Universitat Politècnica de Catalunya

Parallelism

# Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

*Carlota Catot Bragós*
*Ferran Martínez Felipe*
PAR1110

Quadrimestre Primavera 2017-2018

# Contents

# 1 Introduction

The main application of the parallelization methods developed until this days are based in improve the performance of nearly all the applications that can be computed using classical computer architectures. However, there are different ways to parallelize the same application, based on different ideas.

In our case the application to study is the Mandelbrot set generation, in which we are going to play with the task granularity and different parallelization methods.

The Mandelbrot set is a particular set of points, in the complex domain, whose boundary generates a distinctive and easily recognizable two-dimensional fractal shape. To obtain information about how the Mandelbrot set is calculated, please visit this link of Wikipedia:

http://en.wikipedia.org/wiki/Mandelbrot_set

To generate the Mandelbrot Set we have to color the points that don't belong to the set. This coloring, in order to make the problem feasable, is done by limiting the maximum number of steps that the code does to calculate if a number belongs to the Mandelbrot Set. If the limit is reached, the code considers that the point belongs to the Mandelbrot set.

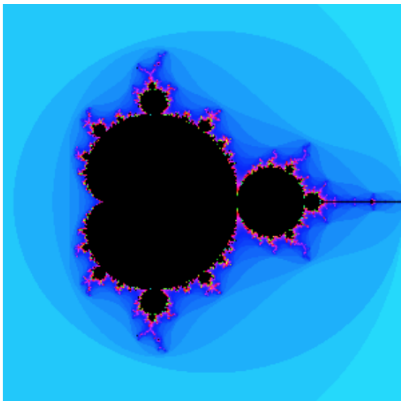


Figure 1: Mandelbrot Set

In order to study the different parallelization methods, the first that we are going to do is study the different granularities in the task generation. After that, we are going to study the task generation for task, taskloop and for methods.

## 2 Task granularity analysis

In this section we are going to talk about task granularity in mandel-tareador version, with -w 8. We will see de difference and the common parts with the task graphs generated for the task granularities Row and Point, in each case with 8 tasks.
In the other hand we will try to do the code more parallel after analyzing the serial part of it.

In the fist part, we execute the mandle-tareador in the non-graphical version, we extract the dependence graph of both of, in figure 2 we can see the Row task graph and in the figure 3 we can see the Point task graph. We can observe and compare both of it and extract what we think that are the 2 most imporant common characteristics of the dependance graph.

Common caractheristics after analysing the images:

- There is no dependences between task in none of the dependency graphs, we can see in the figures bellow, we see all the tasks (green nodes) have the dependence on the main loop (blue node).
- In the second point, we can see that not all the tasks have the same amount of work, we can see that in the two cases, the nodes in the middle have more granularity than the tasks in the extremes.

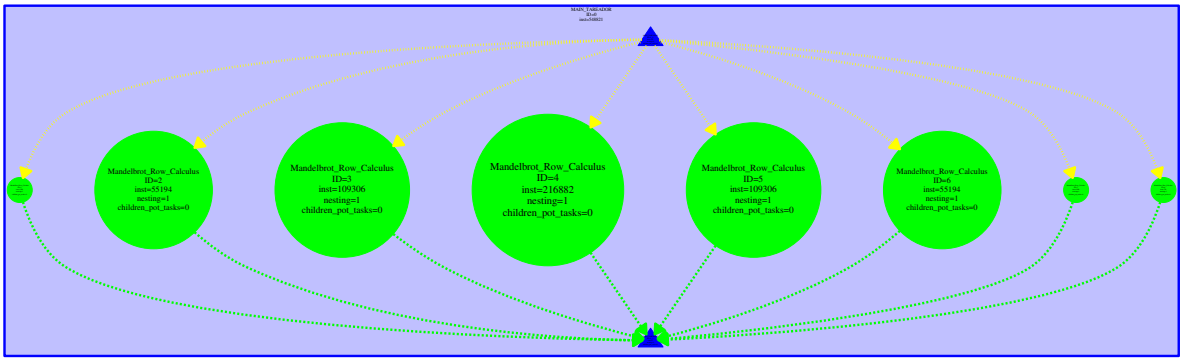



Figure 2: Row task graph

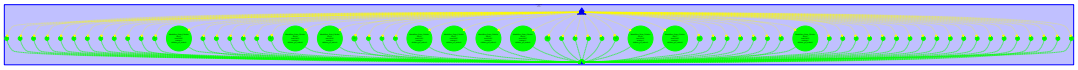


Figure 3: Point task graph

In the second part, we analize the code to find the code that cause the serialization of tasks.

```
/* Scale color and display point  */

long color = (long) ((k-1) * scale_color) + min_color;
if (setup_return == EXIT_SUCCESS) {
    XSetForeground (display, gc, color);
    XDrawPoint (display, win, gc, col, row);
}
```

If we have this part uncomented the tasks will be executed in the secuential way. If we want to protect this section of code in the parallel OpenMP code, we should use the *#pragma omp critical* directive to define a region of mutual exclusion where only one thread can be working at the same time.

# 3 OpenMP task–based parallelization

In this section we are going to parallelize the code which generates a Mandelbrot Set by using a task strategy. Specifically, we are going to do two kinds of parallelization: One based on parallelizing the code by *Rows* and another one parallelizing the code by *Points* (being the rows and points positions of the matrix generated to calculate the Mandelbrot set).

## 3.1 Row Decomposition

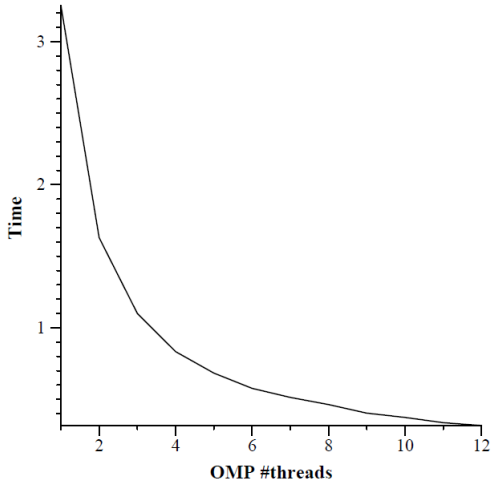
We are going to start talking about the *Row* parallelization. To parallelize by rows, we have used the following pragma before the two for loops in code lines 92 - 96 and also added a pragma after the first for (to check all the code changes, please visit the code *mandel-omp-task-row.c* attached with this report):

```
#pragma omp parallel
#pragma omp single
//#pragma omp for schedule(runtime)
for (row = 0; row < height; ++row) {
    #pragma omp task firstprivate(row) private(col)
    for (col = 0; col < width; ++col) {
    }
}
```

Figure 4: Code to parallelize based on task and row strategy

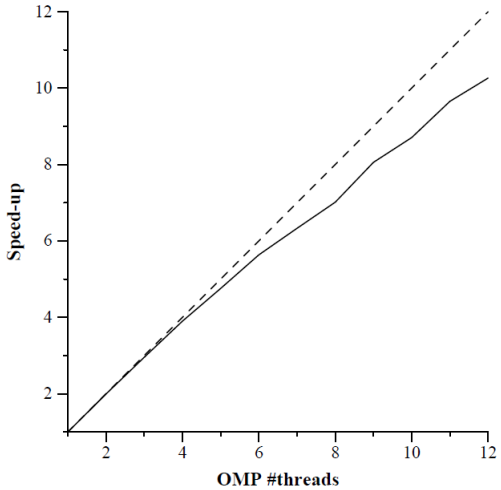
The idea behind this parallelization is to generate tasks that make a set of iterations of the extern loop (and, consequently, executing the inner loop from 1 to height times, depending of the gransize). The private clause exists in order to avoid problems of data races, and the firstprivate is because it's necesary to have the old value.

The following plots gives the Time and the Speed-Up obtained for a different number of Threads:




Average elapsed execution time

Figure 5: Time vs. Threads




Speed-up wrt sequential time

Figure 6: Speed-Up

As we can see this grainsize scales very well with the number of threads used (fact that we could expect for the dynamic grainsize). If we look to the plot of *Time vs. Threads* we can see that the execution time decreases logarithmically with the number of threads, as the speedup increases in a sub-optimal way (it increases linearly, but with a pendant lower than 1).

However, and as a final point, if we look to the speed-up plot for 12 threads we can see that the speed-up has a bigger decrease. With the number of threads used is impossible to point out that if this decrease is due to the overhead produced by the parallelism or by fluctuations of the Boada cluster.

## 3.2 Point Decomposition

The next parallelization to study is the *Point* parallelization. To parallelize by points, we have used the following pragma before the two for loops, and a pragma after the two loops in code lines 92 - 97 (to check all the code changes, please visit the code *mandel-omp-task-point.c* attached with this report):

```
#pragma omp parallel
#pragma omp single
//#pragma omp for schedule(runtime)
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
    #pragma omp task firstprivate(row, col)
    { }
    }
}
```

Figure 7: Code to parallelize based on task and point strategy

The idea behind this parallelization is to generate tasks that make a set of iterations of the inner loop (and then, depending of the grainsize, doing the calculus of one or multiple points). The firstprivate clause exists in order to avoid problems of data races, and in this case the change from private to firstprivate is due to the variable row is initialized before the pragma execution.

The following plots gives the Time and the Speed-Up obtained for a different number of Threads:

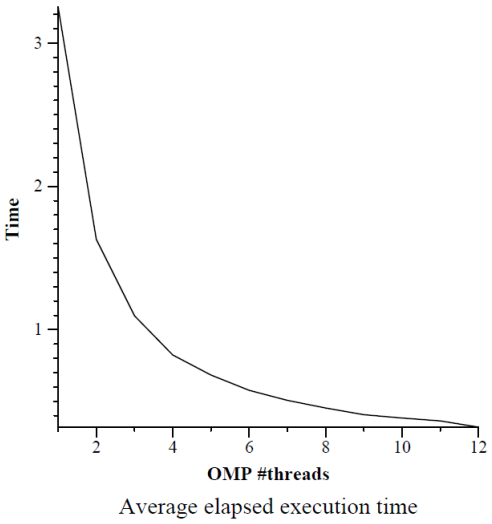



Figure 8: Time vs. Threads

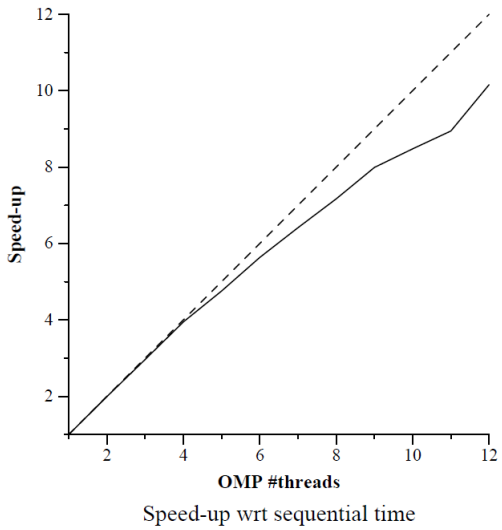



Figure 9: Speed-Up

As we can see this task strategy has similar results than the previous one. The Time vs Threads plot shows that with this strategy the time decreases in a similar way as in the previous one. Nevertheless, the Speed-Up plot shows that this strategy doesn't have the sudden decrease that have the previous strategy. That difference can mean two things: This strategy is better than the previous one (have less overhead) or that the old strategy have the decrease due to Boada perturbations (this is probably the correct answer, because this strategy generates more tasks than the previous one, and because that this task strategy has more overhead).

# 4 OpenMP taskloop–based parallelization

In this section we are going to parallelize the code which generates a Mandelbrot Set by using a taskloop strategy. Specifically, we are going to do two kinds of parallelization: One based on parallelizing the code by *Rows* and another one parallelizing the code by *Points* (being the rows and points positions of the matrix generated to calculate the Mandelbrot set).

## 4.1 Row Decomposition

We are going to start talking about the *Row* parallelization. To parallelize by rows, we have used the following pragma before the two for loops in code lines 96 - 97 (to check all the code changes, please visit the code *mandel-omp-taskloop-row.c* attached with this report):

```
int num_threads = omp_get_num_threads();
#pragma omp parallel
#pragma omp single
//#pragma omp for schedule(runtime)
#pragma omp taskloop grainsize(height/num_threads) private(col)
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
    ...
    }
}
```

Figure 10: Code to parallelize based on taskloop and row strategy

The idea behind this parallelization is to generate tasks that make a set of iterations of the extern loop (and, consequently, executing the inner loop from 1 to height times, depending of the gransize). The private clause exists in order to avoid problems of data races.

After trying different constant values for the grainsize, we have decided that the best approach to deal with the task generation is to divide the loop using the height and the number of threads as a parameter. This election is done due to the fact that choosing a constant grainsize doesn't scale as well as this division as we increase the number of threads (but for the number of threads used we can obtain similar results with a grainsize of around 8). With this election the number of iterations of the outer loop are divided is tasks such that we have as many tasks as threads and the sum of the iterations of each tasks gives the total iterations.

The following plots gives the Time and the Speed-Up obtained for a different number of Threads:

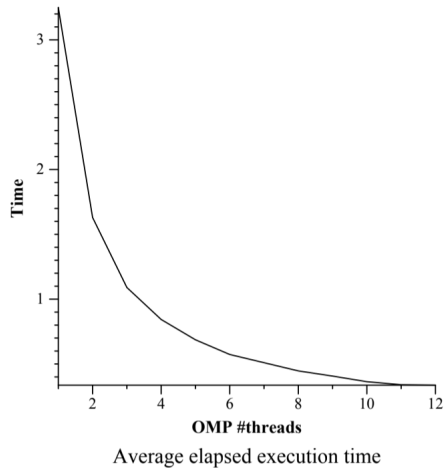



Figure 11: Time vs. Threads

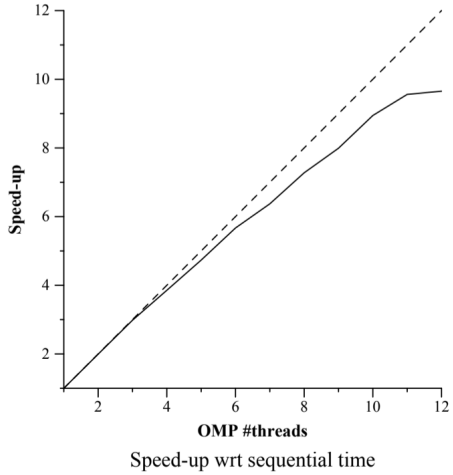



Figure 12: Speed-Up

As we can see this grainsize scales very well with the number of threads used (fact that we could expect for the dynamic grainsize). If we look to the plot of *Time vs. Threads* we can see that the execution time decreases logarithmically with the number of threads, as the speedup increases in a sub-optimal way (it increases linearly, but with a pendant lower than 1).

However, and as a final point, if we look to the speed-up plot for 12 threads we can see that the speed-up has a bigger decrease. With the number of threads used is impossible to point out that if this decrease is due to the overhead produced by the parallelism or by fluctuations of the Boada cluster.

## 4.2 Point Decomposition

The next parallelization to study is the *Point* parallelization. To parallelize by points, we have used the following pragma between the two for loops in code lines 95 - 97 (to check all the code changes, please visit the code *mandel-omp-taskloop-point.c* attached with this report):

```
int num_threads = omp_get_num_threads();
#pragma omp parallel
#pragma omp single
//#pragma omp for schedule(runtime)
for (row = 0; row < height; ++row) {
  #pragma omp taskloop grainsize(width/num_threads) firstprivate(row)
    for (col = 0; col < width; ++col) {
    ...
    }
}
```

Figure 13: Code to parallelize based on taskloop and point strategy

The idea behind this parallelization is to generate tasks that make a set of iterations of the inner loop (and then, depending of the grainsize, doing the calculus of one or multiple points). The firstprivate clause exists in order to avoid problems of data races, and in this case the change from private to firstprivate is due to the variable row is initialized before the pragma execution.

As in the row decomposition, after trying different constant values for the grainsize, we have decided that the best approach to deal with the task generation is to divide the loop using the width and the number of threads as a parameter. This election is done due to the fact that choosing a constant grainsize doesn't scale as well as this division as we increase the number of threads (but for the number of threads used we can obtain similar results with a grainsize of around 4-8). With this election the number of iterations of the inner loop are divided is tasks such that we have as many tasks as threads and the sum of the iterations of each tasks gives the total iterations.

The following plots gives the Time and the Speed-Up obtained for a different number of Threads:

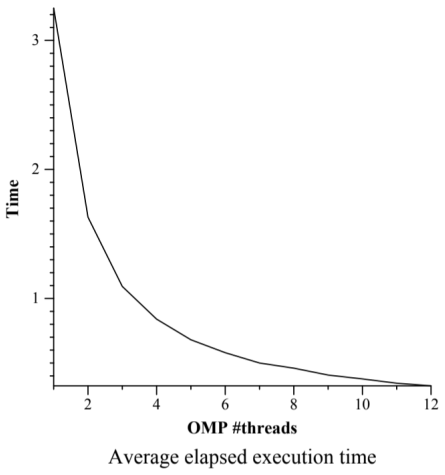



Figure 14: Time vs. Threads

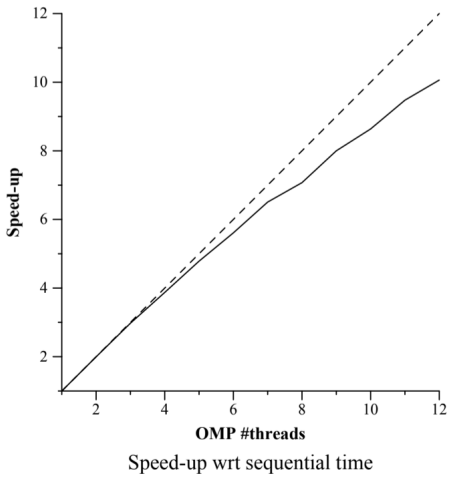



Figure 15: Speed-Up

As we can see this task strategy has similar results than the previous one. The Time vs Threads plot shows that with this strategy the time decreases in a similar way as in the previous one. Nevertheless, the Speed-Up plot shows that this strategy doesn't have the sudden decrease that have the previous strategy. That difference can mean two things: This strategy is better than the previous one (have less overhead) or that the old strategy have the decrease due to Boada perturbations (this is probably the correct answer, because this strategy generates more tasks than the previous one, and because that this task strategy has more overhead).

# 5 OpenMP for–based parallelization

The schedule of the for directive in OpenMP decides how the work is going to be distributed along the threads. There are 3 types of schedule:

- **static:** In this schedule the tasks that each thread is going to do is fixed from the beginning (being the nunmber of tasks equal for all the threads).
- **dynamic:** In this schedule each of the tasks generated is going to be done by the first free thread.
- **guided:** In this schedule the number of tasks executed for each thread decreases in order to try to reach a load balance

In this section we are going to study the behaviour of the Mandelbrot set for each of the schedules explained in the previous paragraph. To do so, we have executed the Mandelbrot Set code by changing the schedule (but with the same number of threads, that was 8) in order to compare the differences between each schedule.

Next we are going to show the plots obtained by executing each of the schedules:

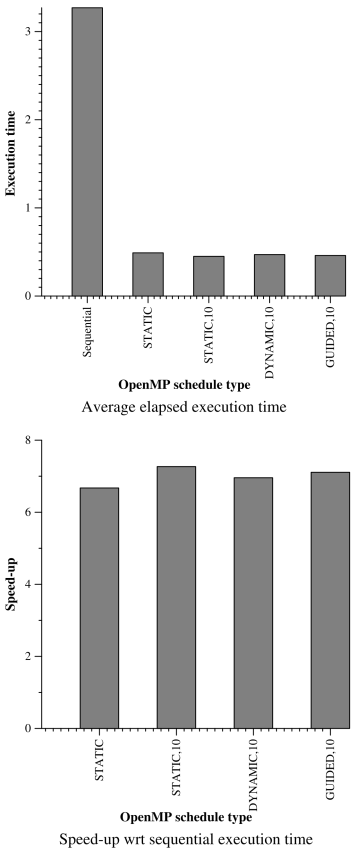



Figure 16: Speed-Up Row

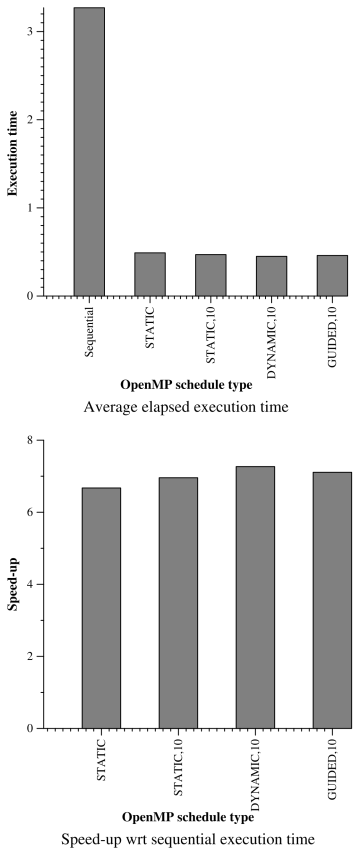



Figure 17: Speed-Up Point

As we can see from the plots, the results of the *Row* and *Point* strategies are quite similar (which was explained in the taskloop section). In addition, we can observe the following:

- **static:** This schedule shows a little worse results than the other ones, due to the fact that the same number of threads is given to each of the threads, provoking load unbalance between threads. However, the expected results for this schedule were worse than the shown, because usually the load unbalance is higher in this schedule than the shown in this plot.
- **static, 10:** This schedule shows the better results in the row strategy, and one of the best in the point strategy. This fact doen't make sense at all, because the expected is that schedules that dynamically distribute the work between threads should obtain better results than the ones with a fixed distribution of work between threads (but better results than the static schedule because the chunk size is bigger).
- **dynamic, 10:** As soon as one thread finishes the work to do, this schedule assign another task to the thread. This behaviour should achieve one of the best results, but the expected behaviour was to obtain a bigger difference in speedups between static and dynamic,10 due to load unbalance.
- **guided, 10:** As the dynamic schedule, this schedule should achieve one of the best results, but it's similar than the static (and that should not be the case). The explanation is similar to the dynamic case.

For the *Row* strategy, using *Paraver* and *Extrae* we obtained the following results (in which all the times are expressed in nanoseconds):

| | static | static, 10 | dynamic, 10 | guided, 10 |
|---|---|---|---|---|
| Running average time per thread | 482.404.904 | 487.531.429 | 492.109.414 | 478.167.217 |
| Execution unbalance | 0.69 | 0.59 | 0.7 | 0.7 |
| SchedForkJoin | 782.112 | 772.597 | 1.151.777 | 1.782.320 |

We can see that as explained in the previous paragraphs, the results obtained by each of the schedules are really similar. That results, which as we can see in the plots gives a big speedup in comparision with the sequential version, doesn't show clearly the difference between each of this schedules (in our opinion the differences between schedules should be bigger).

# 6 Conclusions

In the real world, the overheads produced by task generation/synchronization provoke that not always the higher number of tasks generated achieves the better results.

The main goal of this lab session was to check how the different task generation strategies and grainsize can achieve different results.

After doing the first approach to parallelize the code using *Tareador*, we could observe that the sequential code can be distributed into different tasks of different size (for the *Row* and *Point* strategies).

After the first approach we could observe that different task generation strategies can be done in order to parallelize the code with the same results (in this case using the pragmas task, taskloop and for). In addition, we obtained similar results for the *Row* and *Point* strategies, which apriori doesn't make sense because the *Point* strategy has an orders of magnitude higher overhead than the *Row* strategy.

Finally we have to say that we could observe that the different for schedules have obtained similar results, that it isn't what our intuition predicted (we thought that static schedule should give worst results than the dynamic and guided ones, due to work unbalance between threads).