PAR

THIRD DELIVERABLE

Marlen Avila

Ricard Meyerhofer

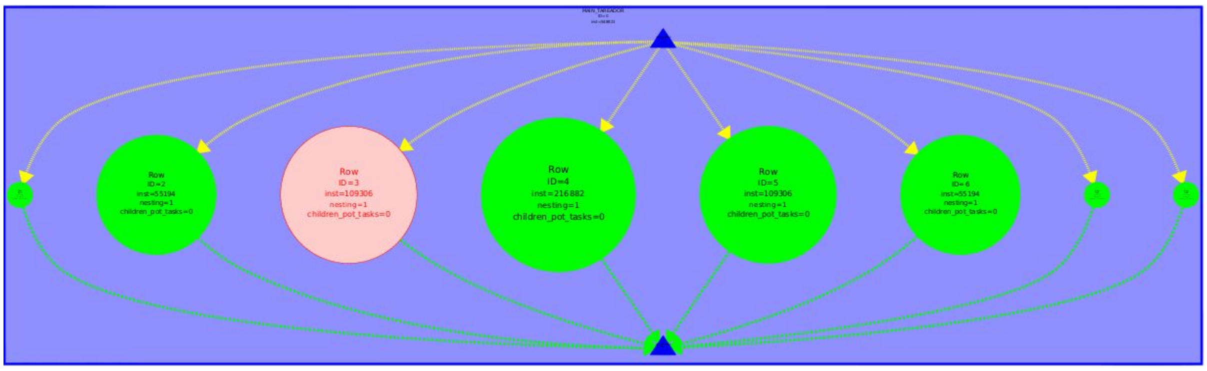
2015-2016 Q1

5.1 Task Granularity Analysis

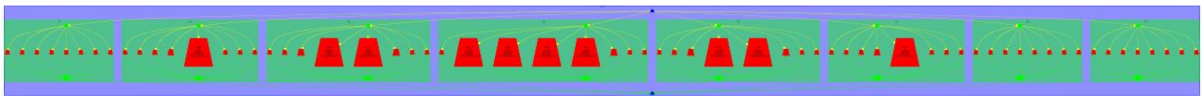
1. Which are the two most important common characteristics of the task graphs generated for the two task granularities (Row and Point) for the non-graphical version of mandel-tareador? Include the task graphs that are generated in both cases for -w 8.

R/

This is the Row task granularity graph:



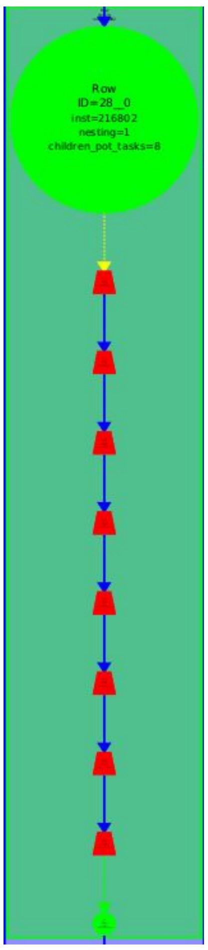
And this is the Point task granularity graph:



The two most important common characteristics are:

- In both cases we have **task independency** (all tasks can be executed at the same time).
- The **granularity is different between tasks** (in the row task graph we can see that the sizes of some tasks are smaller than others, and in the point task graph it happens the same). That means that some tasks are doing more work than others (unbalanced tasks).

-Even though we have task independency, the work done inside each task cannot be parallelizable because there is a variable dependency as we can see in the right task dependency graph.



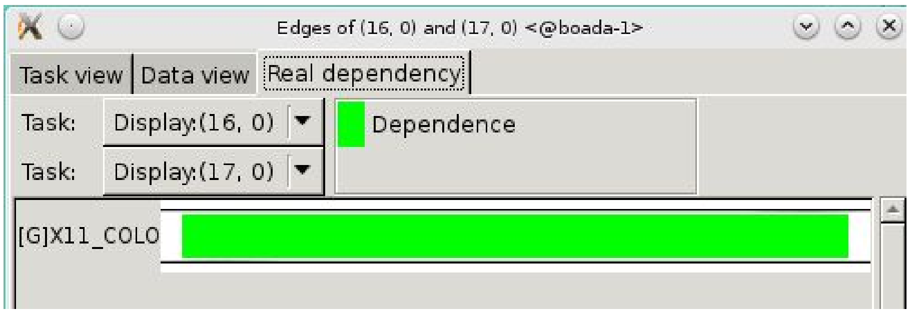
2. Which section of the code is causing the serialization of all tasks in mandeld-tareador? How have you protected this section of code in the parallel OpenMP code?

R/

The section of the code that is causing the serialization of the tasks is the following:

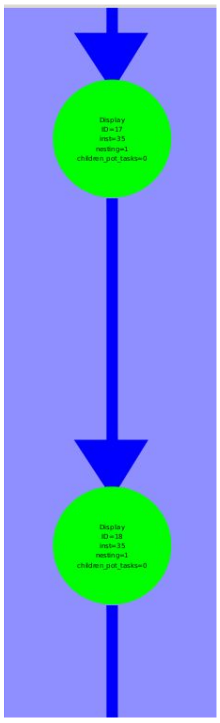
```
tareador_start_task("Display");
        long color = (long) ((k-1) * scale_color) + min_color;

        if (setup_return == EXIT_SUCCESS) {

                XSetForeground (display, gc, color);
                XDrawPoint (display, win, gc, col, row);

        }
tareador_end_task("Display");
```

Creating the new task above we've been able to find the dependence variable by clicking the blue arrow:



Task:	Display:(16, 0)	🟩 Dependence
Task:	Display:(17, 0)	

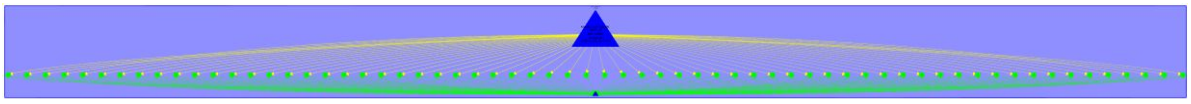
[G]X11_COLO 🟩



Once we know which is creating the serialization, we can disable this variable by adding the following clauses:

```
/* Scale color and display point  */
   tareador_start_task("Display");
   tareador_disable_object(&X11_COLOR_fake);
        long color = (long) ((k-1) * scale_color) + min_color;
        tareador_disable_object(&color);
        if (setup_return == EXIT_SUCCESS) {
                XSetForeground (display, gc, color);
                XDrawPoint (display, win, gc, col, row);
        }
   tareador_enable_object(&X11_COLOR_fake);
   tareador_end_task("Display");
```

We can see that the dependence **X11_COLOR_fake** is eliminated so the program now can be parallelizable.



To protect this section of the code in OpenMp we've added:

```
#if _DISPLAY_
        #pragma omp critical
        {
                /* 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);
                }
        }
```

We know that the code will be executing in parallel, and by adding the critical clause we can be sure that this region will be executed always by just one thread.

5.2 OpenMP task–based parallelization

1. Include the relevant portion of the codes that implement the task-based parallelization for the Row and Point decompositions (for the non–graphical and graphical options), commenting whatever necessary.
R/

mandel row

```
/* Relevant part of the code: */
{
#pragma omp parallel
#pragma omp single
   for (row = 0; row < height; ++row) {
      #pragma omp task private(col) firstprivate (row)
      {
           for (col = 0; col < width; ++col) {
               /* Code */
          }
          #if _DISPLAY_
            #pragma omp critical
            {
                /* Code */
             }
          else /* More code */
         }
      }
   }
}
```

Why we use:

- **#pragma omp parallel:** Because we want a parallel implementation. We create as much threads as tasks we have, and all of them execute the same code region.
- **#pragma omp single:** One thread creates the tasks of the traversal. The rest (and this one once the task generation is finished) cooperate to execute them.
- **#pragma omp task private(col) firstprivate (row):** We've added these clauses because the col and row values are shared by default so we're having a data race.
- In the col variable case, we've used the private clause. At first it will be undefined but then it will take the value col = 0.
  In the row variable case we've used the firstprivate clause. This way the row variable will take the value that has at the moment the task is executed, which will be the correct one.

mandel point

```
/* Relevant part of the code: */
{
#pragma omp parallel
#pragma omp single
   for (row = 0; row < height; ++row) {
        for (col = 0; col < width; ++col) {
           complex z,c;
           #pragma omp task firstprivate(col,row)
           {
            /* Code */
          }
          #if _DISPLAY_
            #pragma omp critical
            {
                /* Code */
             }
          else /* More code */
         }
      }
   }
}
```

Why we use (parallel and single explained before)

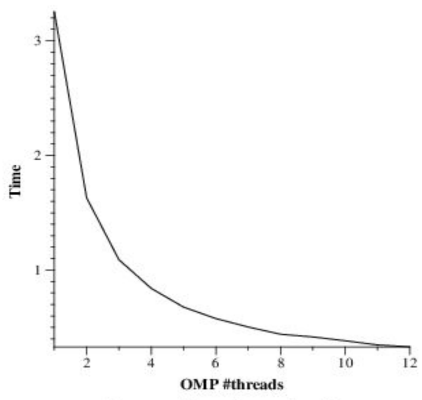
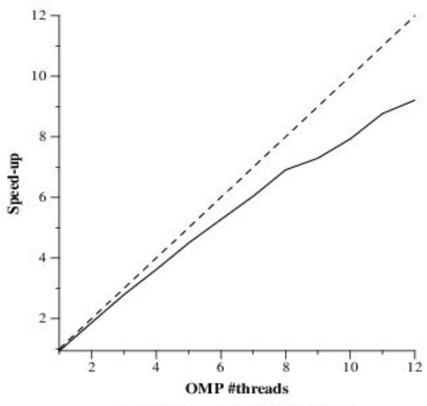
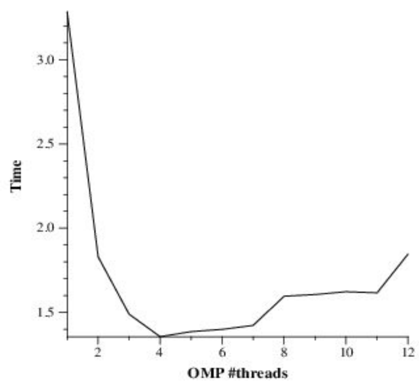
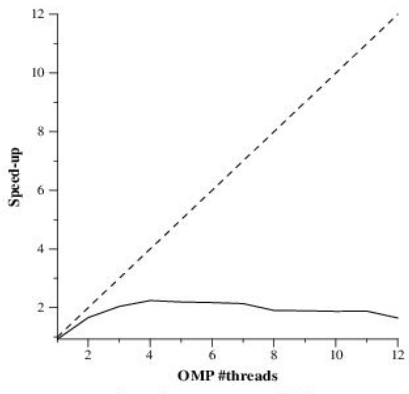
- **#pragma omp task firstprivate(col,row):** Same reason explained above, but in this case we change the col variable from a private to a firstprivate.
- **complex z,c** we've put them outside even we could put it inside.

2. For the the Row and Point decompositions of the non-graphical version, include the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000). Reason about the causes of good or bad performance in each case.

R/

First we have executed the graphical version to see that the result was correct and once known that the output was correct in both cases, we've opened the speedup.txt to see if the values were the expected and at this way, know that our programs correctness.

Once we know that our programs are doing the same that we've instrumented on tareador, let's analyze the results:

mandel row	mandel point
 Average elapsed execution time  Speed-up wrt sequential time	 Average elapsed execution time  Speed-up wrt sequential time

Strong scalability problem definition: The number of threads is changed with a fixed problem size. In this case parallelism is used to reduce the execution time of our program.

Reason about the causes of good or bad performance in each case:

- **mandel row:**
  - The performance of the mandel row is good because we're distributing the work by tasks but not with too much tasks (we create 800 tasks) so we're really obtaining a benefeit of the tasks usage and we're appliying a good strategy in a strong scalability case.
- **mandel point:**
  - The performance is really bad due to the amount of tasks created (800*800) and the overhead that this implies.

<u>5.3 OpenMP for–based parallelization</u>

1. Include the relevant portion of the codes that implement the for-based parallelization for the Row and Point decompositions (for the non–graphical and graphical options), commenting whatever necessary.
R/

mandel-for row

```
/* Relevant part of the code: */
{
#pragma omp parallel
#pragma omp for schedule(runtime) private (col)
   for (row = 0; row < height; ++row) {
           for (col = 0; col < width; ++col) {
               /* Code */
          }
          #if _DISPLAY_
            #pragma omp critical
            {
                /* Code */
             }
          else /* More code */
         }
   }
}
```

Why we use:

- **#pragma omp for schedule(runtime) private (col):** We use the private clause in col for the same reason than in mande-row. What we are now not including is the firstprivate of row because in the for construct, the induction variable(s) are automatically privatized

mandel-for point

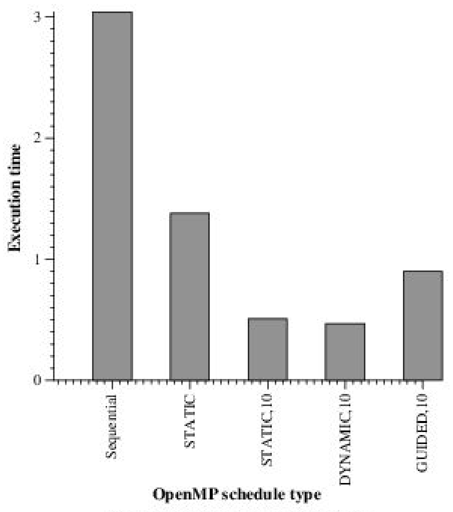
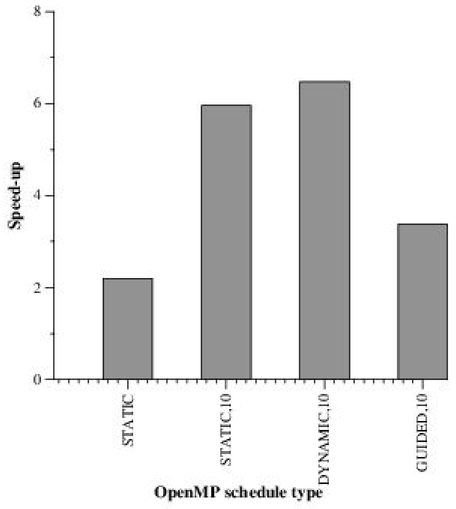
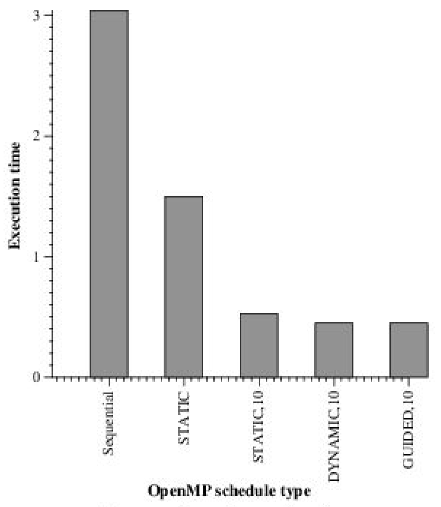
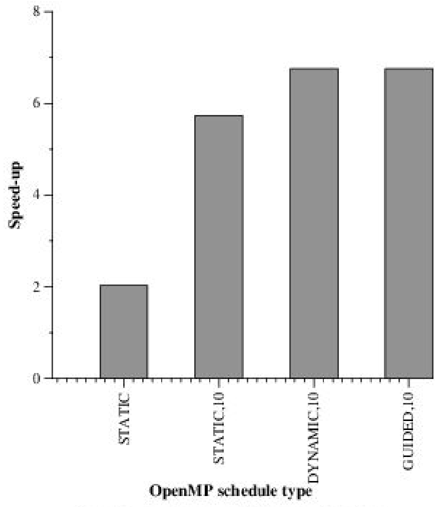
```
/* Relevant part of the code: */
{
#pragma omp parallel private(row)
   for (row = 0; row < height; ++row) {
           #pragma omp for schedule(runtime) nowait
            for (col = 0; col < width; ++col) {
                /* Code */
           }
           #if _DISPLAY_
             #pragma omp critical
             {
                 /* Code */
              }
           else /* More code */
          }
   }
}
```

Why we use:

- **#pragma omp parallel private(row):** Because row will be undefined at first but then will be declared to 0. What we are avoiding with the private clause is a data race of the variable because by default in this case, row is shared. The col variable is private due to the for clause.
- **#pragma omp for schedule(runtime) nowait:** We can make more efficient our code if we use the nowait clause because there's no sense in make the threads wait on a barrier when once the implicit barrier of the for is reached by al threads, all of them will fall in the barrier of the parallel clause.

2. For the Row and Point decompositions of the non-graphical version, include the execution time and speed–up plots that have been obtained for the 4 different loop schedules when using 8 threads (with -i 10000). Reason about the performance that is observed.

R/

mandel-for row	**mandel-for point**
 Average elapsed execution time  Speed-up wrt sequential execution time	 Average elapsed execution time  Speed-up wrt sequential execution time

STATIC: the iteration space is broken in chunks of approximately size N/num threads. Then these chunks are assigned to the threads in a Round-Robin fashion. The execution has a low speed-up due to the fact that is not balanced because of the difference of cost between iterations (there’s no significant difference between row and point).

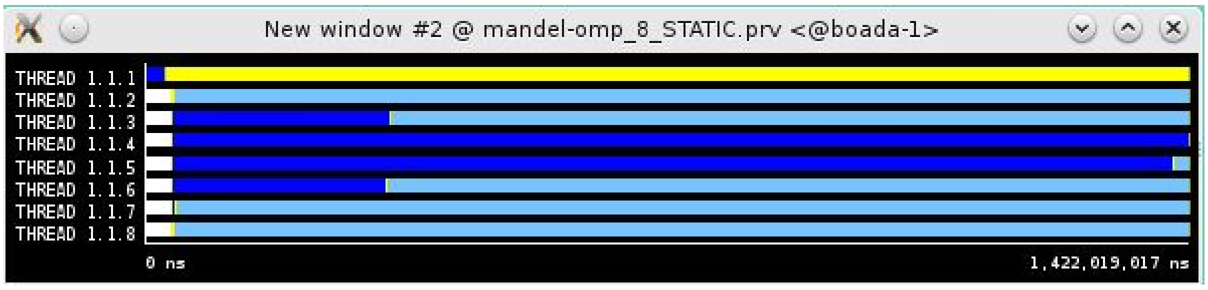
STATIC,10: Same as in STATIC but chunk is size 10. Even thought the planification is similar, using a chunk of 10 minimizes the chance for a thread to perform more work than the other ones (there’s no significant difference between row and point)

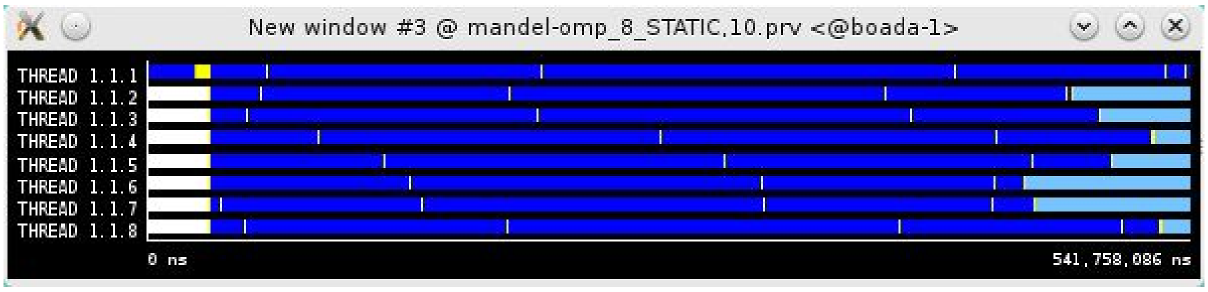
DYNAMIC,10: Threads dynamically grab chunks of N iterations until all iterations have been executed. The point implementation is slightly better because the granularity is higher so we are having less scheduling fork/join cost.

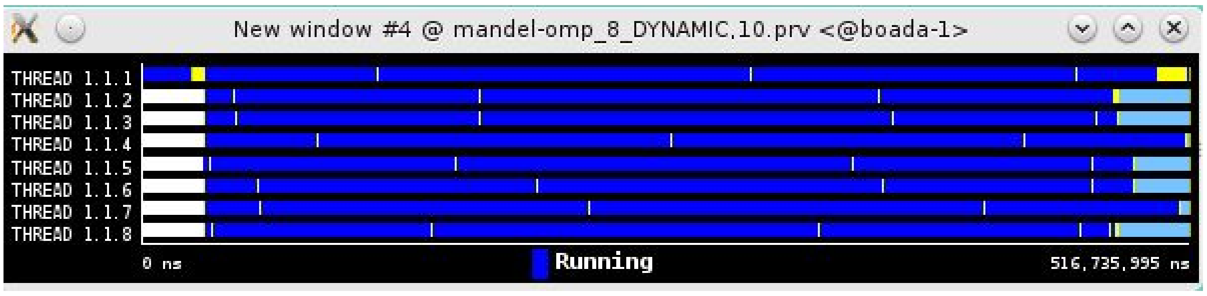
GUIDED,10: Variant of dynamic. The size of the chunks decreases as the threads grab iterations, but it is at least of size N. The guided implementation due to the scheduling of guided, firsts iterations will have a chunk similar to STATIC and this will cause the unbalance of tasks in the row implementation in the case of point we've that the result is way better than the row one, this is due to a better repartition of iterations and less scheduling fork/join cost.

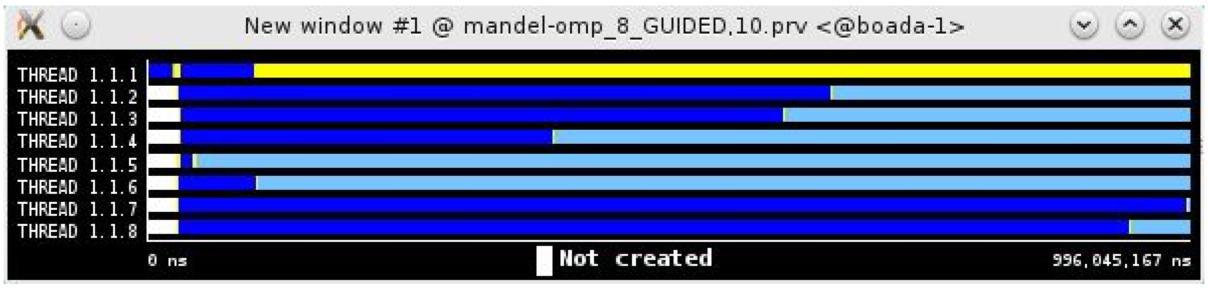
With the Row decomposition static,10 and dynamic,10 are the best options, because the execution unbalance is higher and all the threads are running most of the time, static and guided as said do have threads in idle state because of the chunk size.
And in the point decomposition dynamic and guided are the best planifications because of the task balance that we obtain using them, in this case the guided,10 does obtain a good result due to the correct balance of work.

3. For the Row parallelization strategy, complete the following table with the information extracted from the Extrae instrumented executions (with 8 threads and -i 10000) and analysis with Paraver, reasoning about the results that are obtained.
R/
First we generated the following traces to help with the explanation and to make more visual the results obtained (among with the row plot of the last exercise), then we show the results of the **OMP_profile** which gives us the result desired and finally the table and the explanation of its content.
Execution Plots without profile:

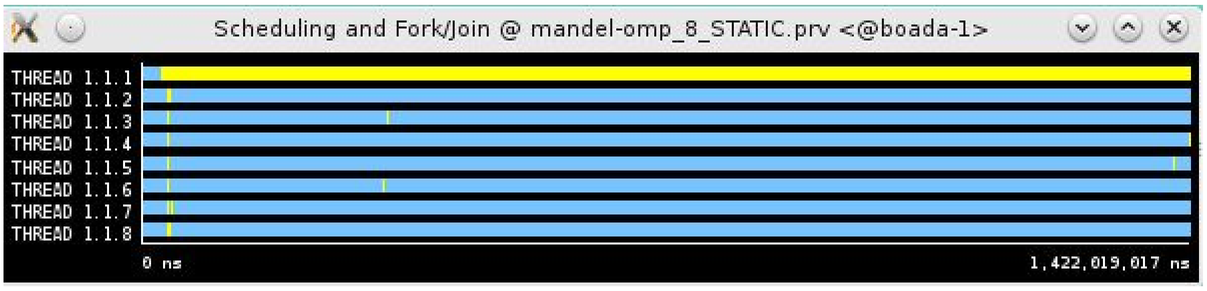


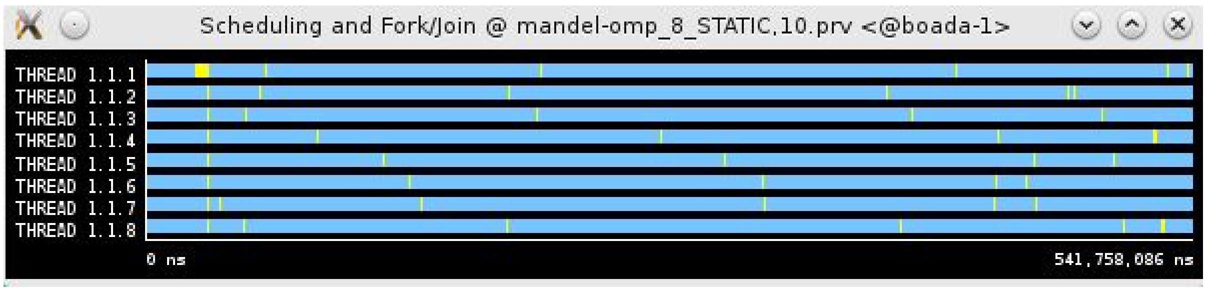


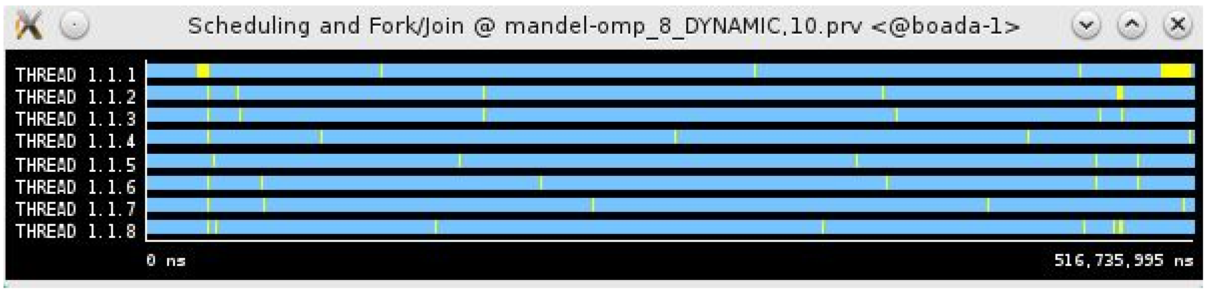


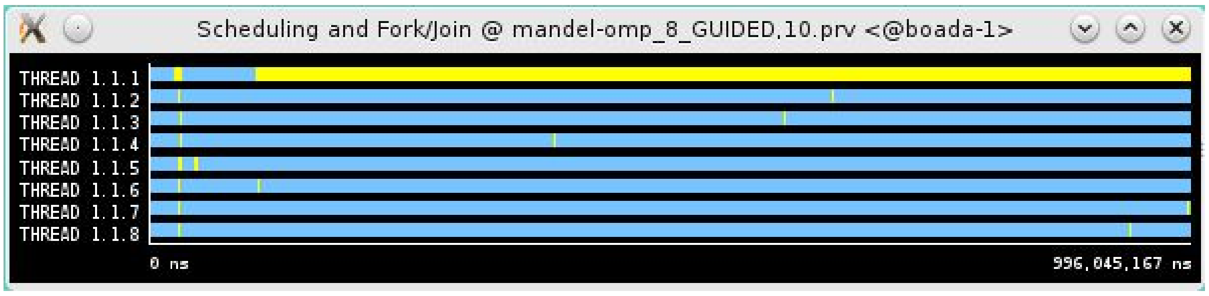


Plots with profile Scheduling & fork/join filter:

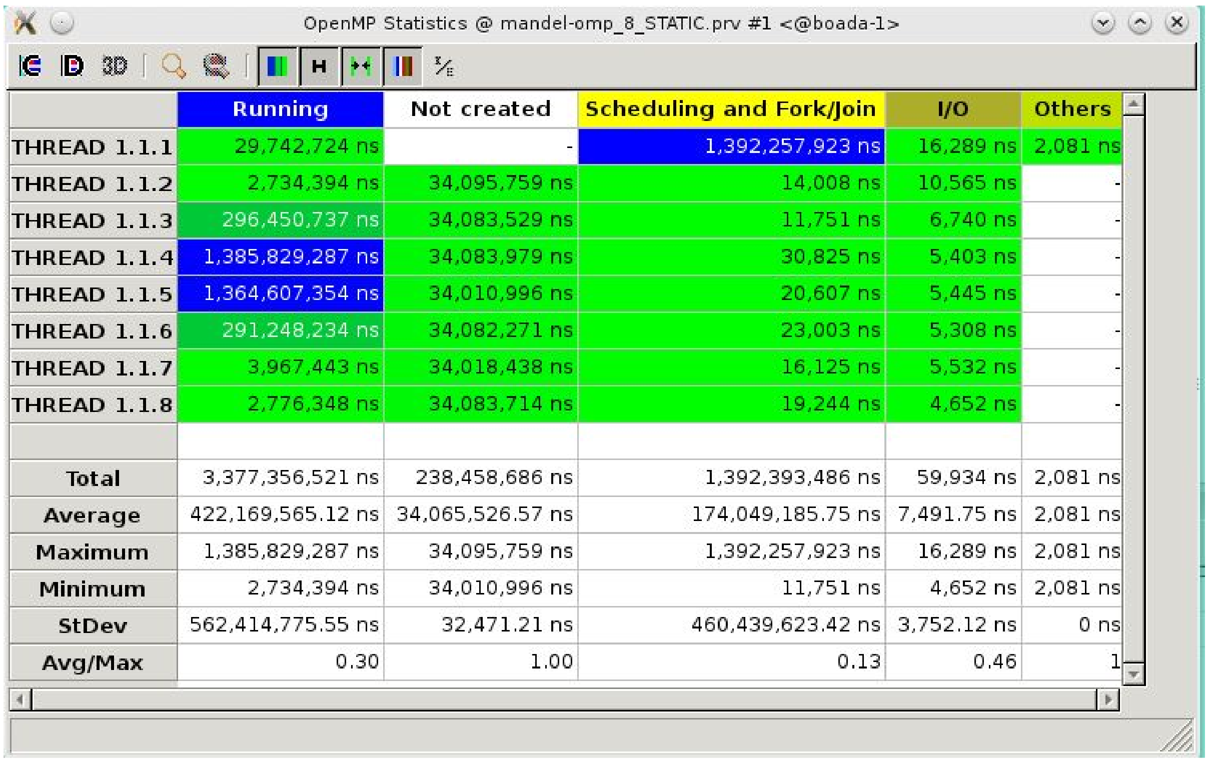




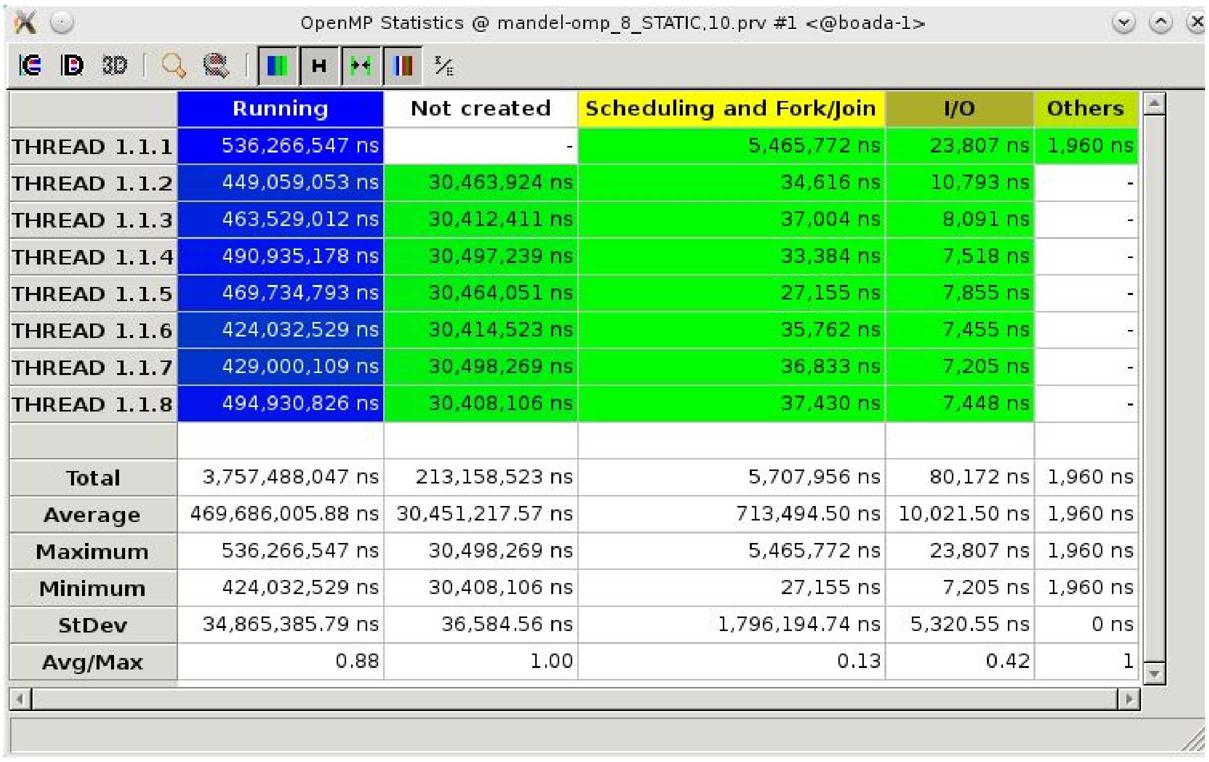




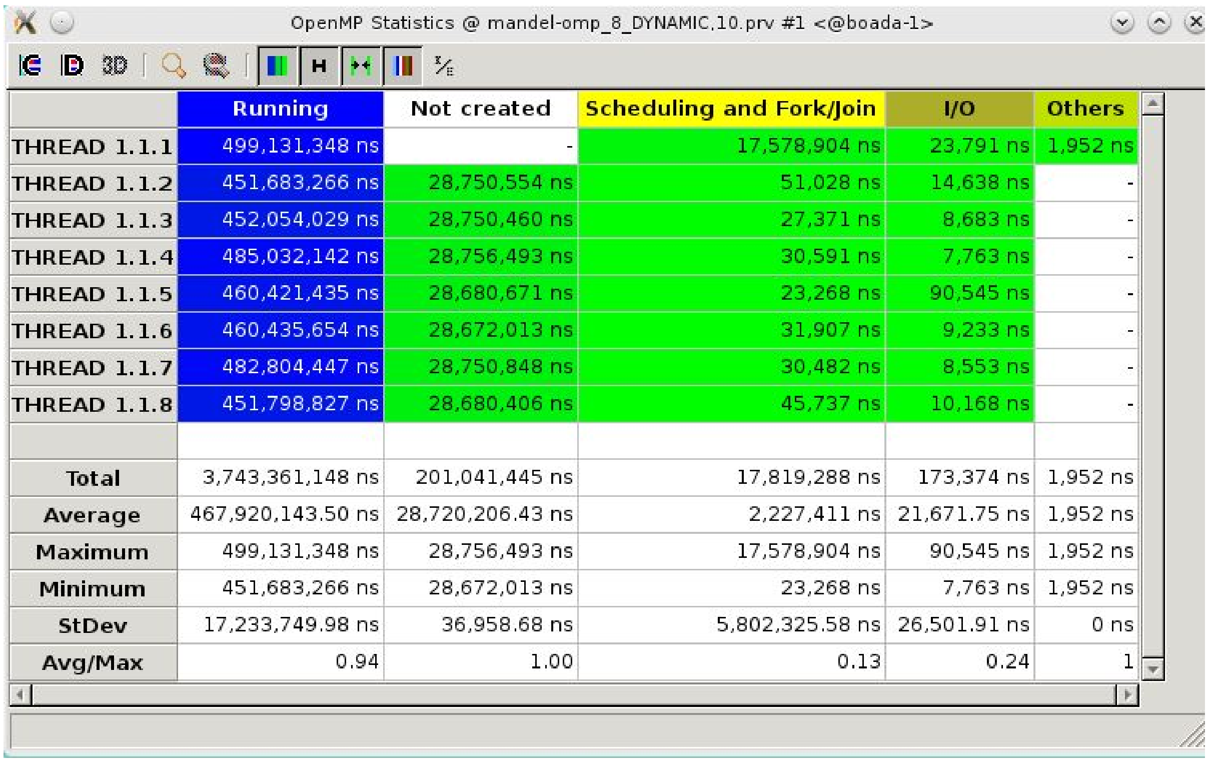
Plots with OMP_Profile filter:


OpenMP Statistics @ mandel-omp_8_STATIC.prv #1 <@boada-1>

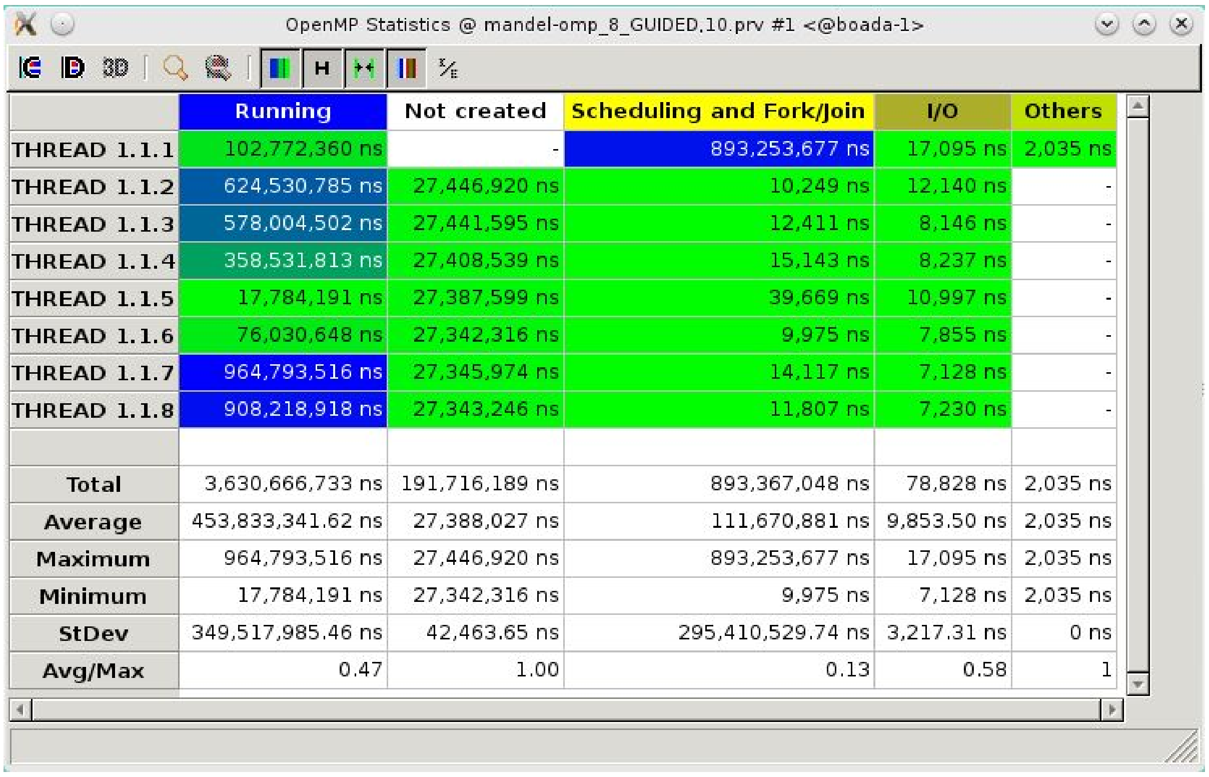
	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	29,742,724 ns	-	1,392,257,923 ns	16,289 ns	2,081 ns
THREAD 1.1.2	2,734,394 ns	34,095,759 ns	14,008 ns	10,565 ns	-
THREAD 1.1.3	296,450,737 ns	34,083,529 ns	11,751 ns	6,740 ns	-
THREAD 1.1.4	1,385,829,287 ns	34,083,979 ns	30,825 ns	5,403 ns	-
THREAD 1.1.5	1,364,607,354 ns	34,010,996 ns	20,607 ns	5,445 ns	-
THREAD 1.1.6	291,248,234 ns	34,082,271 ns	23,003 ns	5,308 ns	-
THREAD 1.1.7	3,967,443 ns	34,018,438 ns	16,125 ns	5,532 ns	-
THREAD 1.1.8	2,776,348 ns	34,083,714 ns	19,244 ns	4,652 ns	-
Total	3,377,356,521 ns	238,458,686 ns	1,392,393,486 ns	59,934 ns	2,081 ns
Average	422,169,565.12 ns	34,065,526.57 ns	174,049,185.75 ns	7,491.75 ns	2,081 ns
Maximum	1,385,829,287 ns	34,095,759 ns	1,392,257,923 ns	16,289 ns	2,081 ns
Minimum	2,734,394 ns	34,010,996 ns	11,751 ns	4,652 ns	2,081 ns
StDev	562,414,775.55 ns	32,471.21 ns	460,439,623.42 ns	3,752.12 ns	0 ns
Avg/Max	0.30	1.00	0.13	0.46	1


OpenMP Statistics @ mandel-omp_8_STATIC,10.prv #1 <@boada-1>

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	536,266,547 ns	-	5,465,772 ns	23,807 ns	1,960 ns
THREAD 1.1.2	449,059,053 ns	30,463,924 ns	34,616 ns	10,793 ns	-
THREAD 1.1.3	463,529,012 ns	30,412,411 ns	37,004 ns	8,091 ns	-
THREAD 1.1.4	490,935,178 ns	30,497,239 ns	33,384 ns	7,518 ns	-
THREAD 1.1.5	469,734,793 ns	30,464,051 ns	27,155 ns	7,855 ns	-
THREAD 1.1.6	424,032,529 ns	30,414,523 ns	35,762 ns	7,455 ns	-
THREAD 1.1.7	429,000,109 ns	30,498,269 ns	36,833 ns	7,205 ns	-
THREAD 1.1.8	494,930,826 ns	30,408,106 ns	37,430 ns	7,448 ns	-
Total	3,757,488,047 ns	213,158,523 ns	5,707,956 ns	80,172 ns	1,960 ns
Average	469,686,005.88 ns	30,451,217.57 ns	713,494.50 ns	10,021.50 ns	1,960 ns
Maximum	536,266,547 ns	30,498,269 ns	5,465,772 ns	23,807 ns	1,960 ns
Minimum	424,032,529 ns	30,408,106 ns	27,155 ns	7,205 ns	1,960 ns
StDev	34,865,385.79 ns	36,584.56 ns	1,796,194.74 ns	5,320.55 ns	0 ns
Avg/Max	0.88	1.00	0.13	0.42	1


OpenMP Statistics @ mandel-omp_8_DYNAMIC,10.prv #1 <@boada-1>

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	499,131,348 ns	-	17,578,904 ns	23,791 ns	1,952 ns
THREAD 1.1.2	451,683,266 ns	28,750,554 ns	51,028 ns	14,638 ns	-
THREAD 1.1.3	452,054,029 ns	28,750,460 ns	27,371 ns	8,683 ns	-
THREAD 1.1.4	485,032,142 ns	28,756,493 ns	30,591 ns	7,763 ns	-
THREAD 1.1.5	460,421,435 ns	28,680,671 ns	23,268 ns	90,545 ns	-
THREAD 1.1.6	460,435,654 ns	28,672,013 ns	31,907 ns	9,233 ns	-
THREAD 1.1.7	482,804,447 ns	28,750,848 ns	30,482 ns	8,553 ns	-
THREAD 1.1.8	451,798,827 ns	28,680,406 ns	45,737 ns	10,168 ns	-
Total	3,743,361,148 ns	201,041,445 ns	17,819,288 ns	173,374 ns	1,952 ns
Average	467,920,143.50 ns	28,720,206.43 ns	2,227,411 ns	21,671.75 ns	1,952 ns
Maximum	499,131,348 ns	28,756,493 ns	17,578,904 ns	90,545 ns	1,952 ns
Minimum	451,683,266 ns	28,672,013 ns	23,268 ns	7,763 ns	1,952 ns
StDev	17,233,749.98 ns	36,958.68 ns	5,802,325.58 ns	26,501.91 ns	0 ns
Avg/Max	0.94	1.00	0.13	0.24	1


OpenMP Statistics @ mandel-omp_8_GUIDED.10.prv #1 <@boada-1>

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	102,772,360 ns	-	893,253,677 ns	17,095 ns	2,035 ns
THREAD 1.1.2	624,530,785 ns	27,446,920 ns	10,249 ns	12,140 ns	-
THREAD 1.1.3	578,004,502 ns	27,441,595 ns	12,411 ns	8,146 ns	-
THREAD 1.1.4	358,531,813 ns	27,408,539 ns	15,143 ns	8,237 ns	-
THREAD 1.1.5	17,784,191 ns	27,387,599 ns	39,669 ns	10,997 ns	-
THREAD 1.1.6	76,030,648 ns	27,342,316 ns	9,975 ns	7,855 ns	-
THREAD 1.1.7	964,793,516 ns	27,345,974 ns	14,117 ns	7,128 ns	-
THREAD 1.1.8	908,218,918 ns	27,343,246 ns	11,807 ns	7,230 ns	-
Total	3,630,666,733 ns	191,716,189 ns	893,367,048 ns	78,828 ns	2,035 ns
Average	453,833,341.62 ns	27,388,027 ns	111,670,881 ns	9,853.50 ns	2,035 ns
Maximum	964,793,516 ns	27,446,920 ns	893,253,677 ns	17,095 ns	2,035 ns
Minimum	17,784,191 ns	27,342,316 ns	9,975 ns	7,128 ns	2,035 ns
StDev	349,517,985.46 ns	42,463.65 ns	295,410,529.74 ns	3,217.31 ns	0 ns
Avg/Max	0.47	1.00	0.13	0.58	1

(time in ns)	static	static,10	dynamic,10	guided,10
Running average time per thread	422,169,565.12	469,686,005.88	467,920,143.50	453,833,341.62
Execution unbalance (average time divided by maximum time)	0.30	0.88	0.94	0.47
SchedForkJoin (average time per thread or time if only one does)	174,049,185.75	713,494.50	2,227,441	111,670,881

As we can see the values of the table are obtained from the **OMP_Profile filter** from the analysis with paraver from the results the traces that the script generate.

Also from the **Plots with OMP_Profile filter,** we can see the gradient of work done by each thread and with the execution plots and scheduling plots we have a graphical idea of what is happening.

Static is slower than static,10 because in static 10 we have a better execution unbalance because the chunk is smaller.

Dynamic and static,10 will have aproximately the same execution time but in dynamic implementation we will have even a better execution unbalance because when a thread finishes its task it will be assigned a new one and with the static one each thread does only have a certain amount of tasks.

guided,10 does have a worse time than static,10 and dynamic,10 because the execution balance is bad so we are not using at all the potential of all our threads at the same time. Why it's not balanced? Well the size of chunks will be decrementing at each iteration but at the firsts executions the chunk is really big (aprox the same as static).

In **guided,10** and **static** we have such unbalance because one thread (as we can see in the scheduling profile) basically does scheduling fork/join job.

In conclusion we can say that execution time will be defined by factors as:

- Granularity.
- Schedule planification used.
- Task unbalance.
- Overhead (related with the granularity and the schedule).
- Other factors non related with our program.