PAR Laboratory Assignment

Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

Marc Domínguez de la Rocha
Joan Sánchez García
par2103

Fall 2017-2018
18/11/17

Index

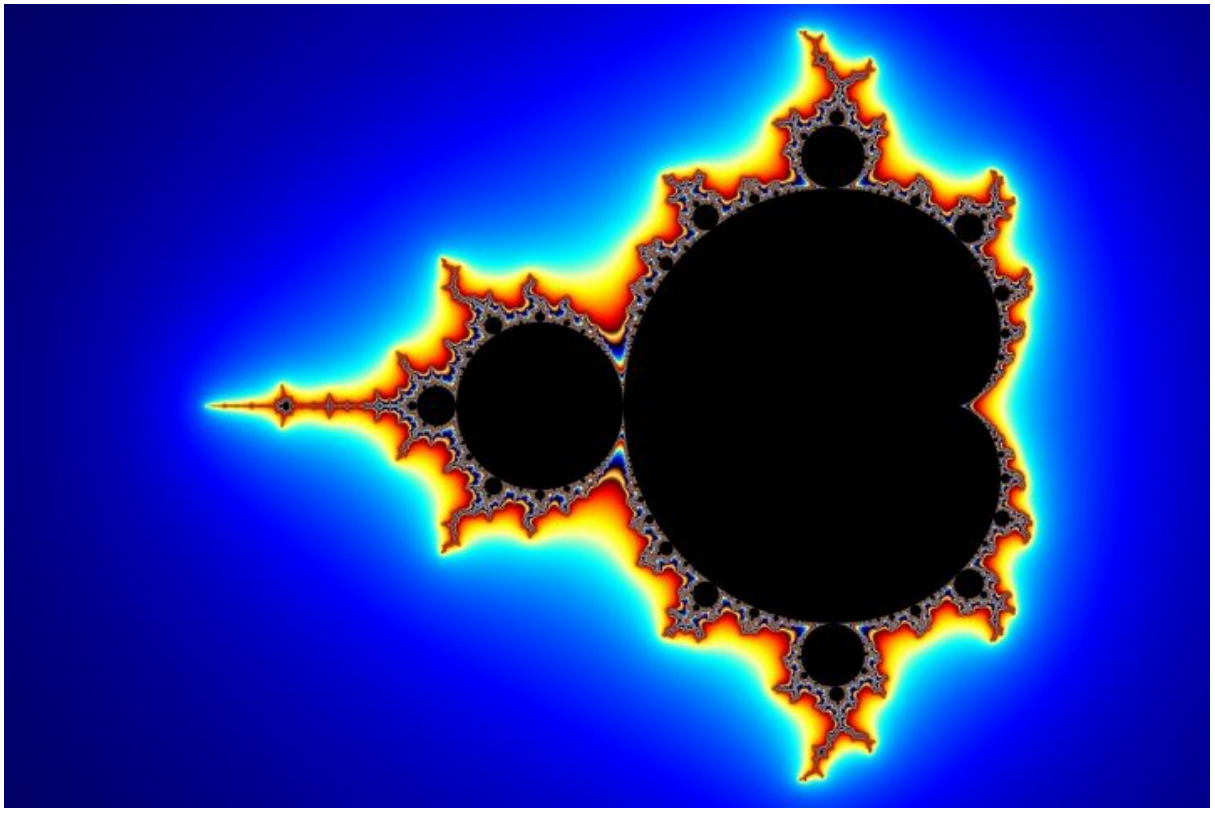
3.1 Introduction

In this lab session we are asked to study the task granularity and the parallelization of the Mandelbrot set code.

The Mandelbrot set is the set of complex numbers c for which the function $z_{n+1} = z_n^2 + c$ does not diverge when iterated from $n = 0$, that is it, the set of numbers for which the sequence z_0, z_1, etc, remains bounded in absolute value.

The function $z_{n+1} = z_n^2 + c$ is repeated, while monitoring the magnitude of z_n and counting the number of iterations. If this magnitude never exceeds a certain value B, then the point C is said to belong to the Mandelbrot Set. If the magnitude of z_n does exceed B then we say that the point C does not belong to the Mandelbrot set.

To generate the Mandelbrot set plot we have to color the outside points, the ones that do not belong to the set. This coloring is usually based on the number of iterations (the Escape Time) that the magnitude of z_n takes to escape the radius B, for the point under consideration.



Before starting with the analysis of the parallelization strategies we will study the task granularity between a row or a point decomposition.

3.2 Task granularity analysis

1. Which are the two most important common characteristics of the task graphs generated for the two task granularities (Row and Point) for the non-graphical version of mandel-tareador? Obtain the task graphs that are generated in both cases for -w 8.

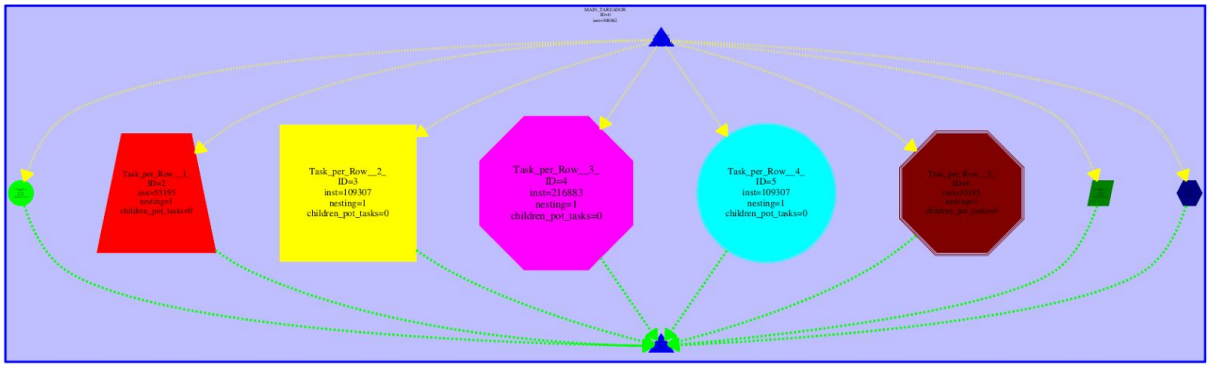


Figure 1. Row task graph (HD at annex mandelbrot-tareador-nodisplay-row)

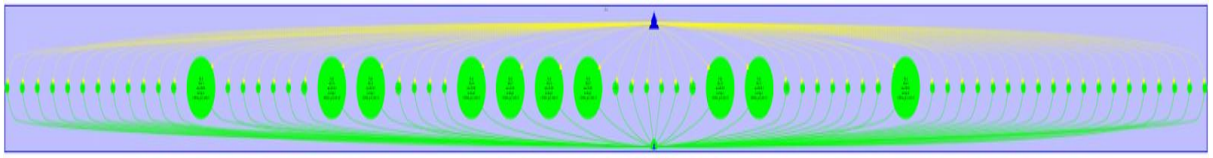


Figure 2. Point task graph (HD at annex mandelbrot-tareador-nodisplay-point)

The two most important common characteristics we can observe in both graphs are:

- In both graphs the tasks don't have the same granularity. Central tasks have more work to do than the peripheral ones.
- Tasks don’t have dependencies between them.

2. Which section of the code is causing the serialization of all tasks in mandeld-tareador? How do you plan to protect this section of code in the parallel OpenMP code?

```
/* 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);
}
```

This piece of code refers to the print process, the one responsible of drawing the outside points, the ones that do not belong to the Mandelbrot set.

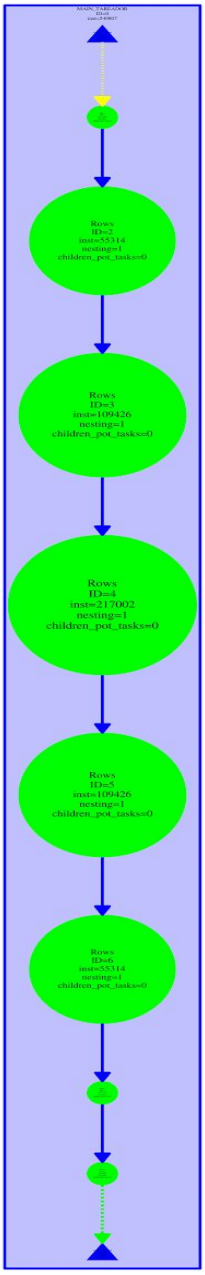


Figure 3. Row Task graph with dependences (HD annex mandelbrot-tareador-display-row)

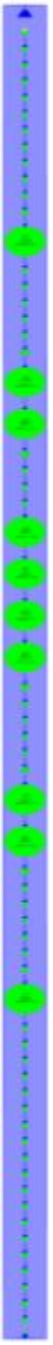


Figure 4. Point Task graph with dependences (HD annex mandelbrot-tareador-display-point)

This dependence is caused by the use of global variables inside the Xlib.h.
So if we want to protect this region of the code we should use the #pragma omp critical, clause that provides a mutual exclusion region where just one thread can operate at the same time.

3.3 OpenMP-task based parallelization

First of all, we must know what an *OpenMP-task based parallelization* is. When we are talking about tasks we mean a kind of pointer to a function (a piece of code) which will be assigned to a thread in order to execute it. So, doing a Row decomposition consists in picking an entire row for one task, and a Point decomposition consists in spliting the execution of a row between tasks, so each one will have assigned one or more columns.

3.3.1 Row decomposition

Code can be found at mandel-omp-task-row.c annex.
In this case there are as many tasks as rows, thus, if our displays is N height and M width, we will have N tasks with same load, M iterations. So the first thought coming to mind is to think that with infinite processors is possible to have one at each row, reducing the execution time to T_1 divided by $T\infty$ (remember that *N* is the number of rows of our display). Then, the speed-up will be:

$$s(\infty) = \frac{T1}{T\infty} = N$$

This is ideal, assuming that the ϕ parallel region is equal to one and that there are no overheads at fork/join, task creation, nor synchronizations due to mutual exclusion regions. The real one is quite worst than the ideal, exactly due to the overheads we supposed null in the ideal execution.

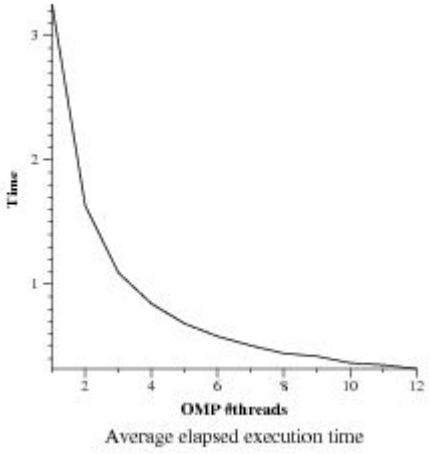


Figure 5. Execution time graph for the row decomposition task parallelization.

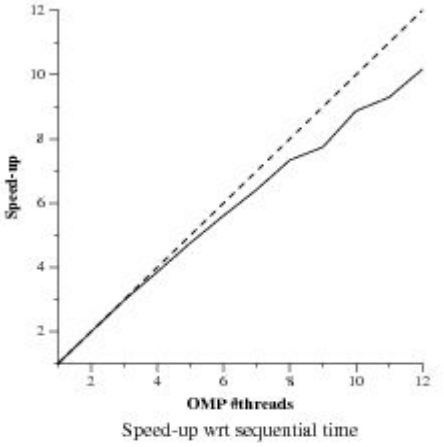


Figure 6. Speed-up graph for the row decomposition task parallelization.

The first plot shows the execution time with strong scalability. You can see how the time goes down as we increase the number of threads. And at the second one you can take a look at the speed-up respect the sequential version, which starts lineal and goes slightly down as the number of threads increase.

So, the *Row* decomposition is very good, the only remarkable bad feature is that with a big number of threads, we have more synchronization times (the print display is a mutual exclusion region), which is like executing little pieces of code sequentially, but as the number of threads increase, also increases the portion of the code executed sequentially.

3.3.2 Point decomposition

Code can be found at mandel-omp-task-point.c annex.
Once again, we will introduce first the ideal version of our task decomposition. If we would assign as many tasks as pixels of our display (*NxN pixels*), the execution time for our parallel program with infinite processors theoretically would be like *N* times faster than our *Row* decomposition version. This is because there is a processor per pixel, so if the Row decomposition was computing N rows at the same time, we will be computing NxN points at once, so the speed-up respect the row version would be N, that means that the point decomposition version should be N times faster.

This is theoretically, the one that we would like to have, but the reality it's quite different. Remember the fork/join, task creation and mutual exclusion overheads?
For every task we use we have to add their creation cost and the fact they will be trying to print within the display at the same time. As in point 3.2 we concluded that we should use a critical clause to protect the dependences, that means that just one task will execute it at a time, so they will get a lot of time blocked, waiting on a thread to release the mutual exclusion region and have its turn to print.

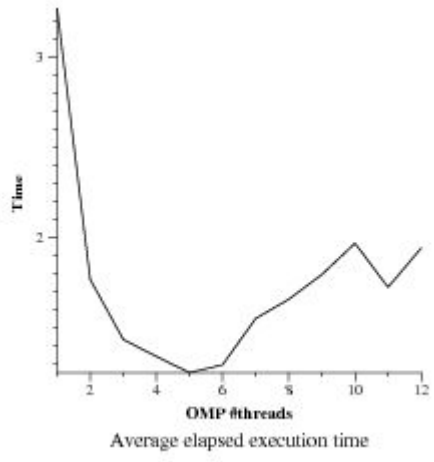


Figure 7. Execution time graph for the point decomposition task parallelization.

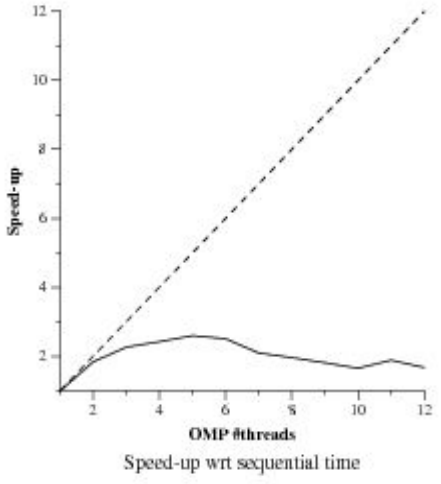


Figure 8. Speed-up graph for the point decomposition task parallelization.

Take a look to the first plot, and compare it with the *Row* decomposition version. In this case, doing a *Point* decomposition is just making a downgrade.

Take into account that as Mandelbrot set holds every calculated number that is contained in a certain radius, so there is not workload balance at all, there will be numbers that last a lot to diverge or to determine that has converged (would be like saying that it reaches its equilibrium).

So, all the overheads and the not absolutely balanced workload mentioned above made our execution times a complete mess, the ideal version really beats up the *row task* decomposition, but actually, is not the reality.

3.4 OpenMP-taskloop based parallelization

Now we are going to study the parallelization of the code using the taskloop directive. Taskloop specifies that the iterations of one or more associated loops will be executed in parallel using OpenMP tasks.

3.4.1 Row decomposition

Code can be found at mandel-omp-taskloop-row.c annex.

The core of the row decomposition is to split the number of iterations of the first loop in x tasks of a given grain size (number of iterations per task).
This can be archieved in this way:

```
/* Calculate points and save/display */
#pragma omp parallel
{
#pragma omp single
#pragma omp taskloop grainsize(16) private(col)
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
        complex z, c;

        z.real = z.imag = 0;
```

The code above gives 16 iterations of the first loop to each one of the threads that are executing it. This value can be changed. We have studied for 8, 16 and 32, but the last one got such a bad speed up that we are not going to consider it.

Numerically

#threads	Elapsed average
1	3.25000000000000000000
2	1.64000000000000000000
3	1.13666666666666666666
4	.86000000000000000000
5	.71333333333333333333
6	.58333333333333333333
7	.54333333333333333333
8	.48000000000000000000
9	.44666666666666666666
10	.41000000000000000000
11	.39666666666666666666
12	.35000000000000000000

#threads	Elapsed average
1	3.25000000000000000000
2	1.64000000000000000000
3	1.16666666666666666666
4	.91666666666666666666
5	.72666666666666666666
6	.66333333333333333333
7	.57666666666666666666
8	.50000000000000000000
9	.47333333333333333333
10	.42000000000000000000
11	.38333333333333333333
12	.37333333333333333333

#threads	Speedup
1	1.00102564102564102564
2	1.98373983739837398373
3	2.86217008797653958945
4	3.78294573643410852712
5	4.56074766355140186917
6	5.57714285714285714288
7	5.98773006134969325156
8	6.77777777777777777777
9	7.28358208955223880607
10	7.93495934959349593495
11	8.20168067226890756315
12	9.29523809523809523808

Figure 9. Elapsed average time and Speedup for 8 iterations as a grainsize for the row decomposition task-loop parallelization.

#threads	Speedup
1	1.00000000000000000000
2	1.98170731707317073170
3	2.78571428571428571430
4	3.54545454545454545457
5	4.47247706422018348627
6	4.89949748743718592967
7	5.63583815028901734110
8	6.50000000000000000000
9	6.86619718309859154934
10	7.73809523809523809523
11	8.47826086956521739137
12	8.70535714285714285722

Figure 10. Elapsed average time and Speedup for 16 iterations as a grainsize for the row decomposition task-loop parallelization.

And graphically

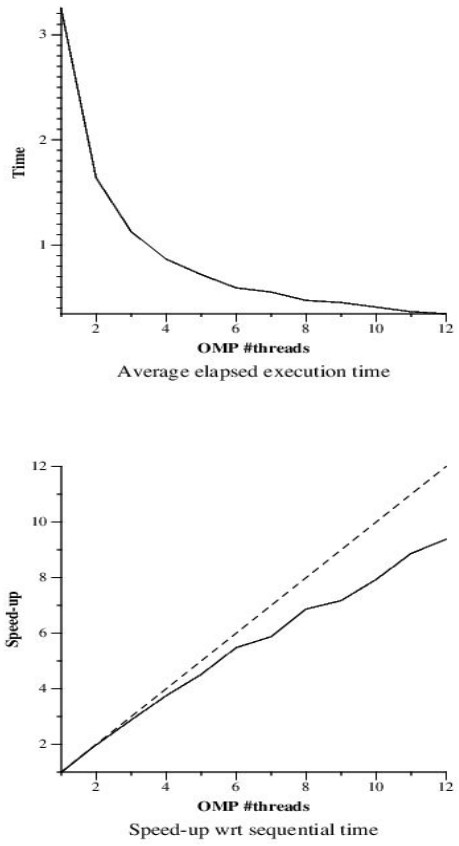


Figure 11. Execution time and Speed Up graphs for 8 iterations as a grainsize for the row decomposition task-loop parallelization.

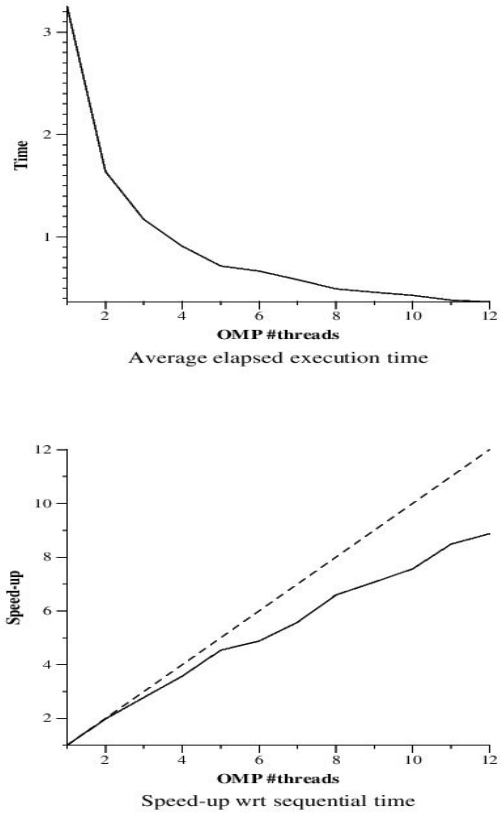


Figure 12. Execution time and Speed Up graphs for 16 iterations as a grainsize for the row decomposition task-loop parallelization.

Both graphically and numerically we can see that the 8 iterations grainsize behaves in a better way than the 16 one. 8 is not a perfect grainsize, but splits the 1000 size of the vector into 125 integer blocks of few iterations. 16 needs 62,5 bigger blocks.

It looks like the overhead of creating the tasks are cheaper than executing 8 extra iterations.

As less iterations per task, as more parallelizable our code will be, but we have to take into account the task generation overheads, so the ideal grainsize will be the one that can balance this two aspects.

3.4.2 Point decomposition

Code can be found at mandel-omp-taskloop-point.c annex.

Now our objective is to see if the pixel decomposition behaves better than the row.
Now the parallelization is done inside one row, so the objective is to generate the rows sequentially but fullfill it in parallel.

```
/* Calculate points and save/display */
#pragma omp parallel
{
#pragma omp single
for (row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) grainsize(8)
    for (col = 0; col < width; ++col) {
        complex z, c;
```

The code above is a grainsize example with 8 points per task, so we can have x number of threads fulfilling, in this case, 8 pixels each one at the same time.
As in the row decomposition, we are going to study graphically and numerically the speedups and execution times. We have studied 10 pixels, 16 and 32, but again, 32 behaves much worse than the other two, so we are not going to consider it.

Numerically

#threads	Elapsed average
1	3.26000000000000000000
2	1.72333333333333333333
3	1.19666666666666666666
4	.96333333333333333333
5	.81666666666666666666
6	.73000000000000000000
7	.65666666666666666666
8	.61333333333333333333
9	.58666666666666666666
10	.55000000000000000000

#threads	Elapsed average
1	3.26000000000000000000
2	1.74666666666666666666
3	1.22000000000000000000
4	.99000000000000000000
5	.86000000000000000000
6	.76000000000000000000
7	.68333333333333333333
8	.65666666666666666666
9	.64000000000000000000
10	.59333333333333333333

11	.52666666666666666666
12	.49333333333333333333

#threads	Speedup
1	.99693251533742331288
2	1.88588007736943907157
3	2.71587743732590529249
4	3.37370242214532871973
5	3.97959183673469387758
6	4.45205479452054794520
7	4.94923857868020304573
8	5.29891304347826086959
9	5.53977272727272727279
10	5.90909090909090909090
11	6.17088607594936708868
12	6.58783783783783783788

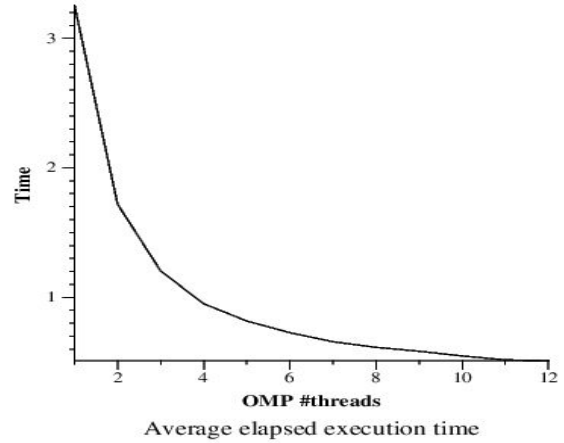
Figure 13. Elapsed average time and Speedup for 10 iterations as a grainsize for the point decomposition task-loop parallelization.

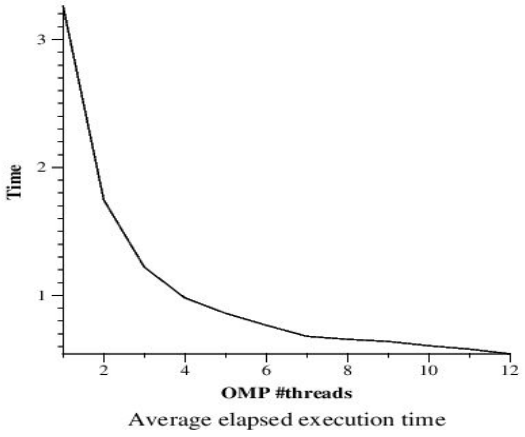
11	.55666666666666666666
12	.54333333333333333333

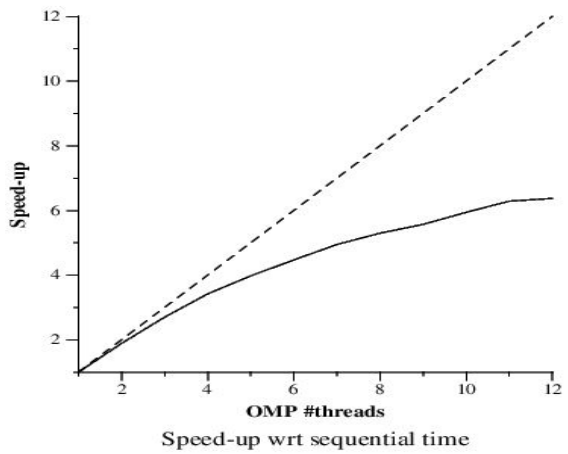
#threads	Speedup
1	1.00000000000000000000
2	1.86641221374045801527
3	2.67213114754098360655
4	3.29292929292929292929
5	3.79069767441860465116
6	4.28947368421052631578
7	4.77073170731707317075
8	4.96446700507614213203
9	5.09375000000000000000
10	5.49438202247191011239
11	5.85628742514970059887
12	6.00000000000000000003

Figure 14. Elapsed average time and Speedup for 16 iterations as a grainsize for the point decomposition task-loop parallelization.

And graphically







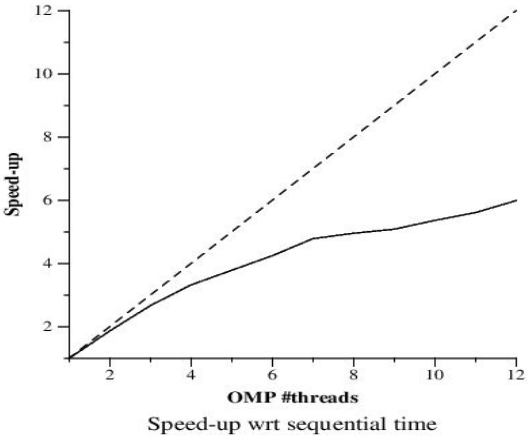


Figure 15. Execution time and Speed Up graphs for 10 points as a grainsize for the point decomposition task-loop parallelization.

Figure 16. Execution time and Speed Up graphs for 16 points as a grainsize for the point decomposition task-loop parallelization.

Again in this case, as smaller we choose the grainsize, as bigger the speedup will be.

So as the Strong Scalability consists in increasing the number of processors maintaining the problem size with the objective to reduce the execution time, the row decomposition shows a better performance than the point. This is like this because as the row decomposition is generating x lines at the same time, the point is just generating one faster, and as the overheads for generating the tasks are not negligible, the point decomposition is slower (it generates much more tasks).

3.5 OpenMP for-base parallelization

All codes can be found at the annex directory.
We will show the plots first and then reason about why, so you can refer to them when you desire.

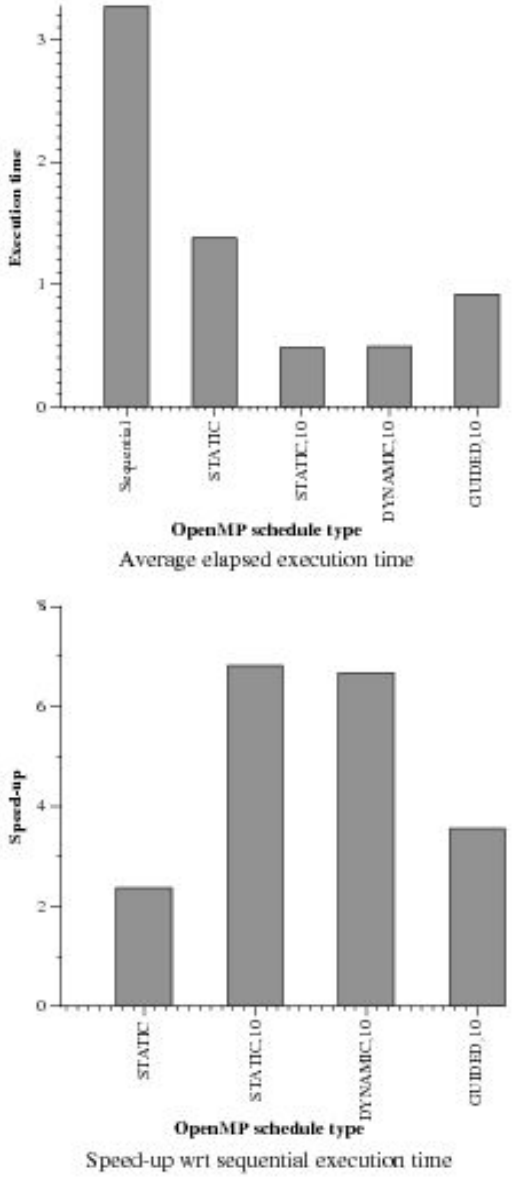


Figure 17. Execution time and Speed Up graphs for the different scheduling techniques row distribution for parallelization.

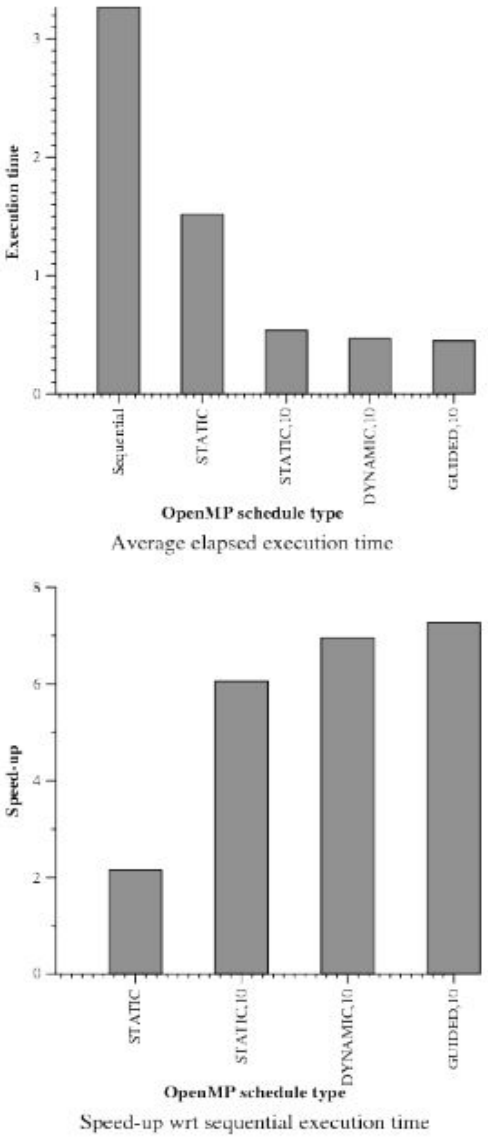


Figure 18. Execution time and Speed Up graphs for the different scheduling techniques point distribution for parallelization.

The schedule in a *for* directive stands for how the iterations of that loop are distributed among the threads. The *static* schedule kind will assign always a constant number of iterations for each thread, *dynamic* is just the opposite, goes up and down distributing iterations dynamically and *guided* starts with a number of iterations per threads and decreases as they are executed.

Thus, as we mentioned in the *3.3 OpenMP task-based parallelization* section the *Row* decomposition is quite good and the *Point* one in the ideal case is even better.
Do you remember why actually the *Point* decomposition was bad? Was because of unbalance, but that's the point about *Dynamic* and *Guided* scheduling.
Appreciate that the *Row* decomposition is good, as always has been, but take a look at the *P*oint decomposition, it has improved drastically in the *dynamic* and *guided* versions.
As we have explained, that solves the unbalance problem reducing a lot the non-doing time of each thread, as the plot shows. These are the remarkable improvements to point out of the *Point* decomposition. But let's analyze each schedule kind one by one:

- *Static*: This one grabs chunks of constant size, so it will create unbalance between threads, we don't get special speed-up for this schedule.

- *Static,10*: The constant size mentioned above is now ten, we can see that results to be better than the first one, grabbing a bigger chunk size avoids big synchronization and assignment time overheads.

- *Dynamic,10*: As soon a thread finish executing the iterations it has been assigned, the scheduler gives it more work to do, reducing considerably the unbalance problems we insisted in.

- *Guided,10*: Quite similar to the Dynamic, but reduces even more the unbalance problems, due to its way to assign the iterations (the fastest thread is not the one who does more work, as the execution of the number of iterations is reduced to a limit).

Using *Paraver* and *Extrae* we extracted these values:

	static	static, 10	dynamic, 10	guided, 10
Running average time per thread	443,577,884	548,493,184	495,371,490	478,860,266
Execution unbalance	0.3226	0.3930	0.7208	0.3662
SchedForkJoin	1,380,705,25	10,958,309	41,364,312.13	790,051,602

Times expressed in nanoseconds (ns)

We got these times, it really makes sense to see the SchedForkJoin increase as we introduce schedulers which do more assignments than *static*.

3.6 Conclusions

After having studied different parallelization strategies, we can determine that the ones which tries to balance the load between the threads behave in a much better way. Then, the possibilities that the *for* clause provides us, being able to determine the scheduling for assigning tasks to threads, are superior than the *task* and the *task-loop* strategy.

Among the three different scheduling possibilities, *dynamic* and *guided* are the most flexible, being *guided* the top one.

The main goal of this lab session was to understand that not only the grain size influences the execution time of a program, but also the task distribution.

If we could execute our code without the overheads of fork/join, task creation, or synchronizations due to mutual exclusion regions the task-based parallelization, without even grain size, would be the best. But as we are in a real environment, we have to take all this extra time into account, so the for strategy with a dynamic or guided schedule will be the best option.