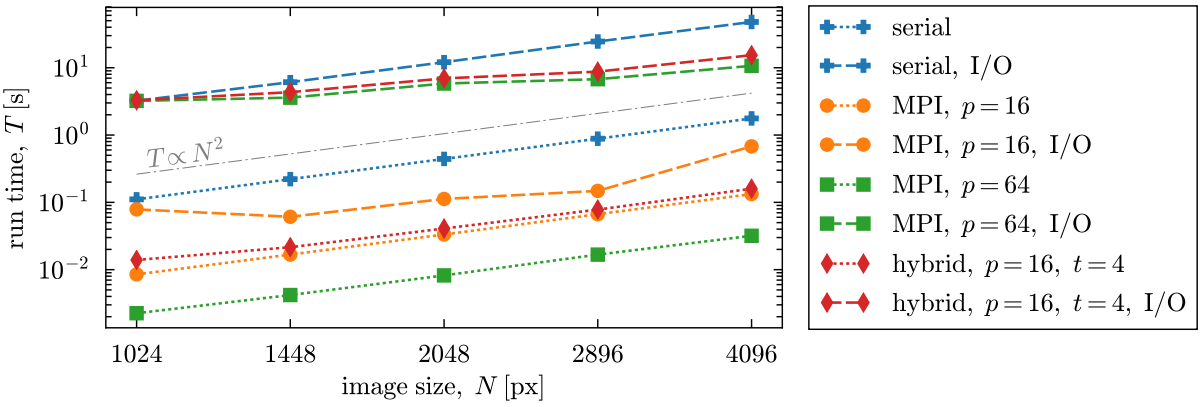

Figure 8: Comparison of run times of various versions of the code, with and without image output. Versions of the code are indicated in the legend. Dotted lines represent results without I/O, dashed lines with I/O. Note that I/O is parallelized for MPI and hybrid version. The dot-dashed line indicates the $\mathcal{O}(N^2)$ complexity.

5 Discussion

5.1 Pure-MPI version

By varying the number of rows in the decomposition of the grid, used to solve the problem of load balancing occurring in the Mandelbrot set computation, one can directly see how important the value n_{row} is this master/workers version of the code. Indeed, on Fig. 3, the higher n_{row}, the better the speedup. In addition, obviously, when there are absolutely no load balancing, results are quite bad. Compared to the prediction from Amdahl's law with 95% of parallelizable code, the code scales very good, even until 64 processors and with $n_{row} = 100$. The case when load balancing is maximum, that is $n_{row} = N$, appears to be the best, but it should not be unsurprising. When the n_{row} is large, the number of MPI communications is large too, which can cause latency thus a loss of performance. We do not observe effects of that kind, even if the size of the image is quite large ($N = 10\,000$).

Weak scaling results (Fig. 4) give relatively equivalent results. Despites the case $n_{row} = 100$ appear

to be a lot worst that other configurations, one see that using a good load balancing improve a lot the parallel efficiency. This time the theoretical prediction from Gustafson's law with $\beta = 0.05$ is not reached, but when the code is running on 64 tasks, the efficiency is close to the prediction. We clearly see that as soon as the number of rows in the division is comparable to the number of workers, the run time dramatically increases, because load balancing is not optimal in such a case. Finally, we recall the ambiguity that appears when working with a master/workers algorithm, when one normalizes the speedup to find the efficiency : either one stays at the plain definition of the efficiency, *i.e.* to divide by the total number of processors, either one try to rectifies it and divide by the number of *working* processors. In this work, we plot both case, without stating what is the best practice. In any case, at high p values ($p \geq 16$ from our results), this ambiguity almost disappears.

5.2 Hybrid OpenMP/MPI version

Usually, when the pure-MPI version of the code gives convenient scaling results, as it is the case in this work, one does not need to add a multi-threaded architecture to the code. For the purpose of this project, we still build and study an hybrid OpenMP/MPI version of the code. As detailed in Sec. 4, we first find what could be the best number of threads per MPI task, before proceeding to strong and weak scaling tests. One observes on Fig. 5 the influence of the problem size on the speedup provided by OpenMP. For image sizes of around 1 000 pixels (medium size), adding multi-threads is not worth it when the number of threads per processors is above 4 or 8. On the contrary, for large images, the speedup is quite close to the ideal case. Then, we decide to keep a maximum of 8 threads per processors, and expecting that a configuration of 4 threads per processors could still give better results that 8, since the weak scaling test concerns medium sized images, up to $N = 4096$.

A first glance on the run times of the hybrid code compared to the pure-MPI code, when varying the number of processors, is sufficient to confirm the small benefit from multi-threading (Fig. 6). Concerning the speedup, it is also evidently the case, event if the difference between the two versions is quite small. We could not go higher than 16 MPI tasks with 4 threads on each on the cluster we used for our computation. Maybe at a higher number of tasks, the speedup of the hybrid code could be slightly more interesting than the pure-MPI, but certainly not by a large amount. The weak scaling test (Fig. 7) sensibly gives same results, again separating a little bit more than the strong scaling test the hybrid and pure-MPI versions. For such images sizes and number of MPI tasks, the pure-MPI code is definitely the best in this situation, at least without any I/O considerations.

5.3 Influence of I/O

We now discuss performance of serial, pure-MPI and hybrid versions of the code, including the I/O part in timing measurements. Results are shown on Fig. 8. One observes a large increase of the run time, and a difference between non-I/O and I/O that increases with the image size. This is an expected result, since there are much more pixels to write in a large image. For all tested versions, the image output make the run time to raise at least of one order of magnitude. The most important increase happens with the MPI code running on 64 processors, with more than two orders of magnitude. It points out the main drawback of the I/O parallelization used in this work. When a lot of worker processors are involved, a lot of new MPI communicators are created (one for each worker) by the algorithm, and access to the file often happens within a different communicator than the previous access. All this MPI directives could be the reason of this result, by introducing a lot of useless latencies at different point of the program. A more detailed study needs to be conducted in order to confirm or not this hypothesis.

Besides the fact that the 64 tasks-MPI version does not give positive results for I/O, one can anyway confirm that all codes running the parallelized algorithm for image output lead to a lower run time than the plain serial code, even if the difference is not so large concerning hybrid and MPI on 64 processors versions. The best result among all versions tested in this work is achieved by the MPI code running on 16 processors. It even gives better timings than the serial code without I/O. Moreover, it is the one which causes the tiniest increase of run time between versions with and without image output. Consequently, this configuration appears to be a good balance between MPI communications, load balancing, and image size.

Table 1: Resources budget grand total.

Total number of requested cores	$8 - 16$ cores
Minimum total memory	100 Mo / MPI process
Maximum total memory	400 Mo / MPI process
Temporary disk space for a single run	None without I/O, 400 Mo for large image
Permanent disk space for the entire project	Depends on the number of images on the disk at the same time
Communications	pure-MPI, with OpenMP for hybrid version
License	MIT
Code publicly available ?	Yes (Github)
Library requirements	–
Architectures where code ran	Sandy Bridge (Bellatrix cluster, EPFL)

5.4 Suggested improvements

The main drawback of the implementation described and studied in this work is related to the I/O. As indicated in Sec. 2.2, the algorithm used to allow the workers to write seamlessly in the same file could lead to unnecessary latencies through the presence of a lot of communicators, but could also be fragile because it by-passes the built-in communication processes of the MPI environment. An improved version of it could be to first split the main communication into master-only and all-workers communicators. Then, each time a worker has to quit, one splits the worker communicator, to make it again to contain only current workers. And iterate this process until no workers are left.

An improvement can also be brought to the master/workers algorithm. When a worker ends its job on a row, almost at the time as another one, one of these two workers could wait more than it should until the master send a new row to it. A solution to avoid these latencies could be to make the master to send two rows at a time to each worker. This way, workers can compute a second row in place of waiting to a new row from the master rank, which could hopefully lead to some gain in speedup.

6 Resources budget

In this section, we describe briefly the required resources to run our code. Results exposed in this paper imply that one needs up to 16 cores to run MPI tasks. On each core, one should be capable of running up to 8 threads, if one wants to get reliable results from the hybrid version. For a large image ($N = 10\,000$), which is usually too large to be opened in a standard image viewer application without latency, the memory usage per processors is about 350 Mo. In order to be conservative, we set the maximum memory usage by a single run of the program to 400 Mo. Table 1 summarizes the necessary budget to run our code on a cluster.

7 Conclusion

In this work we explored a way to parallelize a Mandelbrot set generator. We saw that a simple mathematical formulation can lead to complex structures in the complex plane from the point of view of the image output, while representing a so-called embarrassingly parallel problem from the point of view of high-performance computing. Two paradigms were implemented, OpenMP for multi-threading, and MPI for parallelization over several machines as well as for the I/O. An hybrid version has also been

developed, giving at this stage slightly less positive results than the pure-MPI version, in a given computing environment. We leave for future work a couple of suggested improvements, that could certainly improve the parallel efficiency of our program.

Acknowledgments

The author would like to thank Nicolas Richart (EPFL), who helped him a lot during the realization of this project.

Source code

The entire source code built during this project can be found on Github and c4science repositories :

- Github : https://github.com/aymgal/ParallelMandelbrot
- c4science : https://c4science.ch/source/pmandelbrot/.

The reader can visit one of these repository to see an output example of the program studied here.

References

[1] Douady A. and Hubbard J. H., *Etude Dynamique des Polynômes Complexes*, 2007.

[2] Fredriksson B., *An introduction to the Mandelbrot set*, 2015.

[3] Amdahl G., *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, 1967.

[4] Gustafson, J. L., *Reevaluating Amdahl's Law*, 1988.

[5] Bellatrix cluster specifications, EPFL, http://scitas.epfl.ch/hardware/bellatrix-hardware.

[6] Slurm batch commands documentation, https://slurm.schedmd.com/sbatch.html.

[7] CPP reference, http://en.cppreference.com/w/.

[8] Hunter, J. D., *Matplotlib : A 2D graphics environment*, 2007.

[9] Stéfan v. d. W. *et al.*, *The NumPy Array: A Structure for Efficient Numerical Computation*, 2011.