UPC: Facultat de Informàtica de Barcelona

Lab 3 - Embarrassingly parallelism with OpenMP: Mandelbrot set

PAR Laboratory Assignment

Oriol Moyano Núñez
Martí Ferret Vivó

ID: par2108

24/04/2018

Index

Introduction

The focus of this document is the study of an embarrassingly parallel problem; observing the possible parallelization opportunities it presents, analyzing the overheads it produces and extracting conclusions about the whole process. To this end, first we need to understand some basic concepts, the problem we will be working on and the parallelization strategies we will study.

An embarrassingly parallel problem (also known as a perfectly parallel or pleasingly parallel) is one where there is little to no difficulty when it comes to dividing the problem into different tasks. Usually this happens when there is no need for parallel tasks to communicate and the number of dependencies is minimal (more often than not there aren't any at all). The study of these problems is relevant because it provides a great example of the power of parallelism (it is seen as the best possible case) and it helps to understand basic concepts of this field without worrying about dependencies or inter-task communication. More information about this topic can be found in the web pages linked in our bibliography.

Mandelbrot set

The case study we will be analyzing in this document is the computation of the Mandelbrot set for a fixed window size. This set describes a particular set of points in the complex space whose boundary generates an easy recognizable two-dimensional shape. This problem can be parallelized without any difficulty because each point of the set is calculated individually (there are not any dependencies). Let's try to understand now how we can determine whether or not a point belongs to the Mandelbrot set:

- Given a point c of the complex space, if the absolute value of the number resulting from the application of the recurrence below doesn't exceed a certain value, the initial point c belongs to the set:

$$z_{n+1} = {z_n}^2 + c,\ z_0 = 0 \Rightarrow if\ (z_n < value, n \to \infty)\ c \in Mandelbrot\ set$$

The program we will be tinkering with plots the Mandelbrot set using the following rule: the color of each point c corresponds to the number of steps max for which $|z_{max}|^2 \geq 4$. For obvious reasons there is a restriction on the number of steps (otherwise the computation would not finish) and we assume that, if a point gets to this maximum number, it belongs to the Mandelbrot set. As stated before, more information about the problem can be found on the pages linked on the last section of the document.

Parallelization strategies

To understand the considered parallelization strategies, first we will briefly analyze the sequential code without any parallel constructs (the code sections irrelevant to the parallelization of the program will be overlooked, and only explained in words to avoid unnecessary information).

```
void mandelbrot(...)
{
    // Computes the color of each point of the screen (height, width) using the
Mandelbrot set formula
    //------------------R1----------------------//
    for (row = 0; row < height; ++row) {
        for (col = 0; col < width; ++col) {
        //----------------------------------------//
                            complex z, c;

                            z.real = z.imag = 0;

                            /* Scale display coordinates to actual region  */

                            /* Calculate z0, z1, .... until divergence or maximum
iterations */
                            /* Scale color and display point  */
                            //------------------R2----------------------//
                            long color = (long) ((k-1) * scale_color) +
min_color;
                            if (setup_return == EXIT_SUCCESS) {
                                  XSetForeground (display, gc, color);
                                  XDrawPoint (display, win, gc, col, row);
                            }
                            //----------------------------------------//
        }
    }
}
```

Most of the program is already commented above but there are a couple of relevant regions marked as R1 and R2 that deserve special attention if our objective is to parallelize effectively and correctly this code:

- R1: both loops will be the main focus of this document because the two proposed strategies revolve around iteration parallelization.
- R2: it's a region that contains methods to draw on the screen that can not be used simultaneously by two different threads, thing that we will have to ensure when we make the parallel versions of the code.

The parallelization strategies proposed are the following:

1. **Row:** there is a task for each row of the image (a task corresponds to the computation of a whole row of the Mandelbrot set)
2. **Point:** there is a task for each point of the image (a task corresponds with the computation of a single point of the Mandelbrot set).

In the following sections we will study among other things, the effects each strategy has on the performance of the program and the different parallelization techniques that OpenMP provide us to obtain a parallel code.

Performance evaluation

Analyzing potential parallelism with Tareador

First we will analyze the potential parallelism the two proposed strategies have. In order to accomplish this, Tareador will be used (tool to study the time and load of different tasks before the program is truly parallel). The code used the get the result we will comment below is the following:

1. **Row:** decomposition into tasks using tareador, one task per row granularity:

```
  /* Calculate points and save/display */
       for (row = 0; row < height; ++row) {
           tareador_start_task("Row_task_loop");
           for (col = 0; col < width; ++col) {
              ...
           }
           tareador_end_task("Row_task_loop");
       }
```

2. **Point**: decomposition into tasks using tareador, one task per point granularity:

```
/* Calculate points and save/display */
       for (row = 0; row < height; ++row) {
           for (col = 0; col < width; ++col) {
               tareador_start_task("Point_task_loop");
               ...
              tareador_end_task("Point_task_loop");
           }
       }
```

Tareador also indicates the dependencies between task which is useful to know if some part of the code needs a critical section to maintain the correctness of the program. In the figures below we can see the results obtained by executing Tareador (with the codes above) with a window of 8x8:

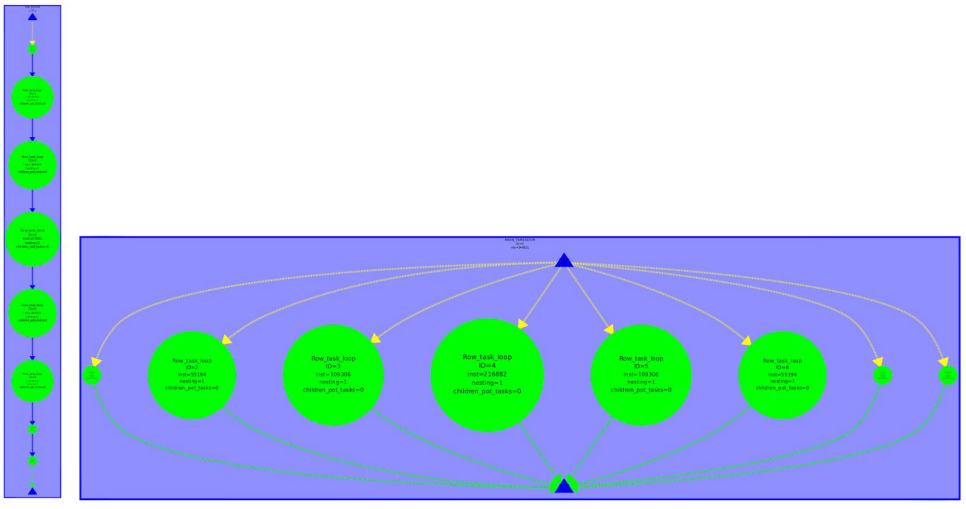

Figure 1 and 2. *Results of Tareador using the Row parallelization with an 8x8 window, the image on the left is the one corresponding to the code with display while the one on the left doesn't have those methods activated.*

We can see clearly that the method to print the dots on the display can not be parallelized so all the tasks have to be processed in a sequential manner. However, when the results are not printed on a window, the task can easily be parallelized, in this case 8 tasks (1 task for each row). Despite this, as we can see in the picture, the work between tasks is not equal, that is because each point of the Mandelbrot set doesn't take the same amount of time to be calculated (as more white a dot is, more iterations it needs, in other words, if the point c takes more steps to converge >= 2 or to exhaust the limit of iterations it needs more computation). And as we can see in the picture, the white dots are more abundant in the rows of the center, so this rows will require more time to finish as it is also reflected on figure 2. The following image shows the result of the sequential program with an 8x8 display:

Figure 3. *A 8x8 Mandelbrot test (taken with* ***$ ./mandeld -w 8*** *and zoomed).*

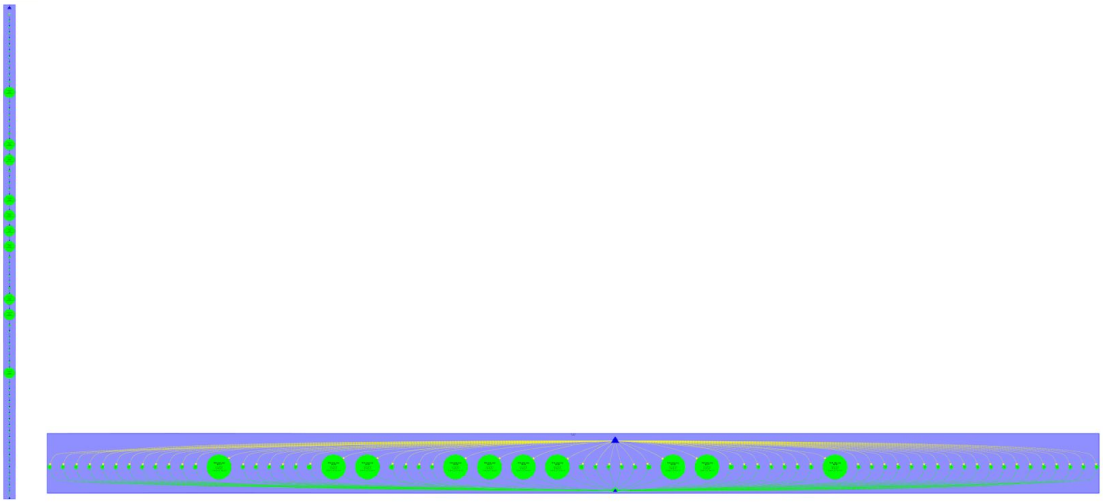

Figure 4 and 5. *Results of Tareador using the Point parallelization with an 8x8 window, the image on the left is the one corresponding to the code with display while the one on the left doesn't have those methods activated.*

The task decomposition on figure 4 executing the mandeld (d for display) is sequential for the same reason as the one in the row parallelization (display can not be executed in parallel). Using a task decomposition with point (each task is used to calculate a point of the Mandelbrot test), we can easily see in the dependence graph that there are tasks with a lot of work compared with others, that is because this tasks correspond to the white dots (the ones who exhaust all the iteration limit to know his color). As we can see in the picture 8x8, there are 10 white dots and in the graph there are 10 heavy tasks and in the same order as they appear in the picture (figure 3).

The conclusion we get from both cases is that we need to protect the display methods with a critical section and that there is clearly a load imbalance implicit on the problem due to the point having

different computation times. Also, that, obviously, the number of tasks generated by the row strategy is smaller than the number of task of the point one.

Parallelization using tasks

In this section we will analyze the performance of both parallelization strategies once those are implemented with the omp tasks. The codes below show the changes we have done to accomplish this:

1. **Row:**

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
        #pragma omp task firstprivate(row) private(col)
    for (col = 0; col < width; ++col) {
                ...
                #pragma omp critical
                {
                long color = (long) ((k-1) * scale_color) + min_color;
                      ...
                }
```

2. **Point:**

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
                #pragma omp task firstprivate(row, col)
                ...
                #pragma omp critical
                {
                long color = (long) ((k-1) * scale_color) + min_color;
                      ...
                }
```

We can see how the dependencies we have seen in the Tareador decomposition are respected (basically for the critical region). It is also important to make first private one or both variables to make the program continue from the correct row/column. We have done tests with those programs and, as expected, the graphical versions slow considerably the execution due to the limitation of parallelism (force by the critical directive). The image obtained was correct on both cases and can be seen on figure 6.

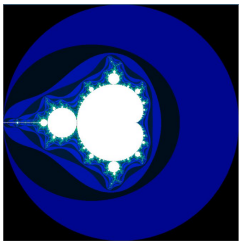

Figure 6. *Image obtained by executing the mandel-omp-point.c and mandel-omp-row.c codes.*

To finish the section we will briefly analyze the strong scalability of the two strategies. In figures 7 and 8 we can see the plots obtained:

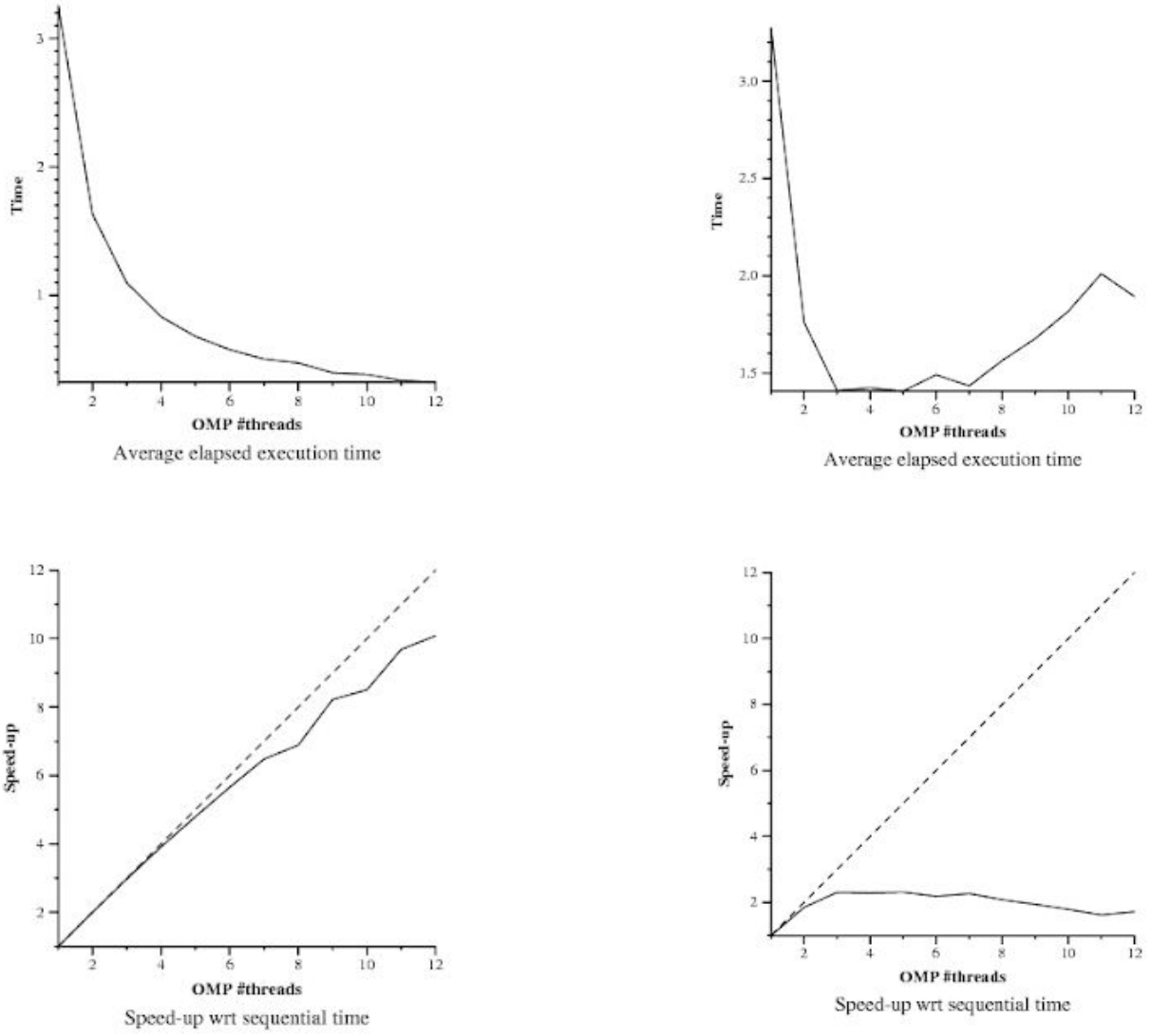

Figure 7 and 8. *Results of the submit-omp-strong.sh. On the left the plots showing the scalability of the row strategy and on the right the same for the point one.*

In both speedup plots, we can see that the as we use more threads the overhead grows, as expected. On the row one, it is not a big problem because the execution time keeps decrementing more or less linearly but on the point case, the overheads grow much faster rendering the parallelism practically useless. That happens because the point version generates a lot of tasks and that means more creation and synchronization time is needed whereas in the row version the number of tasks is lower and those overheads have a “minimal” effect. We can conclude that the scalability of the point strategy using tasks is awful and that the row one scales much better in comparison. A more general conclusion detached from our specific problem could be that having less parallel tasks is not always detrimental to parallelism, because if there is a great number of tasks

but the computation done by those doesn't justify their overheads (creation, synchronization and scheduling), the execution time could be bigger in parallel than in a sequential execution.

Parallelization with taskloops

In this section, as in the last one, we will take a look at the performance results of the same programs as before, but this time the parallelization will be done using taskloops. The changes in the code for both versions can be seen down below:

1. **Row:**

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
  #pragma omp taskloop grainsize(width/50) firstprivate(row)
  private(col)
              for (col = 0; col < width; ++col) {
              ...
              #pragma omp critical
              {
              long color = (long) ((k-1) * scale_color) + min_color;
                    ...
              }
```

2. **Point:**

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
    #pragma omp taskloop grainsize(1) firstprivate(row) private(col)
    for (col = 0; col < width; ++col) {
              ...
              #pragma omp critical
              {
              long color = (long) ((k-1) * scale_color) + min_color;
                    ...
              }
```

Doing the tests of both programs we have observed worse performance with the taskloop directive than with normal task decomposition. That is probably the result of overheads generated by the taskloop structure, due to its inherent complexity. As for the grain size values, we have chosen the ones that result in lower execution times, it is important to remark, though, than on the point strategy, we are forced to use *grainsize(1)* or *num_tasks(width*height)* because we want to force a task per pixel. On the following figures we can see the effects of those changes when it comes to scalability (strong scalability, to be more precise):

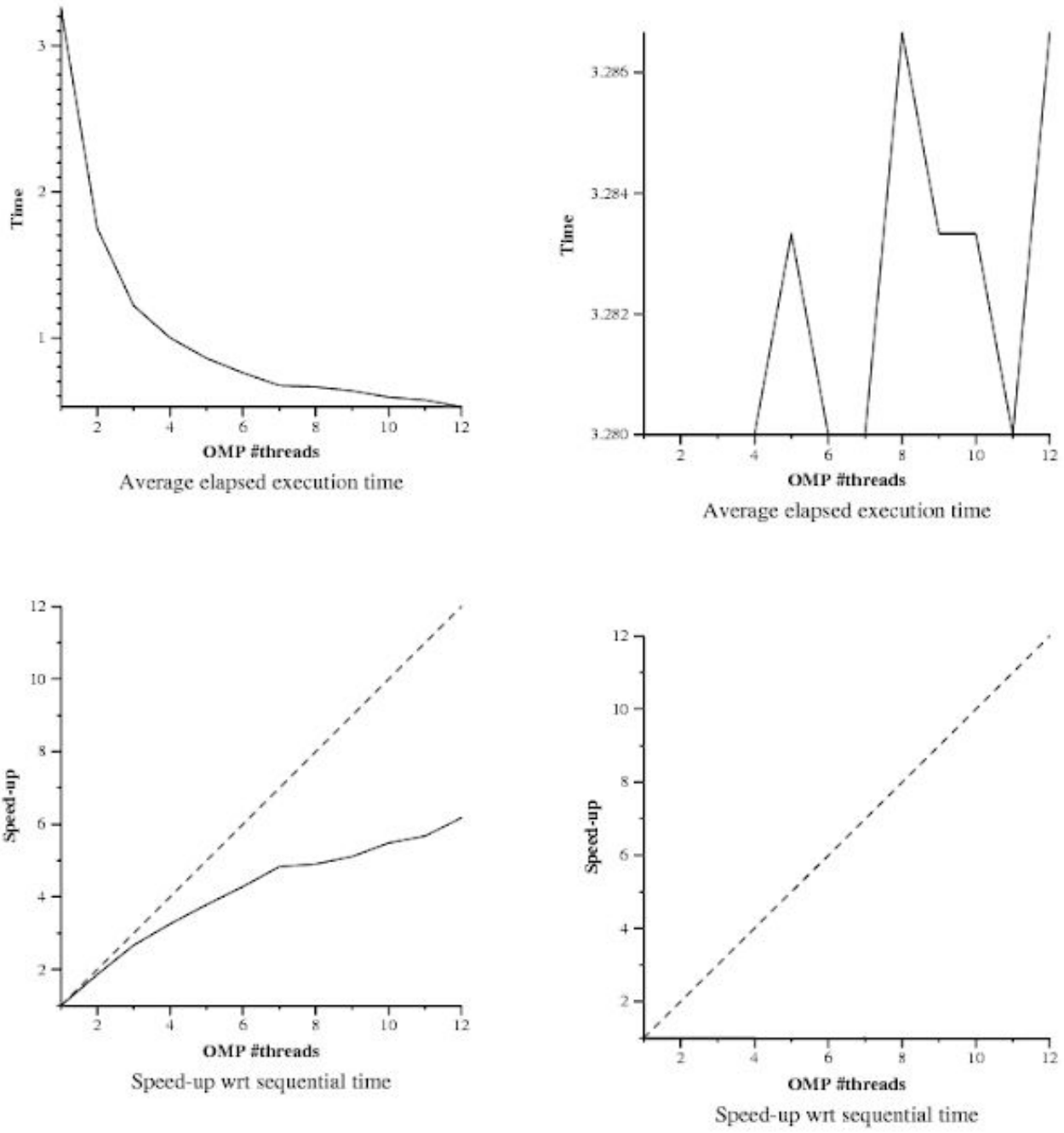

Figure 9 and 10. *Results of the submit-omp-strong.sh. On the left the speedup plots for the row strategy and on the right, the same information for the point version.*

From figure 9 we can deduce that the scalability of the row strategy deviates substantially from the theoretical perfect line. As stated above, that can be the result of the implicit overheads produced by the taskloop construct and other factors as task initialization or synchronization forced by the critical region. As for the figure 10, showing the scalability behaviour related to the point strategy, we can observe that the speedup remains constant no matter how many threads are used. We could say then that the point version scales awfully if implemented with the taskloop directive. As a general conclusion about taskloops, we can state that, at least in this case (but probably it is a more general affirmation), task parallelization makes a better use of the parallel resources than taskloops.

Parallelization with for

In this section we will analyze the performance of both parallelization strategies once those are implemented with the omp for and the different schedules for it. The changes in the code to achieve this are depicted below:

1. **Row:**

```
#pragma omp parallel for schedule(runtime) private(row, col)
for (row = 0; row < height; ++row) {
              for (col = 0; col < width; ++col) {
              ...
              #pragma omp critical
              {
              long color = (long) ((k-1) * scale_color) + min_color;
                    ...
              }
```

2. **Point:**

```
#pragma omp parallel private(row)
for (row = 0; row < height; ++row) {
    #pragma omp for schedule(runtime) private(col) nowait
    for (col = 0; col < width; ++col) {
              ...
              #pragma omp critical
              {
              long color = (long) ((k-1) * scale_color) + min_color;
                    ...
              }
```

In both codes we introduce the schedule(runtime) clause because *runtime* is a variable that will be modified before executing the program (the value it takes comes from the environment variable *OMP_SCHEDULE*), in our case we will assign four different values (“static”, “static, 10”, “dynamic, 10”, “guided, 10”). Note that the number indicates the grain size for the schedule, in other words, the number of iterations given to each thread. In the second code, to implement the point parallelization correctly, we have to use the *nowait* clause, if we didn’t the threads would wait to terminate each iteration of a row, and that is not the intention of the proposed strategy.

We will start off by analyzing the results given by Paraver using the row strategy and making a brief reasoning on the effects of the different schedules:

Static(default):

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	260,085,308 ns	-	1,379,023,705 ns	17,623 ns	2,568 ns
THREAD 1.1.2	1,922,889 ns	245,389,489 ns	1,391,803,546 ns	13,280 ns	-
THREAD 1.1.3	289,637,473 ns	245,389,504 ns	1,104,094,664 ns	7,563 ns	-
THREAD 1.1.4	1,371,146,539 ns	245,389,456 ns	22,587,341 ns	5,868 ns	-
THREAD 1.1.5	1,352,504,992 ns	245,389,519 ns	41,228,441 ns	6,252 ns	-
THREAD 1.1.6	276,379,555 ns	245,370,803 ns	1,117,373,268 ns	5,578 ns	-
THREAD 1.1.7	1,480,983 ns	245,368,978 ns	1,392,273,708 ns	5,535 ns	-
THREAD 1.1.8	840,521 ns	245,289,311 ns	1,392,992,912 ns	6,460 ns	-
Total	3,553,998,260 ns	1,717,587,060 ns	7,841,377,585 ns	68,159 ns	2,568 ns
Average	444,249,782.50 ns	245,369,580 ns	980,172,198.12 ns	8,519.88 ns	2,568 ns
Maximum	1,371,146,539 ns	245,389,519 ns	1,392,992,912 ns	17,623 ns	2,568 ns
Minimum	840,521 ns	245,289,311 ns	22,587,341 ns	5,535 ns	2,568 ns
StDev	542,952,129.53 ns	33,871.45 ns	559,188,757.95 ns	4,189.63 ns	0 ns
Avg/Max	0.32	1.00	0.70	0.48	1

Figure 11. *Times corresponding to each thread (execution and overheads) with the static scheduling obtained with paraver.*

When the schedule is static and we do not indicate a specific grain size, a default value will be assigned automatically, respecting the constraint that each thread must execute a similar number of iterations (the default function is num_iteration/num_threads, so in this case 800 iterations divided by 8 threads resulting in 100 iterations per thread). Because of this, there will be a significant load unbalance between threads (threads that compute the middle rows of the Mandelbrot set, thread 4 and 5, will have more work to do). That provokes a big synchronization overhead on those threads that are left waiting while others do most of the work.

Static, 10:

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	704,094,130 ns	-	12,815,267 ns	21,408 ns	2,360 ns
THREAD 1.1.2	439,894,106 ns	232,679,230 ns	44,349,486 ns	10,343 ns	-
THREAD 1.1.3	396,264,399 ns	232,586,410 ns	88,073,843 ns	8,513 ns	-
THREAD 1.1.4	463,215,696 ns	232,685,348 ns	21,026,063 ns	6,058 ns	-
THREAD 1.1.5	459,190,722 ns	232,685,515 ns	25,051,083 ns	5,845 ns	-
THREAD 1.1.6	428,744,598 ns	232,584,447 ns	55,598,062 ns	6,058 ns	-
THREAD 1.1.7	436,486,785 ns	232,595,830 ns	47,844,522 ns	6,028 ns	-
THREAD 1.1.8	435,154,408 ns	232,667,570 ns	49,104,780 ns	6,407 ns	-
Total	3,763,044,844 ns	1,628,484,350 ns	343,863,106 ns	70,660 ns	2,360 ns
Average	470,380,605.50 ns	232,640,621.43 ns	42,982,888.25 ns	8,832.50 ns	2,360 ns
Maximum	704,094,130 ns	232,685,515 ns	88,073,843 ns	21,408 ns	2,360 ns
Minimum	396,264,399 ns	232,584,447 ns	12,815,267 ns	5,845 ns	2,360 ns
StDev	90,377,287.56 ns	45,250.83 ns	22,274,016.89 ns	4,982.42 ns	0 ns
Avg/Max	0.67	1.00	0.49	0.41	1

Figure 12. *Times corresponding to each thread (execution and overheads) with the static 10 scheduling obtained with paraver.*

However if the schedule is static but the grain size is assigned to 10, we can see that the aforementioned load unbalance does not happen. The reason for this is that consecutive iterations

done by a thread are smaller making impossible for a single thread to compute most of the central points of the set (and thus causing imbalances). With this chunk size, then we achieve a more balanced work distribution where all threads do a similar amount of work.

Dynamic, 10:

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	706,013,708 ns	-	9,285,454 ns	13,305 ns	2,202 ns
THREAD 1.1.2	434,351,301 ns	249,028,459 ns	31,918,224 ns	16,685 ns	-
THREAD 1.1.3	434,277,124 ns	249,106,946 ns	31,921,581 ns	9,018 ns	-
THREAD 1.1.4	437,184,899 ns	249,043,252 ns	29,078,128 ns	8,390 ns	-
THREAD 1.1.5	435,687,960 ns	249,079,630 ns	30,540,383 ns	6,696 ns	-
THREAD 1.1.6	434,391,789 ns	249,039,317 ns	31,873,780 ns	9,783 ns	-
THREAD 1.1.7	443,786,170 ns	249,043,314 ns	22,478,195 ns	6,990 ns	-
THREAD 1.1.8	436,358,994 ns	249,105,404 ns	29,844,178 ns	6,093 ns	-
Total	3,762,051,945 ns	1,743,446,322 ns	216,939,923 ns	76,960 ns	2,202 ns
Average	470,256,493.12 ns	249,063,760.29 ns	27,117,490.38 ns	9,620 ns	2,202 ns
Maximum	706,013,708 ns	249,106,946 ns	31,921,581 ns	16,685 ns	2,202 ns
Minimum	434,277,124 ns	249,028,459 ns	9,285,454 ns	6,093 ns	2,202 ns
StDev	89,155,672.47 ns	30,550.54 ns	7,342,073.30 ns	3,415.07 ns	0 ns
Avg/Max	0.67	1.00	0.85	0.58	1

Figure 13. *Times corresponding to each thread (execution and overheads) with the dynamic 10 scheduling obtained with paraver.*

With a dynamic scheduling and a grain size of 10, the tasks are not assigned to a thread directly, the tasks are delivered to them as the finish a task, so here will be the less unbalance work. The objective of that schedule is that all threads are always running and not depend from his id.

Guided, 10:

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	807,584,838 ns	-	363,440,473 ns	13,333 ns	2,125 ns
THREAD 1.1.2	19,434,885 ns	250,071,311 ns	901,517,275 ns	17,298 ns	-
THREAD 1.1.3	889,789,627 ns	246,283,780 ns	34,958,137 ns	9,225 ns	-
THREAD 1.1.4	903,932,624 ns	246,288,220 ns	20,813,405 ns	6,520 ns	-
THREAD 1.1.5	557,828,977 ns	246,283,705 ns	366,921,636 ns	6,451 ns	-
THREAD 1.1.6	75,340,385 ns	246,283,765 ns	849,409,931 ns	6,688 ns	-
THREAD 1.1.7	306,493,748 ns	246,280,354 ns	618,260,705 ns	5,962 ns	-
THREAD 1.1.8	64,910,481 ns	246,280,684 ns	859,843,668 ns	5,936 ns	-
Total	3,625,315,565 ns	1,727,771,819 ns	4,015,165,230 ns	71,413 ns	2,125 ns
Average	453,164,445.62 ns	246,824,545.57 ns	501,895,653.75 ns	8,926.62 ns	2,125 ns
Maximum	903,932,624 ns	250,071,311 ns	901,517,275 ns	17,298 ns	2,125 ns
Minimum	19,434,885 ns	246,280,354 ns	20,813,405 ns	5,936 ns	2,125 ns
StDev	359,431,454.77 ns	1,325,488.61 ns	337,081,187.60 ns	3,940.97 ns	0 ns
Avg/Max	0.50	0.99	0.56	0.52	1

Figure 14. *Times corresponding to each thread (execution and overheads) with the guided 10 scheduling obtained with paraver.*

When the runtime variable is assigned to *guided* with a grain size of 10, the schedule starts with a large chunk size and it will periodically decrease it to better handle load imbalance between threads (if a thread has already processed big chunks, the next ones will have a reduced size). The specified

number parameter is the minimum chunk size to use. We can see in the paraver analysis that this technique is not the most appropriate in this problem, and we doubt of its effectivity in exercises where the amount of work done by different iterations is not constant.

Row parallelization -i 10000:

	Static	**Static, 10**	**Dynamic, 10**	**Guided, 10**
Running average time per thread	0.444s	0.470s	0.470s	0.453s
Execution unbalance (average time divided by maximum time)	0.324	0.668	0.666	0.501
SchedForkJoin (average time per thread or time if only one does)	0.980s	0.042s	0.027s	0.501s

In this table we compare the different time and ratios of the executions, in the running average time per thread the best one is the static 10 and dynamic 10, that happens because they take advantage of the cache and have faster memory accesses. In the second row, the threads of the middle do more job in static and guided schedules (as was explained above), and that explains those values. In the last row, when the schedule fork & join is measured, obviously, the threads of the schedules with more unbalanced work will have to wait more for the slower threads to finish, making this overhead significantly bigger.

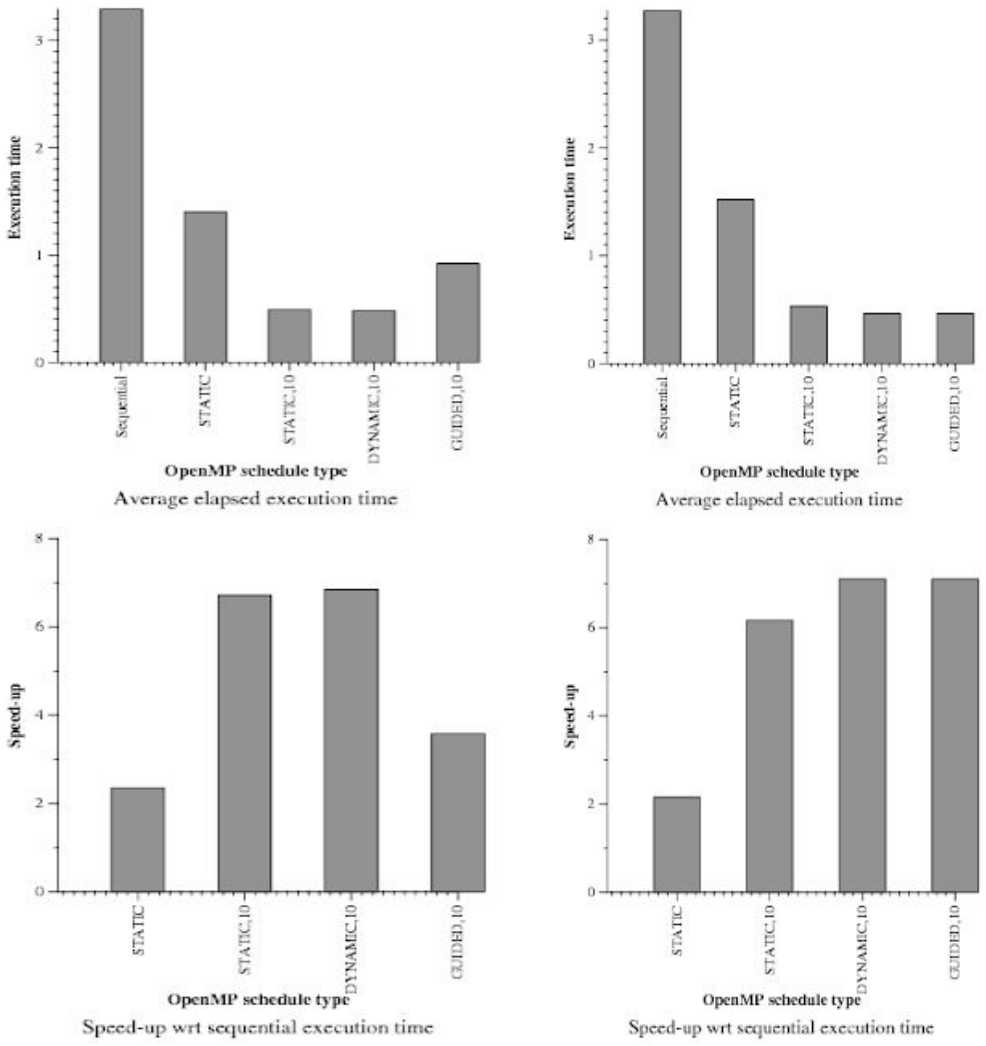

Figure 15. *Image obtained by executing the submit-schedule-omp.sh on the codes shown above. The result is the execution time and speed up plots for the different schedules with the row strategy (left) and the point one (right).*

In the point implementation, we can deduce the same conclusions except for the schedule guided 10, we can see in the chart that it's a good option, so it has a good speed-up, that is because when the point parallel version is applied, the schedule guided have enough time to reduce the chunk size to a reasonable value.

Optional 1: brief analysis of the collapse directive

The relevant code used for this section can be found below:

```
#pragma omp parallel
#pragma omp for collapse(2) schedule(runtime) private(row, col)
for (row = 0; row < height; ++row) {
              for (col = 0; col < width; ++col) {
              ...
              #pragma omp critical
              {
              long color = (long) ((k-1) * scale_color) + min_color;
                    ...
              }
```

We want to analyze the impact the collapse construct has on the different schedules. In order to accomplish this objective, first we need to explain how this directive works:

- Collapse(n): according to the omp specification, collapse is a clause that allows you to parallelize multiple nested loops without introducing parallelism. The *n* stands for the number of those loops we want to parallelize.

For starters we have to point that the results obtained in terms of order of the iterations have a significant variation depending on the chosen schedule. One example of the this difference could be seen executing our code with *schedule(static)* (the order is similar to the one in point parallelization) or with *schedule(dynamic)* (the iterations are executed similarly to the row strategy).

Let's analyze now the different speedups we obtain depending on the schedule:

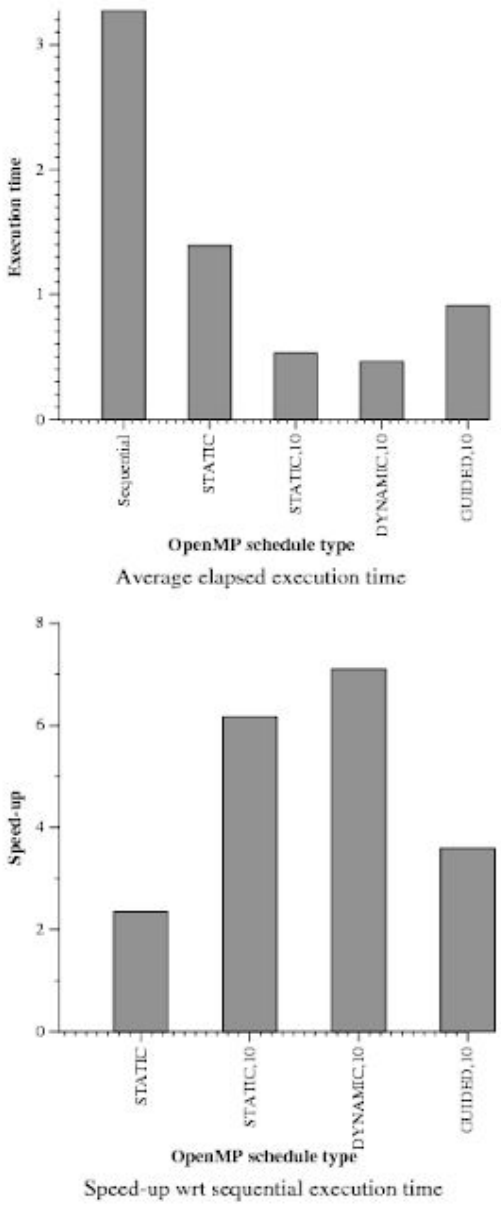

Figure 16. *Image obtained by executing the submit-schedule-omp.sh on the code shown above. The result is the execution time and speed up plots for the different schedules.*

From this figure we can extract that both *static* and *guided* scheduling performance is worse than the *static,10* or *dynamic, 10*. The main reason leading to these differences is the load imbalance between tasks due to the chunk size chosen. In the two worst schedules, the number of iterations per block is bigger than on the others; in the static case because the quotient between num_iterations and num_threads results in a value around a 100 and in the guided one, because even if the minimum size of the chunk is 10, the starting value can be greater. This big chunk sizes can generate (and they do if the information we have gathered is right) an important work imbalance between task on our problem because the middle points of a Mandelbrot set need a lot of computation to converge, so if a thread is assigned a big block of those pixels, it will inevitably work more than the others.

As for the effect of the collapse directive, we expected results similar to those in the point strategy, because collapse should create groups of iterations that compute points (depending on the chunk size, it should have few differences with the point version). Analyzing the figures obtained, we have concluded that our initial assumption was not right, and we have deduced that is probably because there is not a no wait clause per group of iterations (the loop structure does not allow space for this directive), so when a group of points finishes, it needs to wait for the rest of the row/rows, yielding different results from those expected.

Optional 2: parallel creation of tasks

In this last experimental section we will analyze how would the parallel creation of tasks affect the performance of our program. The following code shows the modifications that have been done in order to achieve this objective:

```
#pragma omp parallel
#pragma omp for schedule(runtime) private(row, col)
for (row = 0; row < height; ++row) {
            #pragma omp task private(col) firstprivate(row)
            for (col = 0; col < width; ++col) {
            ...
            #pragma omp critical
            {
            long color = (long) ((k-1) * scale_color) + min_color;
                  ...
            }
```

In this code the tasks are generated in a way reminiscent of the tree strategy used on recursive procedures. Both processes are similar because the task creation is parallelized making the process use better the multiprocessor resources.

As for the results, as expected, in terms of order of execution, the iterations are computed as in the row strategy (mostly because the tasks are created in a similar order). The speed up plots, though show some interesting facts that, upon reflection, make a lot of sense but were not expected at the time of the experimentation:

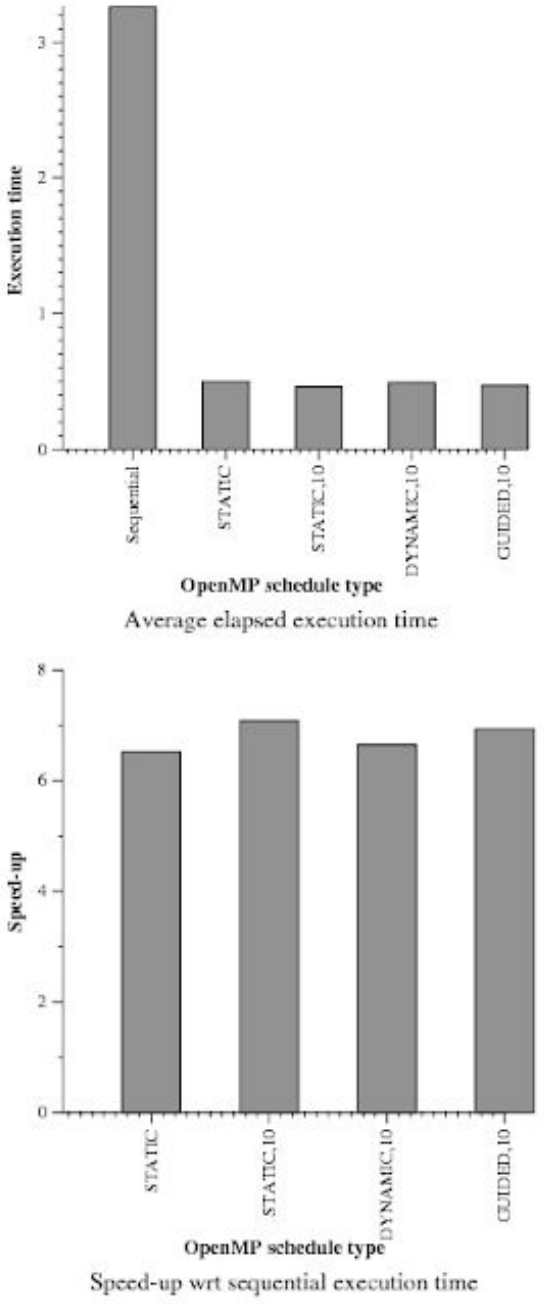

Figure 17. *Image obtained by executing the submit-schedule-omp.sh on the code shown above. The result is the execution time and speed up plots for the different schedules.*

On the figure above we detect a similar speedup no matter which scheduling we choose and the explanation is rather easy: the scheduling only has an effect on the creation of the tasks, but not on their computation, because it is applied on the for directive not inside of each task. The cost of creating tasks in our problem is really similar on the different iterations, and so, with this code we have eliminated the main source of differences between scheduling politics, the load imbalance between tasks.

Conclusions

In this Laboratory Assignment, we've learnt about different techniques of parallelization of the Mandelbrot set and we have analyzed them with paraver, tareador and extrae. We have also tested the different directives offered by OpenMP to parallelize mainly loops, but useful for other structures as well (task, taskloop, for).

On one hand, we have studied the performance of parallelization using task and taskloop. As you can see in our results, we conclude that doing it with tasks is much better than a taskloop (it gave us more speed-up), however, doing it by tasks is more risky and it is easy to do a mistake, because needs more control of the variables and of the problem.

In the other hand, we analyze the performance of for parallelization and his types of schedule to apply it, we concluded that if you want the best balance of threads, (this is important to maximize the speed-up), you should use dynamic schedule, however it takes more time to assign threads to the tasks, so if the problem size is known and all the iterations have the same amount of work, the best option to parallelize with for is with static schedule so this will imply the less overhead.

Finally, we conclude that the best option for parallelizing Mandelbrot set is the one we tested in Optional 2 section, in which the parallel tasks are created parallely, making a posterior dynamic assignment of the tasks that doesn't create a load unbalance or synchronization problems.

*We want to point out that the number of synonyms for the word "parallel" is ridiculously low and we apologize for its repeated use all along the document.

Bibliography

The links below have been consulted to shape this document and provide more information about the topics discussed:

- Wikipedia (2018). *Embarrassingly parallel*. Estados Unidos: Wikipedia. [Date of reference: 22/04/2018]. Available on: <https://en.wikipedia.org/wiki/Embarrassingly_parallel>
- Barry Wilkinson (2016). *Embarrassingly parallel computations*. Estados Unidos: Prentice Hall. [Date of reference: 22/04/2018]. Available on: <http://www.cs.nthu.edu.tw/~ychung/slides/para_programming/slides3.pdf>
- E. Ayguadé, J. Corbalán, J. Morillo, J. Tubella and G. Utrera (2017). *Embarrassingly parallelism with OpenMP: Mandelbrot set*. Spain: UPC. [Date of reference: 22/04/2018]. Available on: <http://atenea.upc.edu/pluginfile.php/2247198/mod_resource/content/11/lab3-PAR.pdf>
- Wikipedia (2018). *Mandelbrot set*. Estados Unidos: Wikipedia. [Date of reference: 22/04/2018]. Available on: <https://en.wikipedia.org/wiki/Mandelbrot_set>
- Brady Haran (2014). *The Mandelbrot Set - Numberphile*. Estados Unidos: Youtube. [Date of reference: 22/04/2018]. Available on: <https://youtu.be/NGMRB4O922I>