Parallel generation of a Mandelbrot set

MIRCO TRACOLLI
MIRCO.TRACOLLI@STUDENTI.UNIPG.IT

Department of Mathematics and Computer Sciences, University of Perugia

ANTONIO LAGANÁ AND LEONARDO PACIFICI
LAGANA05@GMAIL.COM, LEONARDO.PACIFICI@UNIPG.IT

Department of Chemistry, Biology and Biotechnology, University of Perugia

April 21, 2016

Abstract

The generation of the Mandelbrot set is a typical seemingly embarrassingly parallel problem because each initial data set generates an event independent of the others. There are, though, several technicalities implied by the transformation from the sequential algorithm to the parallel one to be performed leading to different performances depending on the computational approach chosen. The paper discusses two different parallelization schemes and rationalizes why they lead to different performances by making some considerations about the tools used.

1 Introduction

A Mandelbrot set[1] is a general class of complex numbers c for which the succession generated by the relationship $z_{(n+1)} = z_n^2 + c$ is limited. In its basic algorithmic implementation for each given value of the complex parameter c, one evaluates if the succession generated by iterating the above given relationship diverges or not. The c value is retained as belonging to the Mandelbrot set only if the succession is limited. It has been shown that whenever $|zn| > 2$ the series diverges and c does not belong to the Mandelbrot set. A typical image generated by a Mandelbrot set is given by the fractal shown in figure 1 below as a function of the real and the imaginary part of c.

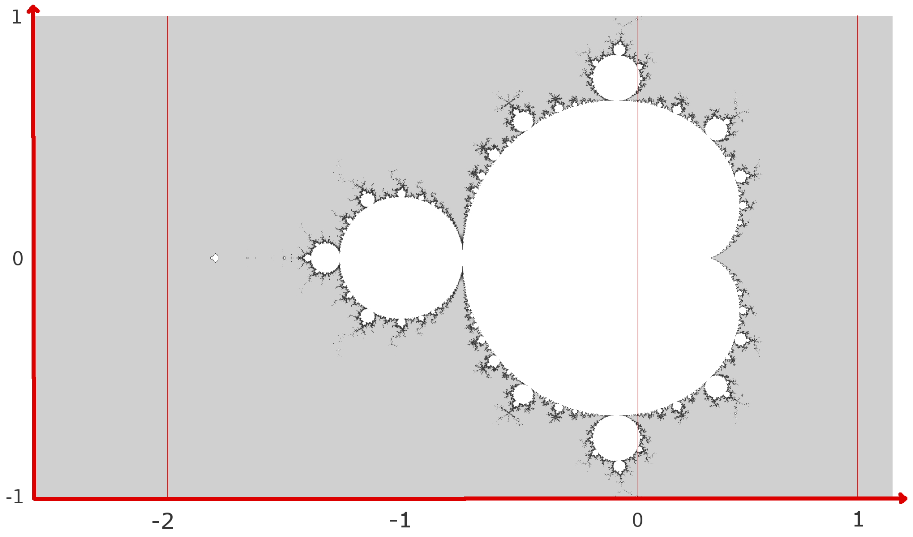


Figure 1: A typical image of a Mandlbrot set generated in our computational study.

The study was started by running the code written for generating the Mandelbrot set in a sequential manner (that means by iterating the above given equations for a set of randomly generated values of c) and then the code was restructured to run in parallel. Parallel runs were performed on a computer network, in which the compute resources are distributed.

The compute platforms used for our calculations belong to two different infrastructures:

- platform A - a cluster of 11 *Intel* dualcore *32 bit* processors with 2 Gb of Ram and *i686* architecture
- platform B - a cluster of 13 *Intel* quadcore *64 bit* processors with 2 Gb of Ram *x86_64* architecture

both working on the *Scientific Linux version 6.5*[2] OS.

These infrastructures can be accessed from two front ends named respectively *cgcw*[3] and *fecw*[4].

In the present paper we sketch in section 2 the main features of *MPI*, the software used for the parallel distribution. In section 3 we present the considered problem and the relative issues to solve. In section 4 we discuss the performance of the tests carried out. In section 5 we draw some conclusions.

2 MPI

MPI (Message Passing Interface[5]) is a standardized and portable message-passing system that allow users to create programs that communicate. The typical MPI user is a programmer that needs to distribute his/her calculations among a network of computer like the one sketched in figure 2:

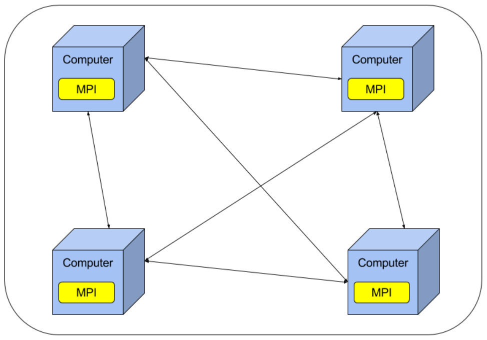


Figure 2: A sketch of a typical *MPI* interconnection.

With the help of *MPI* each computer (called also node) can share resources for use by the programs. From the *MPI* point of view the composition of the computer network (see for example the one given in figure 3) is irrelevant, because what matters is the number of processes that can run concurrently, as illustrated in figure 4 by considering the processes of the previous sketch.

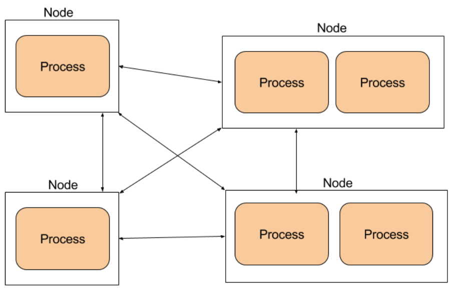


Figure 3: Sketch of *MPI* abstraction over the network.

In doing this, the *MPI* library disregards some hardware constraints (like how many processes compete for the use of the node resources, the characteristics of the network connecting the nodes, etc.) that may still be important in determining the efficiency of the calculation.

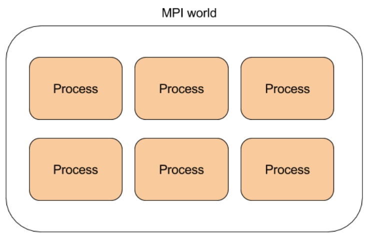


Figure 4: *MPI* basic resources.

This basic *MPI* environment is called *MPI World* in which each process i is marked by an identifier (rank) that is used to create a correspondence among the subsets of matrix elements to be processed in the distributed calculus as indicated in figure 5.

In our work we made use of the *MPI* version *MPICH*[6] and of the functionality that allows to distinguish the considered process from the others both for job submission to the workers and for the collection of their results by the master process (that is, usually the one with the smallest rank (number 0)).

We exploited also the possibility of defining a new type using the command *MPI_Type_create_struct* in which *struct* contains the information needed to send a message to the other processes and tell to the workers the job to run and the exit from the program for all the processes.

Other used *MPI* commands are *MPI_Send* and *MPI_Recv* for the exchange of messages. These commands are blocking and therefore stop the normal flow of the algorithm to wait for the completion of the operation.

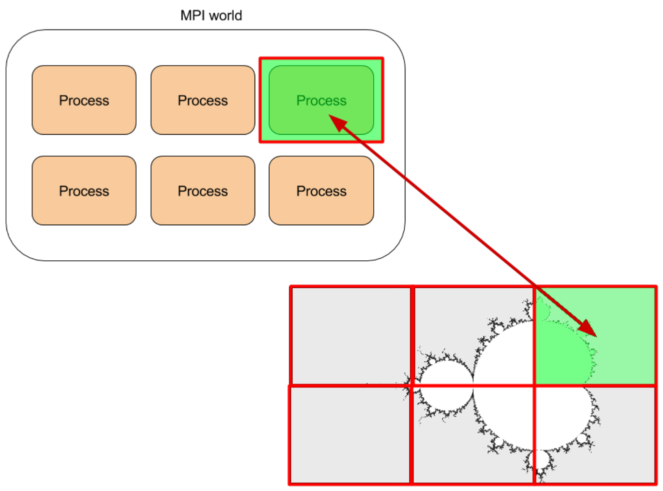


Figure 5: A sketch of how the work is subdivided.

The last two relevant commands are *MPI_Probe* and *MPI_Wtime*, used respectively to check whether there are queued messagese for a process and to take execution times.

Furthermore, the cluster we had available offered us an environment with *qsub*[7] for the management and monitoring of the tasks submitted to the cluster and for this reason we created a little script in Python[8] to simplify the submission of all tests with *qsub*.

3 The problem

The problem tackled in our work is the transformation of the sequential calculation of the Mandelbrot set into a parallel one in which, given the matrix of points defined on a regular grid of values of the real (in the abscissa) and imaginary (in the ordinate) components of c (see the image file in figure 1), it is computed whether a point of the image belongs to it or not.

3.1 Sequential Algorithm

The sequential algorithm calculates the mentioned values starting for example, as in the case shown in figure 6, from the top left hand side corner marked in red and using the grid of points shown there.

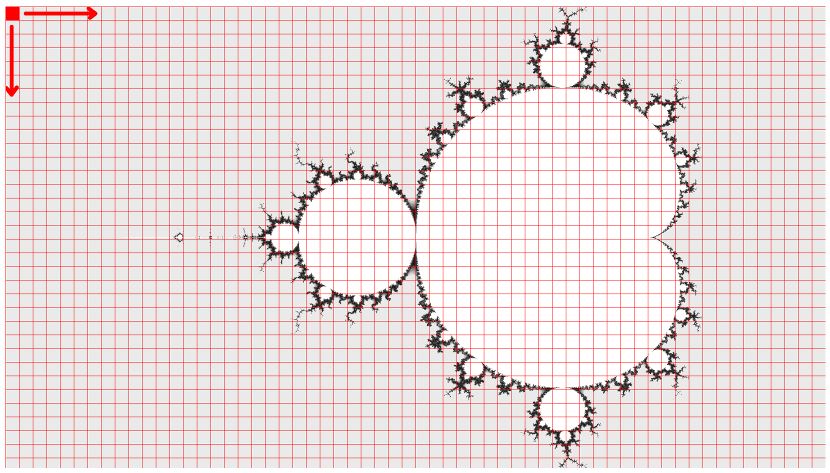


Figure 6: Matrix subdivision of the space to be scanned adopted by the sequential algorithm.

The sequential algorithm is a simple procedure that iterates over the pixels of the matrix and checks whether the considered point belongs to the Mandelbrot set as described in the following pseudo code (Algorithm 1):

Algorithm 1: Sequential.

```
1 function main(iterations, width, height)
2     Initialize MPI
3     start ← MPI_Wtime
4     gen_mandelbrot_set(iterations, width, height)
5     end ← MPI_Wtime
6     Finalize MPI
```

This procedure is executed by running one process on a single node. The *get_mandelbrot_set* function will generate an array of numbers (by default the type is unsigned short) large *width* × *height*. The maximum value of a single number is the value of *iterations* passed to the function.

It is worth pointing out here that, as shown above, also for the sequential algorithm we used some *MPI* functions so as to make more homogeneous the comparison with the parallel algorithms.

In order to implement a parallel version of the Mandelbrot algorithm by exploiting the distributive property of the algorithm 1 the matrix can split into several sub matrices and the evaluation of the Mandelbrot condition for the values of the considered sub matrices is performed concurrently by summing up the individual outcomes the end. This can be performed by adopting two different 2D (two dimensional) distributions of the data by means of the *MPI* library:

- Static Load Balancing (*SLB*) model: given I processes and a $M \times N$ matrix divided in sub matrices of dimension $P \times Q$ each process will have M/P rows and N/Q columns. If the matrix dimensions are not divisible by the number of I processes the reminders of the matrix division could be calculated for simplicity by the edge processes in the matrix grid;

- Dynamic Load Balancing (*DLB*) model: given I processes and a $M \times N$ matrix divided in sub matrices of dimension $P \times Q$ each process will have $K * M/P$ rows and $K * N/Q$ columns. The load of the processes have to be balanced dynamically. The parameter K has been taken in most of our calcualtions to be either 1/4 or 2/4, 3/4 and 4/4 (the latter is the limiting value of $K = 1$ that makes the *DLB* and the *SLB* model coincide).

To manage the communication each implementation has a custom *MPI type* that is used to send jobs and to call the exit of the programs. Each process manages its own buffer and is responsible for cleaning the environment.

3.2 Static Load Balancing Parallel Algorithm

The SLB program divides the matrix and assigns a sub matrix of the image to every process available. The main process (rank 0, the master process) calculates the first sub matrix after sending to each worker the data of its job and collects the results at the end of the calculations in order to construct the final image.

An example of sub matrices assignment is given in figure 7.

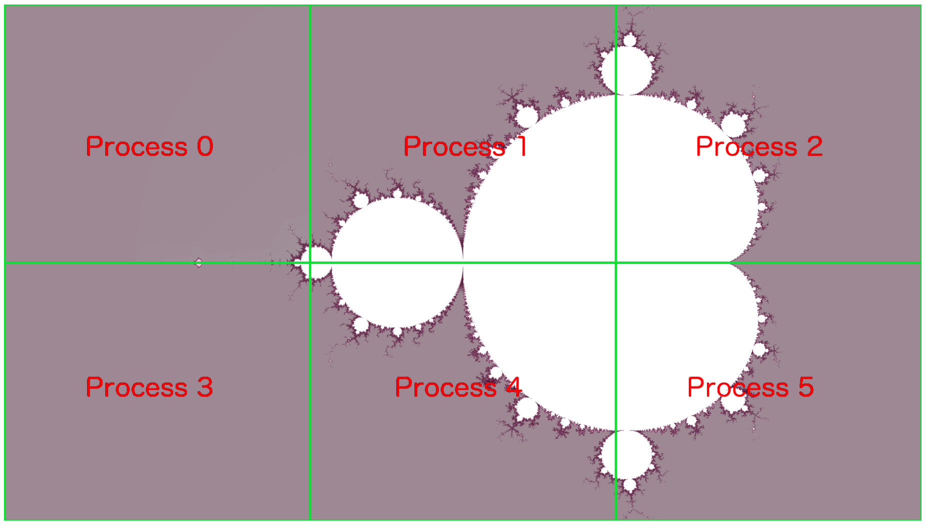


Figure 7: Static subdivision of the work in a 2×3 grid.

The related pseudo code is given in Algorithm 2:

Algorithm 2: Static Load Balancing.

```
1  function main($iterations, width, height, N, M$)      ▷ cell division $= N \times M$
2      Initialize MPI
3      Create message type and model
4      if $rank == 0$ then                                  ▷ Master, process with rank 0
5          Calculate subdivision of the matrix
6          $start \leftarrow$ MPI_Wtime
7          Send jobs to each process
8          gen_mandelbrot_set()                             ▷ part of the master
9          Receive the results from each process
10         Finalize final matrix
11         $end \leftarrow$ MPI_Wtime
12     else                      ▷ Workers, processes with rank $i$ :   $i = 1..(N \times M) - 1$
13         Receive my job
14         gen_mandelbrot_set()                             ▷ part of worker $i$
15         Send result
16     Finalize MPI
```

Figure 7 shows that the workload for the various jobs may differ. Accordingly, some processes may have tasks smaller than those of others (with a consequent negative impact on the performance).

3.3 Dynamic Load Balancing Parallel Algorithm

In the *DLB* approach the main process (rank 0) has only a managerial task and it assigns to the available processes their K dependent tasks (see Figure 8). Because of the dynamical nature of the approach the main process analyzes the incoming messages and assigns the next job to perform to an already ended process. After the completion of all tasks the main process recollects the remaining pieces of information as soon as they become available from the processes that are still busy.

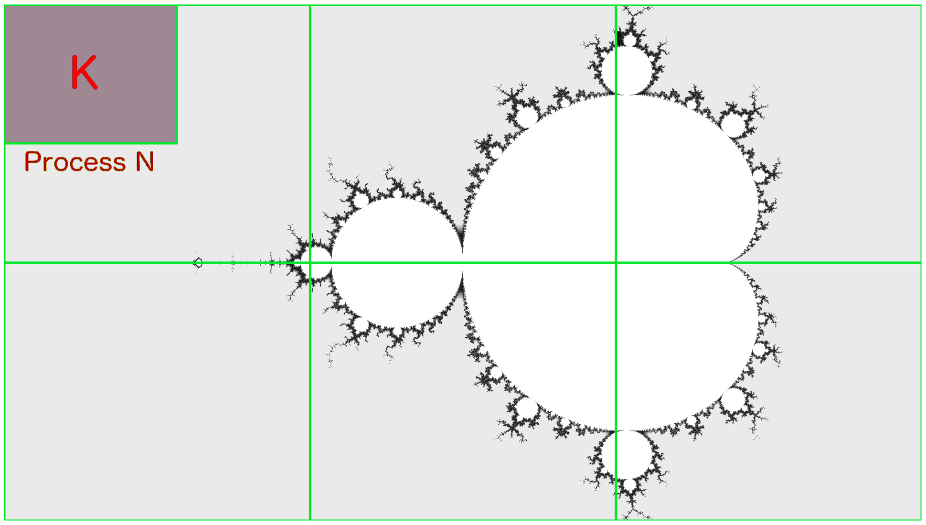


Figure 8: A dynamic assignment of the work in a 2×3 grid.

When $K = 1$, the assignment process is the same as in the *SLB* algorithm. The variation of the K factor is expected to change significantly the performances.

The pseudo code of this algorithm is given in algorithm 3.

Algorithm 3: Dynamic Load Balancing.

```
function main(iterations, width, height, N, M, K)    ▷ cell division= $N \times M$, $K$ = fraction
    Initialize MPI
    Create message type and model
    if $rank == 0$ then                                  ▷ Master, process with rank 0
        Calculate subdivision of the matrix
        $start \leftarrow$ MPI_Wtime
        forall parts of the subdivision do
            if worker available then
                Send job to worker
            else
                while no worker available do
                    Check incoming messages with a probe
                    if message incoming then
                        Collect result of that worker
                        break
        while there is a worker busy do
            Check incoming messages with a probe
            if message incoming then
                Collect result of that worker
        $end \leftarrow$ MPI_Wtime
        Send to all worker the command to exit
    else                     ▷ Workers, processes with rank $i$ :   $i = 1..(N \times M) - 1$
        while no exit command do
            Receive job
            gen_mandelbrot_set()                        ▷ Current part to process
            Send result
    Finalize MPI
```

4 Performance tests

The first performance tests were run on the *32 bit* environment of platform A [3]. The smallest size of the matrix considered for the calculations was of 10000 (100×100) pixels while the largest one was of 163840000 (12800×12800) pixels. We set also a maximum of 10000 for the iterations and partitioned the matrix among 16 processes as 4×4.

4.1 Elapsed times

Figure 9 shows measured elapsed times given in second. As is apparent from the figure, the parallel approach is more efficient than the sequential one with the elapsed time measured for the *SLB* model (see Algorithm 2) for 16 processes being on the average more than 3 times smaller.

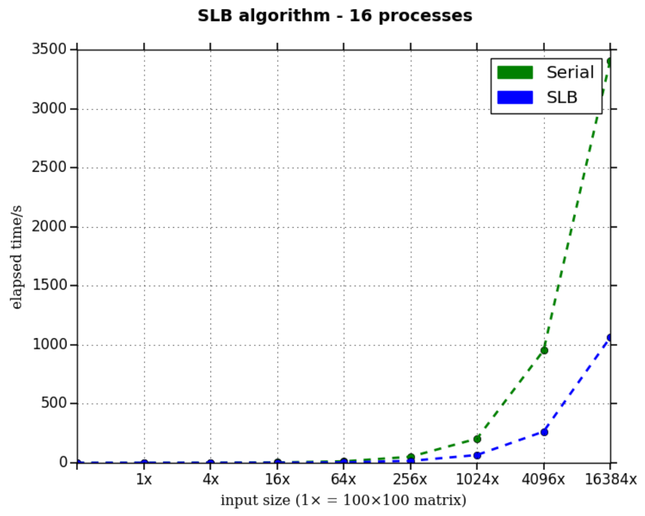


Figure 9: Elapsed time for the serial (green dashed line) and the parallel *SLB* (blu dashed line) calculations performed using 16 processors.

As a double check we performed also the same calculation using the *DLB* algorithm 3 with $K = 1$. As shown in figure 10, the performances obtained are coincident with those of the *SLB* algorithm 2.

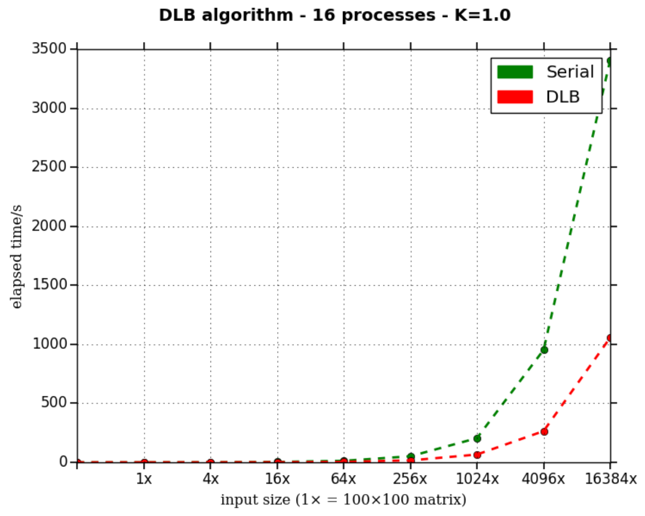


Figure 10: Elapsed time for the serial (green dashed line) and the parallel *DLB* (red dashed line) calculations performed using $K = 1$ and 16 processors.

In order to better exploit the potentialities of the *DLB* algorithm, we carried out further runs by setting $K = 1/4$. Computed elapsed times are plotted again against those of the serial run in figure 10. As apparent from the figure, there is a gain in time due to the fact that in the *DLB* approach the processes operate on various sections of the matrix as soon as they become available. As a matter of fact, the time saving with respect to the *SLB* algorithm is a factor of about 14.

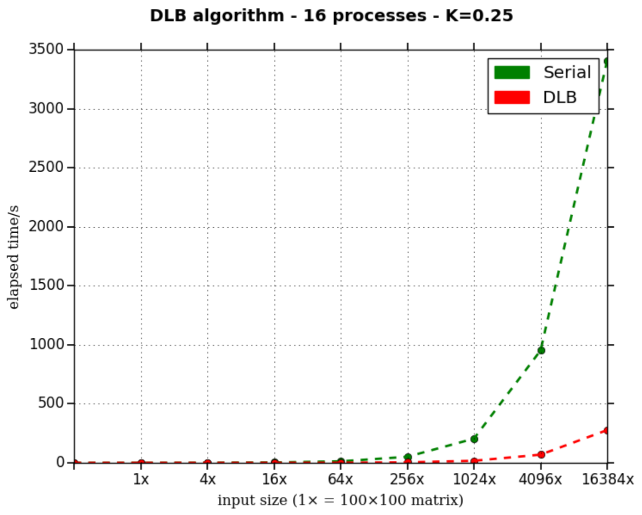
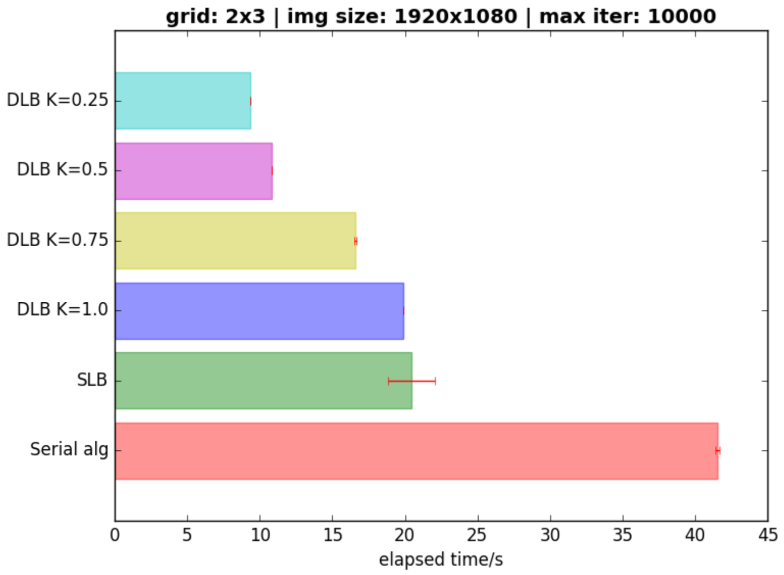


Figure 11: Elapsed time for the serial (green dashed line) and the parallel *DLB* (red dashed line) calculations performed using $K = 1/4$ and 16 processors.

To further check the impact of the subdivision of the image, we tested a fixed matrix resolution and changed the grid subdivision. The image size was 1920×1080 and we illustrate the results obtained for both the 2×3 (upper panel) and 3×2 (lower panel) cases in figure 12.

As shown in the figure, this impacts the performance of the code that varies significantly for both the *DLB* and the *SLB* algorithms as a function of the grid configuration. As a result, the elapsed time of the dynamic approach while it is clearly smaller than that of the static one for the 2x3 case, it is similar for the 3×2 one.



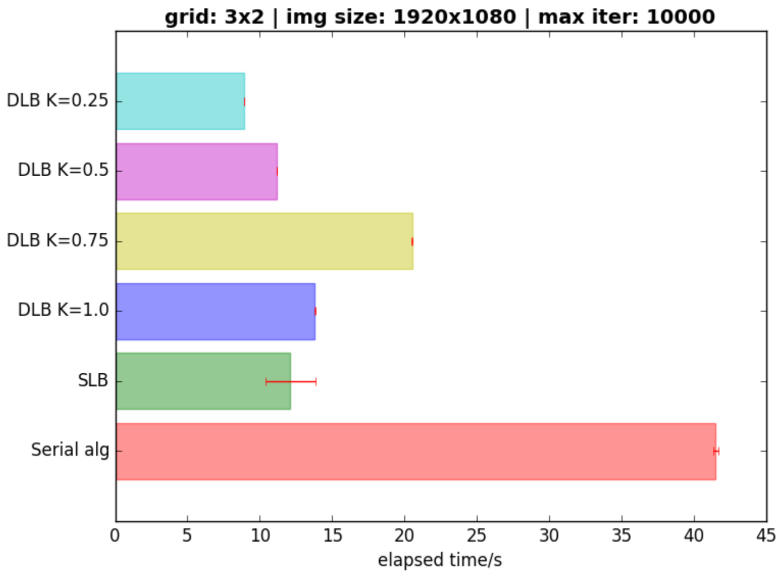


Figure 12: Elapsed time for different choices of the value of K for the DLB algorithm compared with that of the SLB one respectively for a 2×3 (upper panel) and a 3×2 (lower panel) partitioning

5 Speedups

In order to put on a more rigorous basis the criteria to be adopted when partitioning the image to the end of optimizing the performance of the two algorithms, we caculated the speedup (i.e. the ratio between the elapsed time associated with the scalar and that associated with the parallel execution) as a function of the number of processes used.

These measurements were performed on the 64 bit configuration machines of platform B[4]. On this platform the sequential algorithm performed better than on platform A (the elapsed time measured on platform B is 2646.224791 second while that measured on platform A is 3408.752897 second).

For these calculations the size of the image matrix involved was 12800×12800 pixels while the number of iterations was left unaltered. As shown in figure 13, the range of processes used varied from 2 to 20 using for the *DLB* algorithm $K = 1/4$ (that showed to be the best partitioning in the previous tests).

On the average, the *DLB* algorithm shows to perform better that the *SLB* one whose speedup never exceeds 4 regardless of the number of processes used.

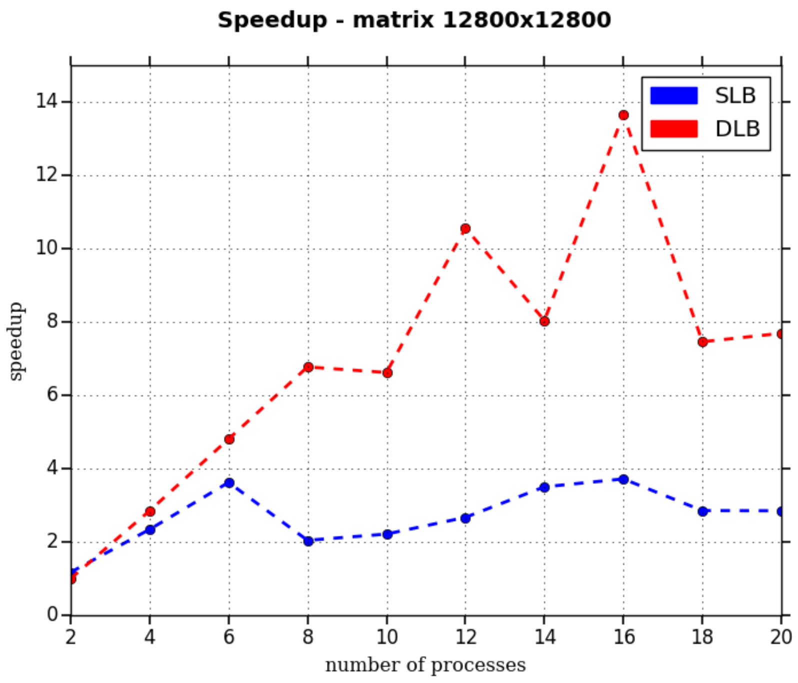


Figure 13: Speedup measured for the *SLB* algorithm (blue line) and the *DLB* one (red line)

The plots of figure 13, however, show a highly structured shape of the speedup plot. That was highly surprising when considering that the *DLB* algorithm is the most efficient one. A rationale for that was found in the fact the matrix subdivision associated with the different number of processes may vary significantly from one subdivision to another and result in some cases quite unbalanced.

To better understand the impact of the assignment of the sub matrices to the different processes we investigated the effect of adopting a more flexible partitioning cryterion by allowing the submatrices to become single row and/or single column. To speed up the calculations the size of the image matrices was reduced to 1600x1600.

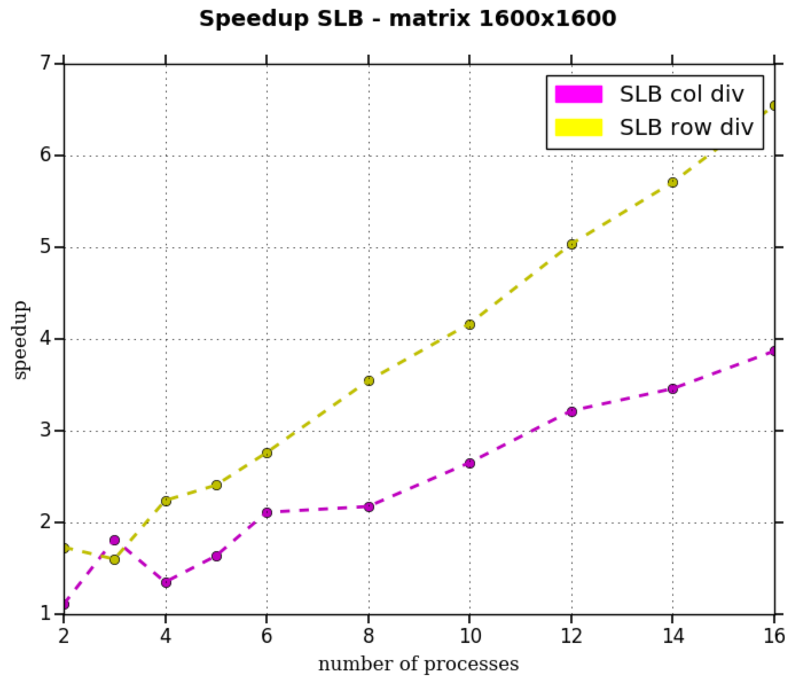


Figure 14: Speedup measured for the *SLB* algorithm by distributing either single columns (yellow line) or single rows (violet line)

The speedup values plotted for the SLB algorithm, which show to be almost double those for the SLB algorithm applied to rows totalling about 6.5 for 16 processes (see figure 14), agree with the fact that the columns of the Mandelbrot set differ on the average more than the relative rows which have a more mixed (belonging and not belonging to the set) nature.

This feature of the rows and columns of the Mandelbrot set is irrelevant for the DLB algorithm that can assign further work as soon as a process is completed. This is confirmed by the substantial agreement of the speedup values computed when applying the DLB algorithms to rows and columns (see figure 15).

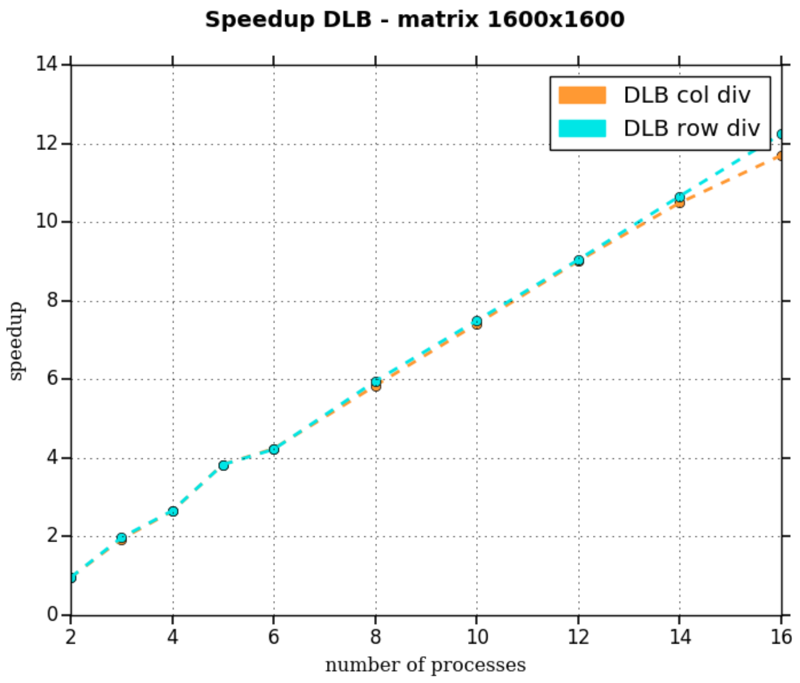


Figure 15: Speedup measured for the *DLB* algorithm by distributing either single columns (blue line) or single rows (green line)

A final comparison of the performances of the SLB and DLB algorithms has been carried out by evaluating the speed up for the single elements case whose results are given in figure 16.

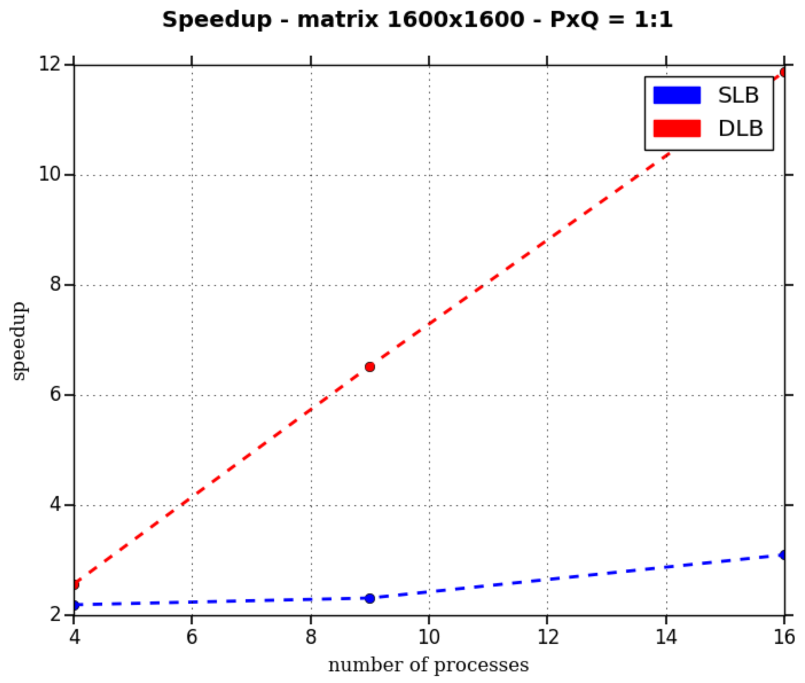


Figure 16: Speedup measured for the *DLB* (red) and the *SLB* (blue) single element algorithm

As shown by the figure while the SLB model is completely inadequate for dealing with such an atomistic approach to the parallel calculation of the Mandelbrot set, the DLB one is able to reach a speedup of 12 for 16 processes.

6 Conclusions and ongoing work

The main result of the tests carried out by us is that the division of the grid on which evaluate the Mandelbrot image affects significantly the measured parallel performances of the calculation. We found, in fact, that the relative elapsed time depends on the way the matrix is partitioned among the processes because the various areas of the image matrix have a different density of calculations and, therefore, require more computing effort than others.

The second result is that the dynamic algorithm copes better with uneven distributions (as is inevitable in cases like the Mandelbrot image whose determination is purely numerical and weakly predicatable) than the static one when using the same number of processes.

This was found to depend from the fact that smaller jobs cope better with the dynamical algorithm because the subdivision of the image among the various processes may differ significantly in terms of compute demand.

Furthermore, we found, that is in any case important to reduce the size of the submatrices in the *DLB* approach (while it is not in a *SLB* one) because uneven distributions are better compensated by the dynamic algorithm as shown by the almost ideal linear increase of the speedup in this case. In this view, further investigations are being performed using an approach of the Monte Carlo type (that is closer to the *DLB* one).

References

[1] *Weisstein, Eric W.* "Mandelbrot Set." From *MathWorld*–A Wolfram Web Resource. `http://mathworld.wolfram.com/MandelbrotSet.html`

[2] *Scientific linux.* `https://www.scientificlinux.org/`

[3] *Accessed, Gen-Apr 2016.* `http://cgcw.herla.unipg.it`

[4] *Accessed, Gen-Apr 2016.* `http://fecw.herla.unipg.it`

[5] *Tanenbaum, Andrew S*, "MPI - Interfaccia a scambio di messaggi, in Architettura dei calcolatori. Un approccio strutturale", *Milano*, *Pearson Education*, 2006, pp. 610-613, ISBN 978-88-7192-271-3.

[6] *MPICH — High-Performance Portable MPI.* `https://www.mpich.org/`

[7] *Qsub, man page.* `http://gridscheduler.sourceforge.net/htmlman/htmlman1/qsub.html`

[8] *Python programming language.* `https://www.python.org/`