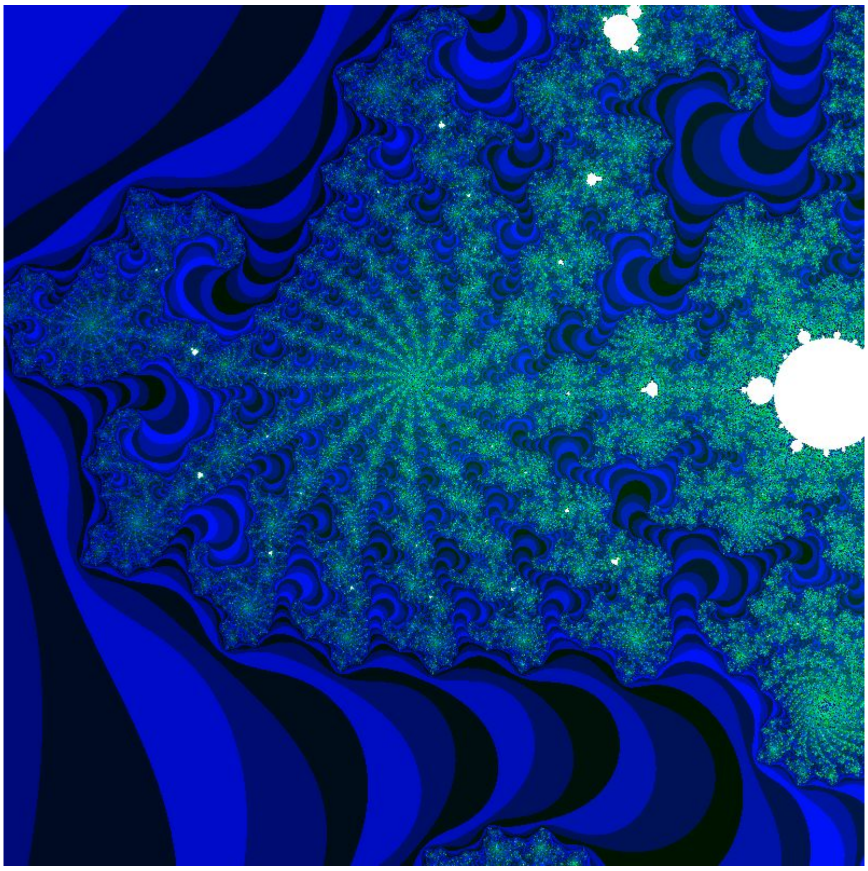
LAB 1

PAR-FIB



Members:

Víctor Pérez Martos

Albert Suàrez Molgó

Group: *par2205*

Date: *13/11/2015*

Academic course: *QT-2015/16*

Task granularity analysis

1.

After some iterations of the program and looking carefully at the code, we can clearly see that both granularities can be parallelized without any dependence between them. Another important characteristic is that, even though they have the same task size, each of them have different execution times due to the fact that the calculation time can vary between points. These characteristics can be seen on images 1 and 2.

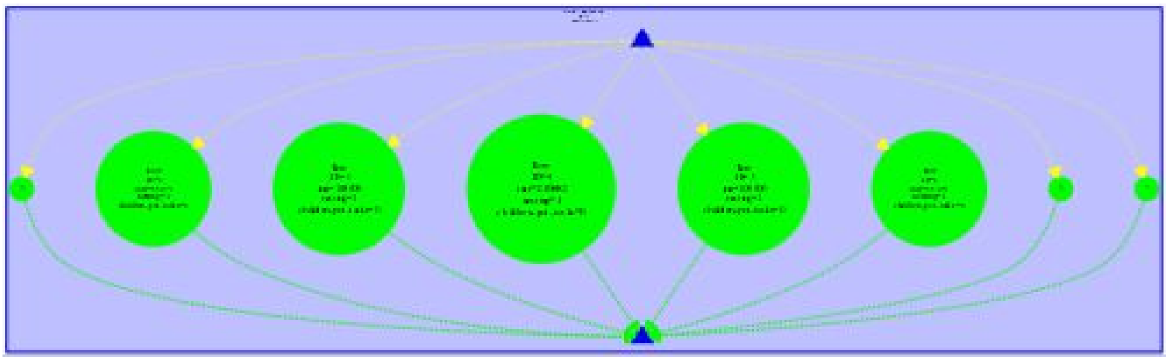


Image 1: Task graph generated by tareador of non-graphical version of mandel with Row granularity.

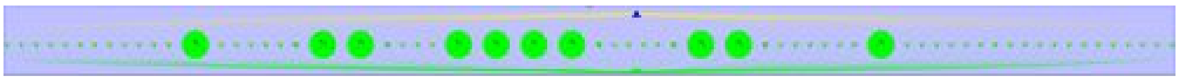


Image 2: Task graph generated by tareador of non-graphical version of mandel with Point granularity.

2.

The serialization of the code is caused by the code used to draw the image. *Gc* and *win* variables generate dependences. To protect this section of the code, we use a *#pragma omp critical* clause to avoid conflicts.

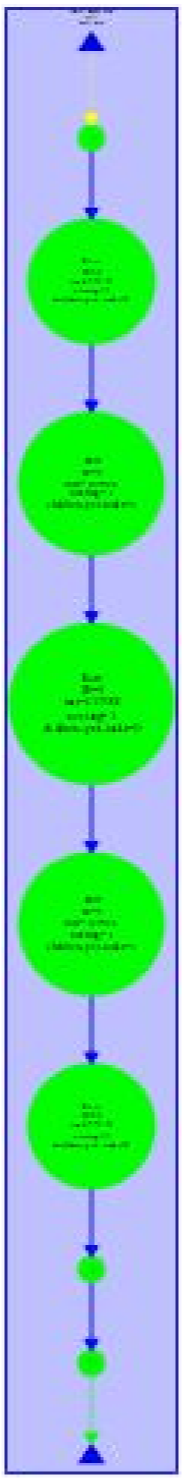


Image 3: Task graph generated by tareador of graphical version of mandel with Row granularity.

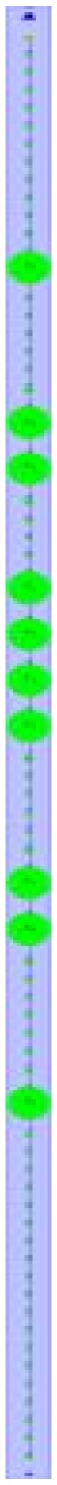


Image 4: Task graph generated by tareador of graphical version of mandel with Point granularity.

OpenMP task-based parallelization

1.

On the task based parallelization, we added a *#pragma omp parallel* clause at the beginning to indicate the region we wanted to parallelize. Then, we used the clause *#pragma omp single* to indicate that only one thread will be creating the tasks (once they are created, the tasks are done by the threads who are waiting). In addition, we use the clause *#pragma omp task* so the thread who enters by the single clause can create the different tasks to be done. We have also used the clause *#pragma omp critical* on the graphical version to protect this section that can causes problems due to dependencies.

Both codes are attached to the zip file: *mandel-omp-task-point.c* and *mandel-omp-task-row.c*

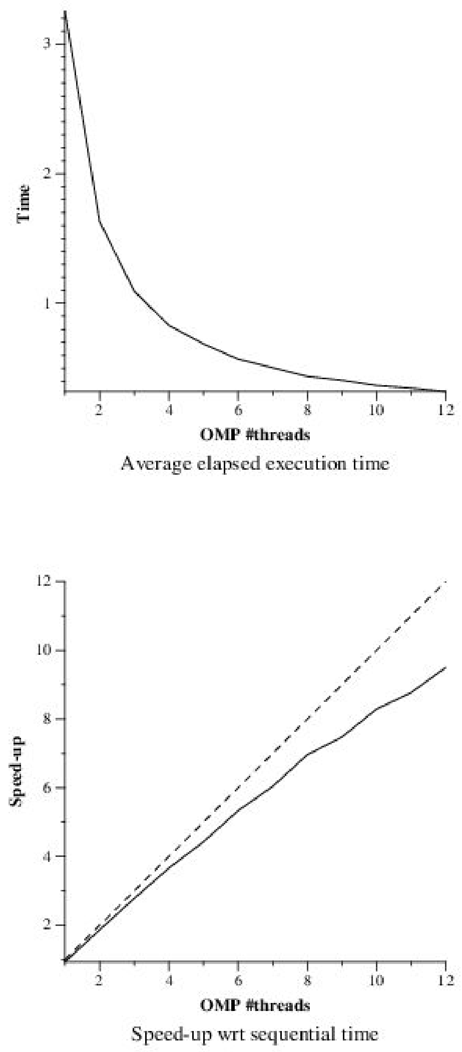
2.

# processors	Execution Time - Row	Execution Time - Point
1	3.253	0.026
2	1.630	0.266
3	1.093	0.443
4	0.830	0.556
5	0.686	0.976
6	0.570	0.866
7	0.503	1.250
8	0.436	1.503
9	0.406	1.896
10	0.366	2.156
11	0.346	2.326
12	0.320	2.750

Table 1: Execution time of both decompositions

# processors	Speedup - Row	Speedup - Point
1	0.934	114.000
2	1.865	11.4000
3	2.780	6.85714
4	3.662	5.46107
5	4.427	3.11262
6	5.333	3.50769
7	6.039	2.43200
8	6.961	2.02217
9	7.475	1.60281
10	8.290	1.40958
11	8.769	1.30659
12	9.500	1.10545

Table 2: Speedup of both decompositions



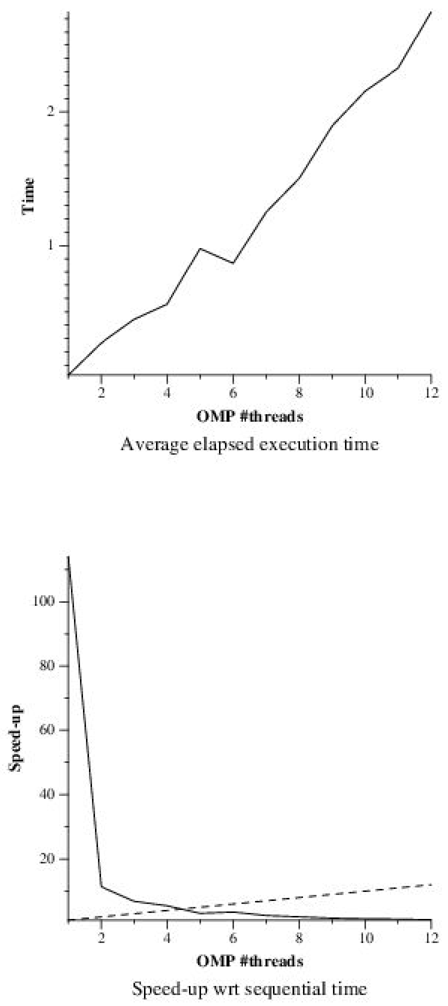


Image 5: Graphics of row decomposition.

Image 6: Graphics of point decomposition.

As we can see on tables 1 and 2, for the parallelized version by rows the improvement of time is almost linear. Nevertheless in the version by dots the time increases if we use more threads, so we have an speed-up under 1. This effect is caused by the overhead associated with the synchronization between threads, which means that the cost of using parallelization is higher than the time we gain with it. The graphics on image 6 show perfectly this effect.

OpenMP for-based parallelization

1.

On the for based parallelization, we added a *#pragma omp parallel* clause at the beginning to indicate the region we wanted to parallelize. We used the clause *#pragma omp for* which creates the tasks implicitly and shares them between threads (we could also omit the *#pragma omp parallel* by using *#pragma omp parallel for* instead of *#pragma omp for*). We have also used the clause *#pragma omp critical* on the graphical version to protect this section that can causes problems due to dependencies.

Both codes are attached to the zip file: *mandel-omp-for-point.c* and *mandel-omp-for-row.c*

2.

Decomposition	**Time (static)**	**Time (static,10)**	**Time (dynamic,10)**	**Time (guided, 10)**
Row	1.368909	0.455255	0.470919	0.922732
Point	1.485973	0.525787	0.450965	0.443371

Table 3: Execution times of different loop schedules with row and point decomposition.

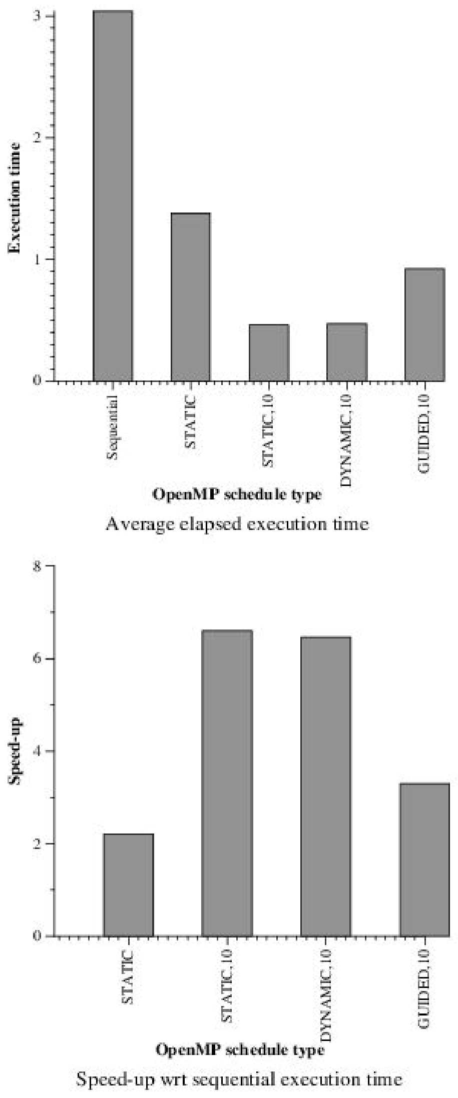


Image 7: Plots of row decomposition.

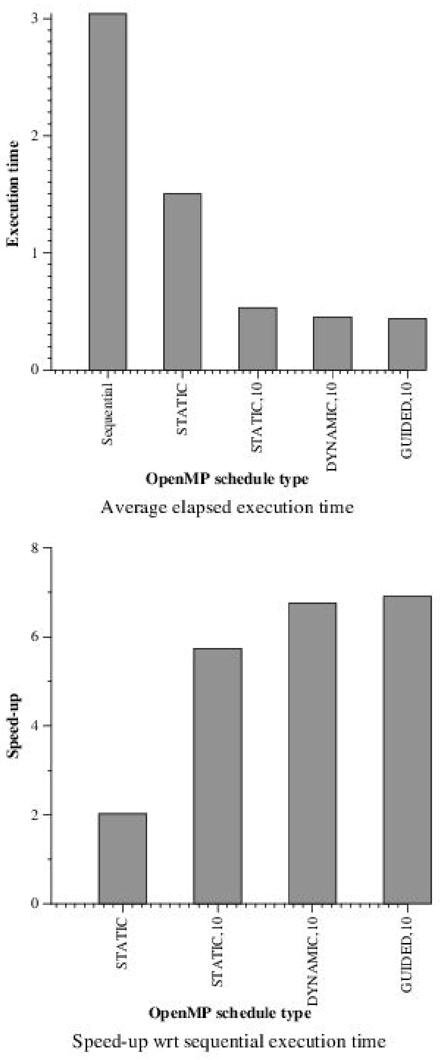


Image 8: Plots of point decomposition.

Looking at Table 3 we can appreciate that the differences between using row or point versions is almost nothing. Nevertheless, the time changes depending on the schedule policy, where dynamic policy gets the best results because of the characteristic explained before that the calculation time can vary between points. However, we can also see that *guided* and *static* with *chunk* also give great results.

3.

	static	static,10	dynamic,10	guided,10
Running average time per thread	420,505,150.25 ns	450,411,467,62 ns	461,032,670 ns	447,964,420.5 ns
execution unbalance (average time divided by maximum time)	0.28	0.85	0.95	0.95
Sched, Fork, Join (average time per thread or time if only one does)	190,032,987.90 ns	37,504,617.2 ns	19,856,626 ns	9,446,070,75 ns

Table 4: distribution time by schedule type

As we can see on table 4 the static schedule is really unbalanced while the other ones respond quite well to this aspect. This is because of the distribution used by *static* which causes to do many forks and joins, so it creates an overhead that affects negatively to our execution time. We can clearly see that adding chunk makes static to create less tasks and, therefore, spent less time on fork and join tasks.

On the other hand, we can see as dynamic schedule produces a bigger running average time due to the fact that it has to assign the tasks between threads during calculation. We can also see that guided type has the same problem because it uses a dynamic distribution form but reduces chunk during execution to obtain a better sharing of the work between threads.