

# Task decomposition analysis for the Mandelbrot set computation

PARALLELISM 2019/2020 Q1

Alejandro Gallego  
Oriol Catalán

Group 1203  
14/11/2019

## 4.0 Introduction

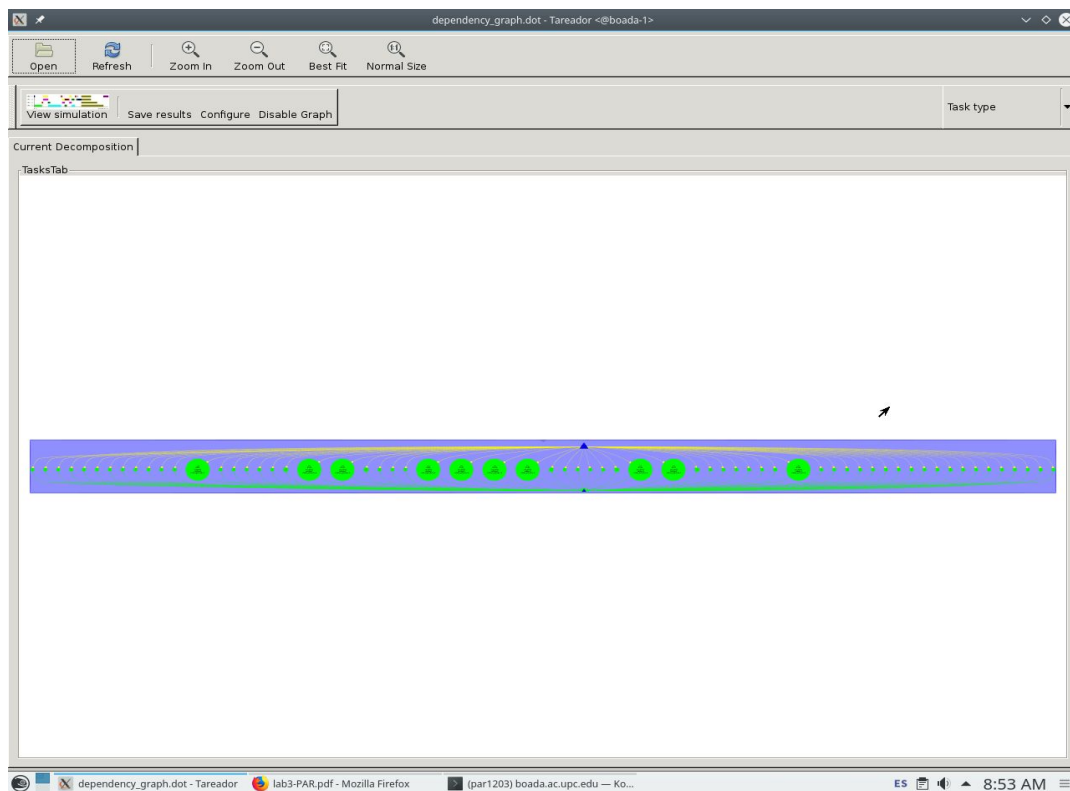
For this assignment, we have used the Mandelbrot set, this algorithm is used to determine if a point in a complex domain belongs or not to the Mandelbrot set, delivering an output shape, which illustrates the previous mentioned behaviour.

The Mandelbrot set will be used to help us explore the different parallelization strategies, such as Point and Row decomposition, as well as three different task utilities in Point decomposition. We will show the data and code extracted from the assignment and use them to discuss which strategy is better in this situation.

## 4.1 Task decomposition and granularity analysis

As mentioned earlier, there are two types of task decomposition in this case: Point and Row.

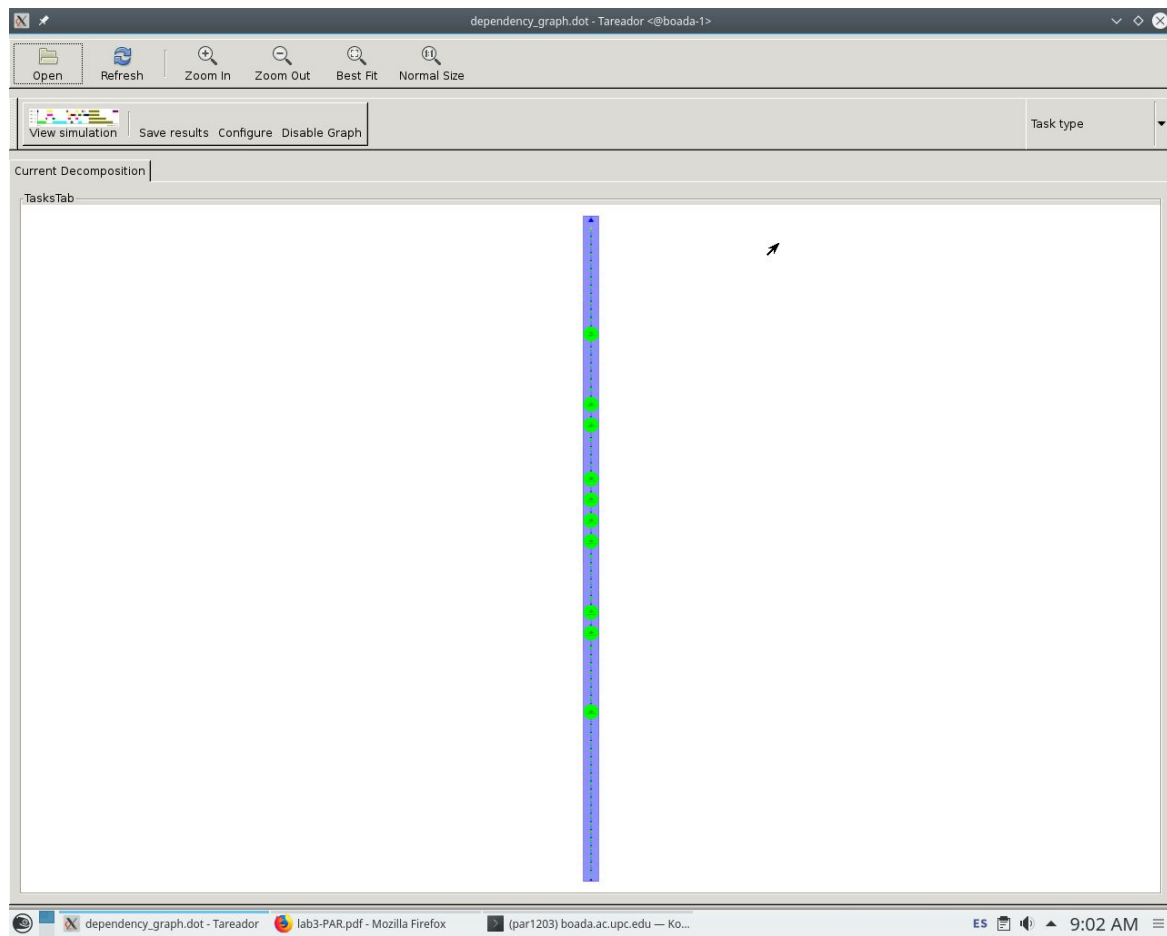
**Point decomposition** consists of parallelize the code in each iteration of the column loop, creating a task every time we visit a new column of a determined row. This strategy has very fine granularity and can create an inefficient task creating overhead. In **Figure 1** we can see the granularity in a more visual way:



*Figure 1: Dependency graph for point decomposition strategy in mandel-tar.c*

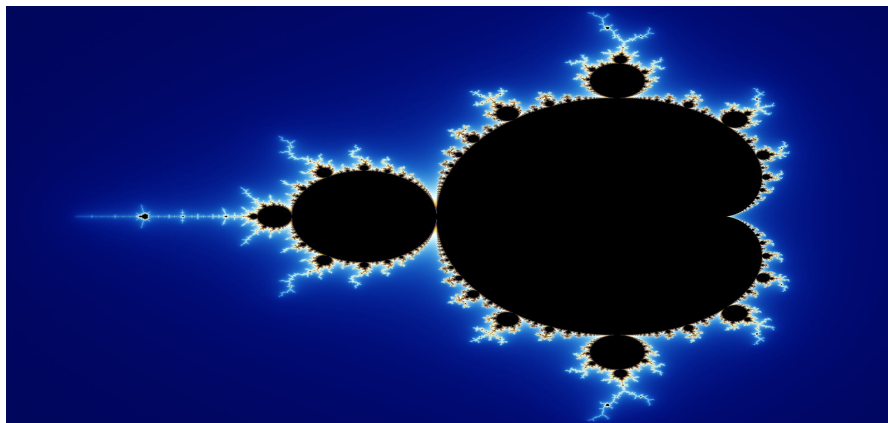
With Tareador, we display the dependency graph, in which every circle, triangle or form is a task. Known this information, not only it is quite clear that with point decomposition the number of tasks is greater, but also the workload of each task is very little. The tasks are executed in parallel, too.

At this point we were doing our research with an “incomplete” version of the code, because of the lack of a Mandelbrot Set image. If we complete the code to output the fractal image we will see differences in our dependency graph (**Figure 1.1**)

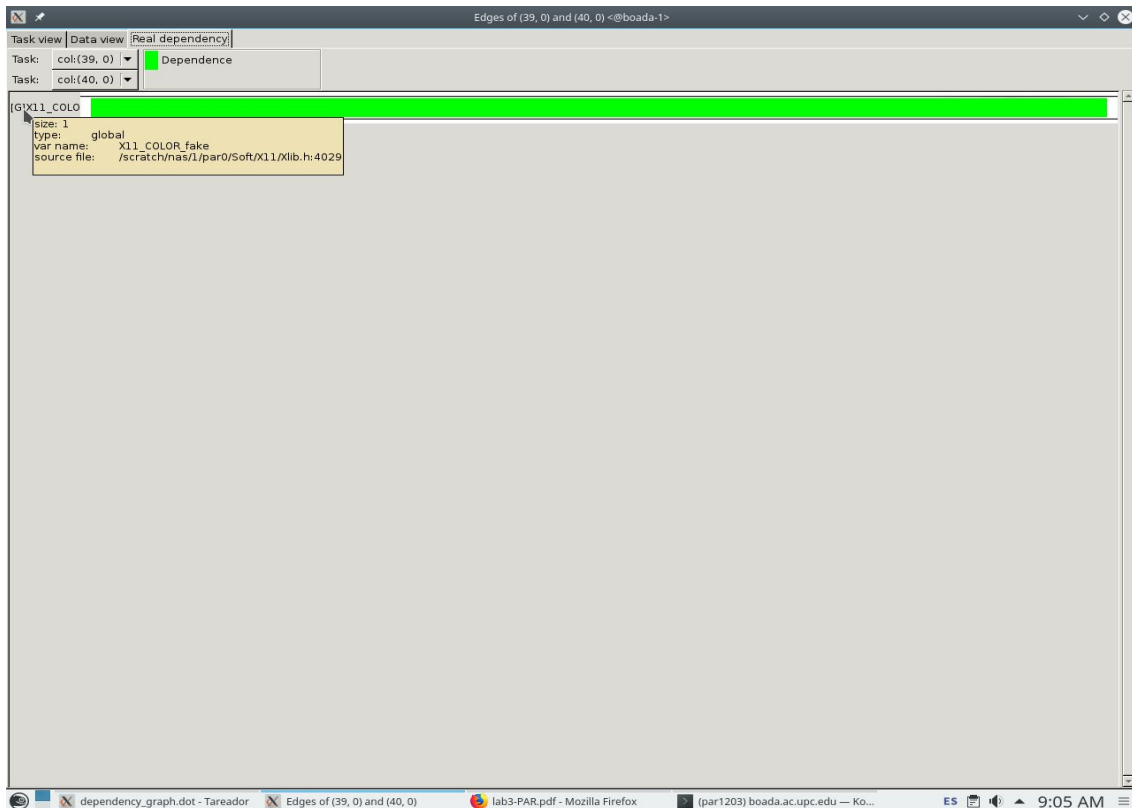


*Figure 1.1: Dependency graph with point decomposition strategy for `mandeld-tar.c`*

Now the parallelization has disappeared, and we face a sequential execution of the tasks. Why has this happened? At implementing the new part of the code we use a particular variable (**Figure 1.3**) that is dependant. This variable is used inside the `__DISPLAY__` if that it is true if the execution requires the output of the Mandelbrot Set graphic (**Figure 1.2**)



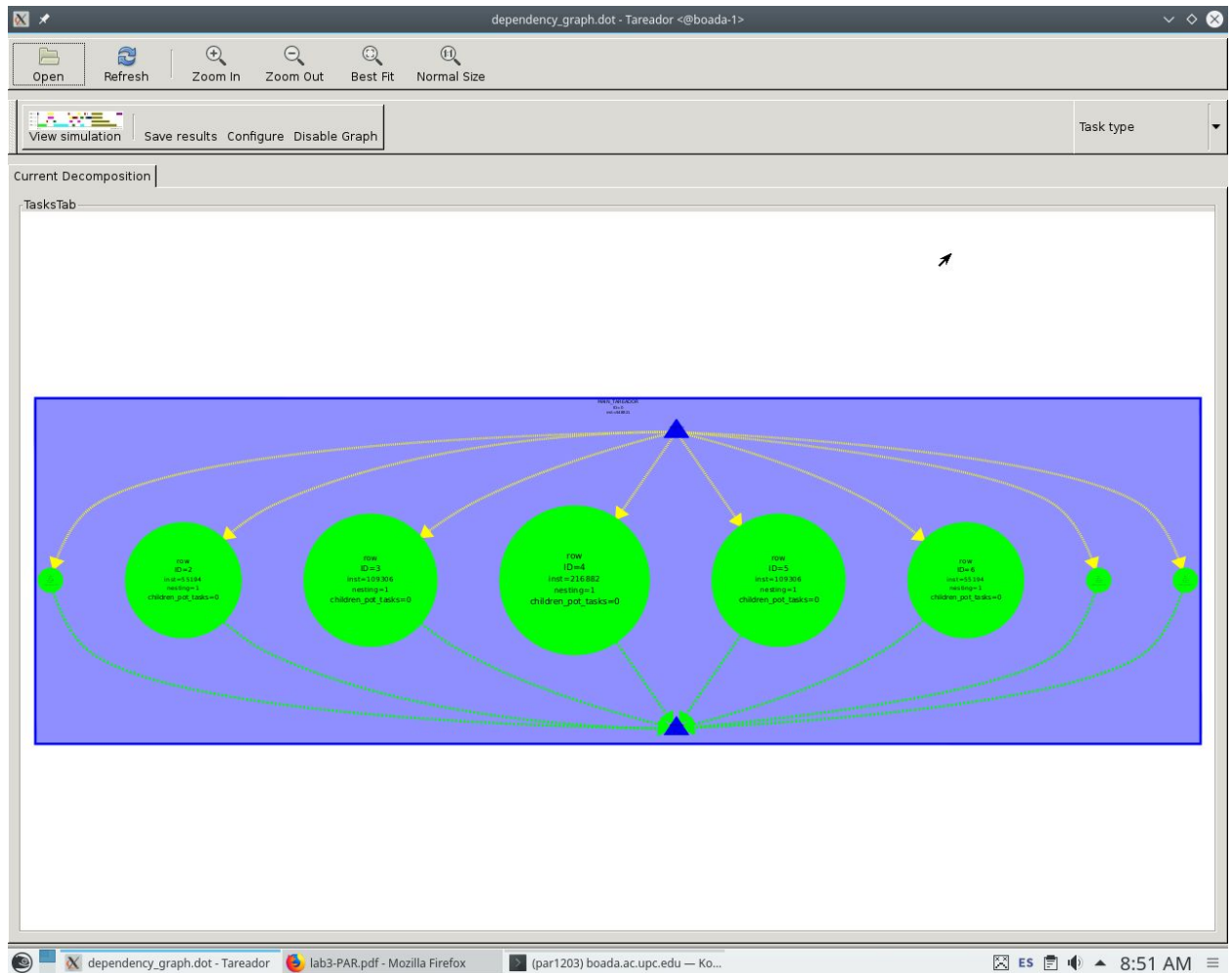
*Figure 1.2: Mandelbrot Set image*



*Figure 1.3 Dependant variable in the execution of mandeld-tar.c*

This variable has caused the tasks created can not execute in parallel, furthermore if we look at the header file of the X11 library, we can observe that `X11_COLOR_fake` is a global variable used to create fake dependencies. We can protect this part using a pragma clause that will privatize this variable in order to avoid this dependency problem. For example, `#pragma omp parallel private(X11_COLOR_fake)`.

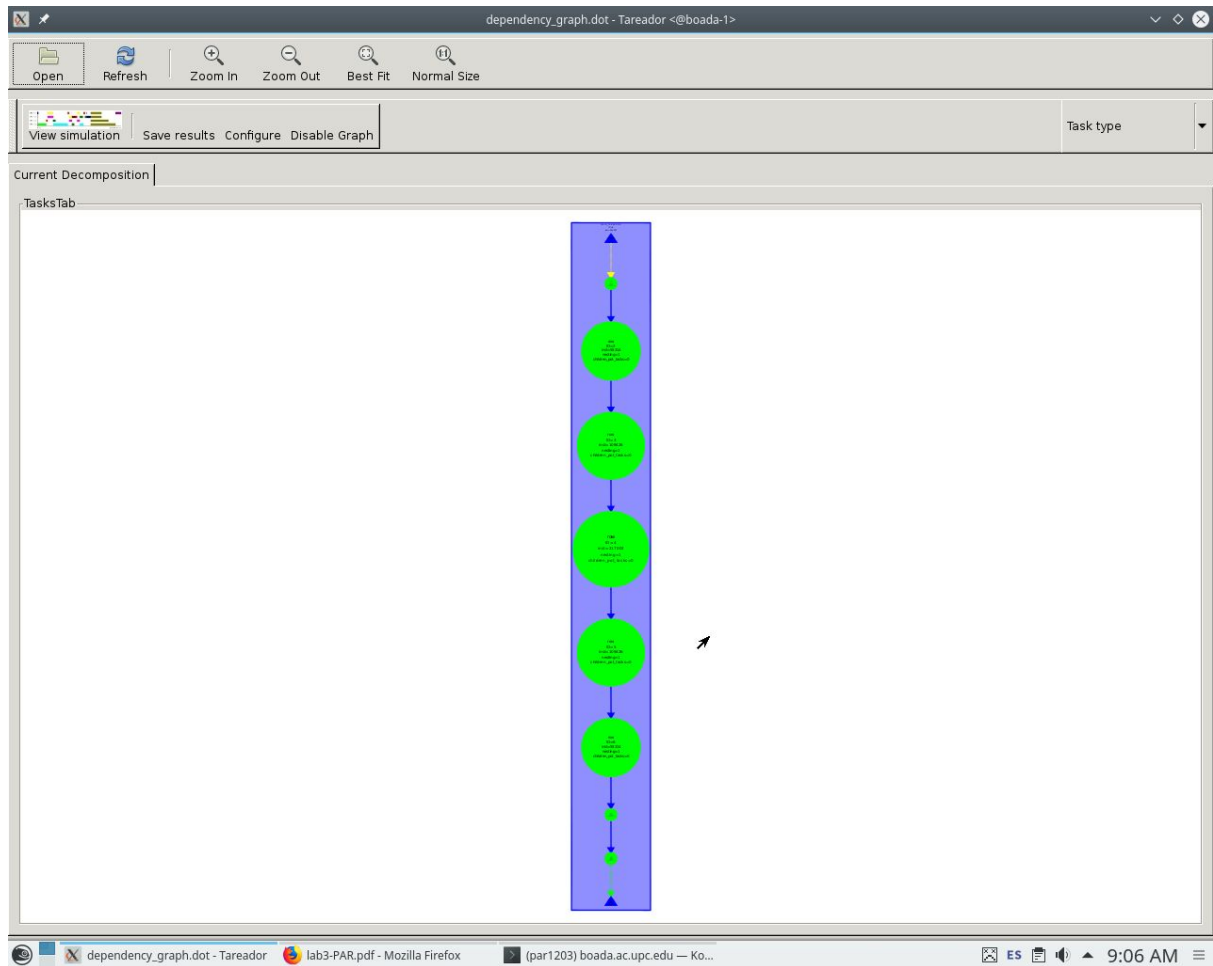
On the other hand, we have the **Row Decomposition** strategy, which creates a task every time we visit a new row. As we will see this strategy does not have such fine granularity as the Point Decomposition one (**Figure 1.4**).



*Figure 1.4: Dependency graph with row decomposition strategy for mandel-tar.c*

As the image shows, now we have a lesser granularity than before, the number of tasks created is minor as it is the task creating overhead. However, the tasks created are heavier, the workload is greater than with point decomposition, and the parallelization is successful.

At the time of executing the graphical version of the code, there is no difference with the point decomposition one, unless the number of tasks created (**Figure 1.5**)



*Figure 1.5 Dependency graph with row decomposition strategy for mandeld-tar.c*

As explained before we continue to see a sequential execution when graphic version is required.

At this point we can have the necessary information to decide which one of the decomposition strategies is more favorable in the execution of the Mandelbrot set algorithm. We assure that a point decomposition is more beneficial in terms of parallelization and tasks workload, although the task creation overhead. Even though, when a graphic version of the Mandelbrot set is required we suggest using a row decomposition strategy, because the parallelization pros of the point strategy are useless in this case.

## 4.2 Point decomposition in OpenMP

### 1.First Version (Task)

```
/* Calculate points and save/display */  
for (int row = 0; row < height; ++row) {  
    #pragma omp parallel  
    #pragma omp single  
    for (int col = 0; col < width; ++col) {  
        #pragma omp task firstprivate(col)  
    }
```

*Figure 2.1 Code of first version*

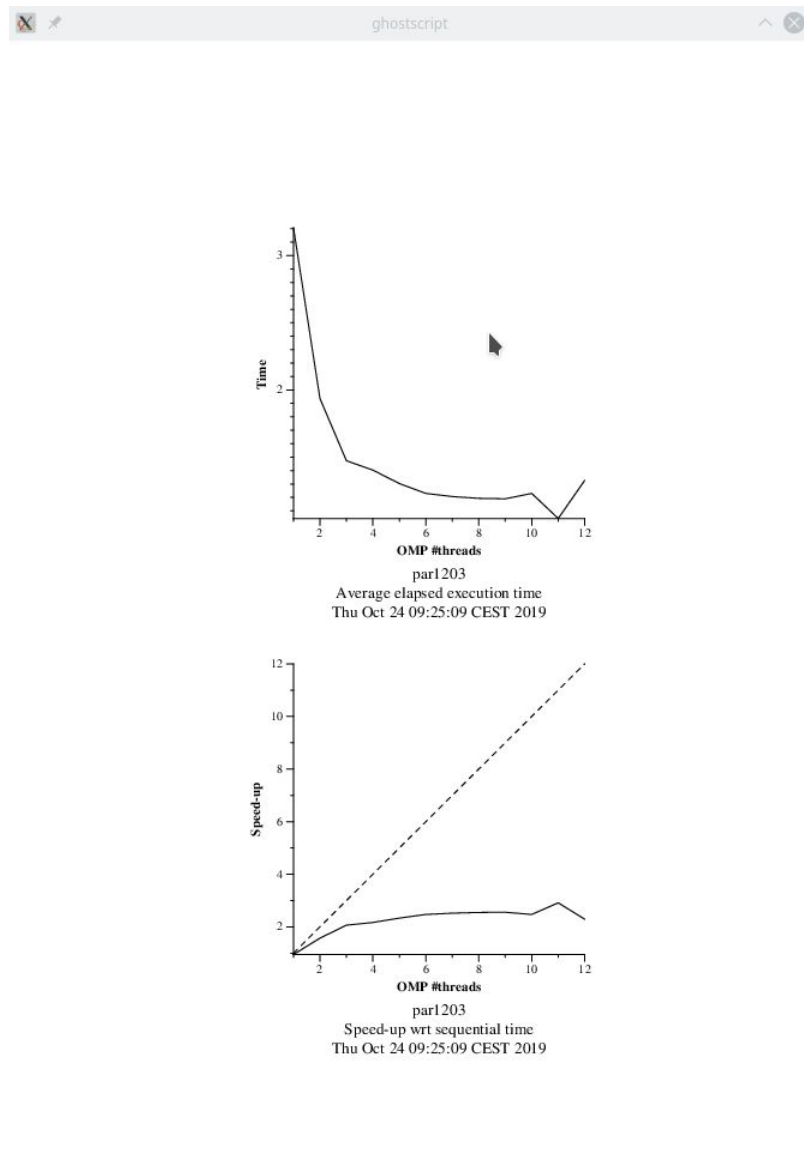
The first version, in Figure 2.1, is worse than a sequential version because the time of overheads of parallel region, created in each iteration of row's loop, is very big; the sequential code doesn't have to do any overhead and is faster.



*Figure 2.2 Timeline of first version*

The major part of the timeline is used for creation of tasks. As mentioned in the introduction the granularity is such excessive that the execution of the tasks is ridiculously little compared with the creation and synchronisation of these tasks.





*Figure 2.3 Strong scalability plots for version 1*

The speed-up plot gives us a clear example of what we were declaring early. This version of point decomposition performs a poor execution decreasing the speed-up and augmenting the execution time when we arrive at a major number of threads.

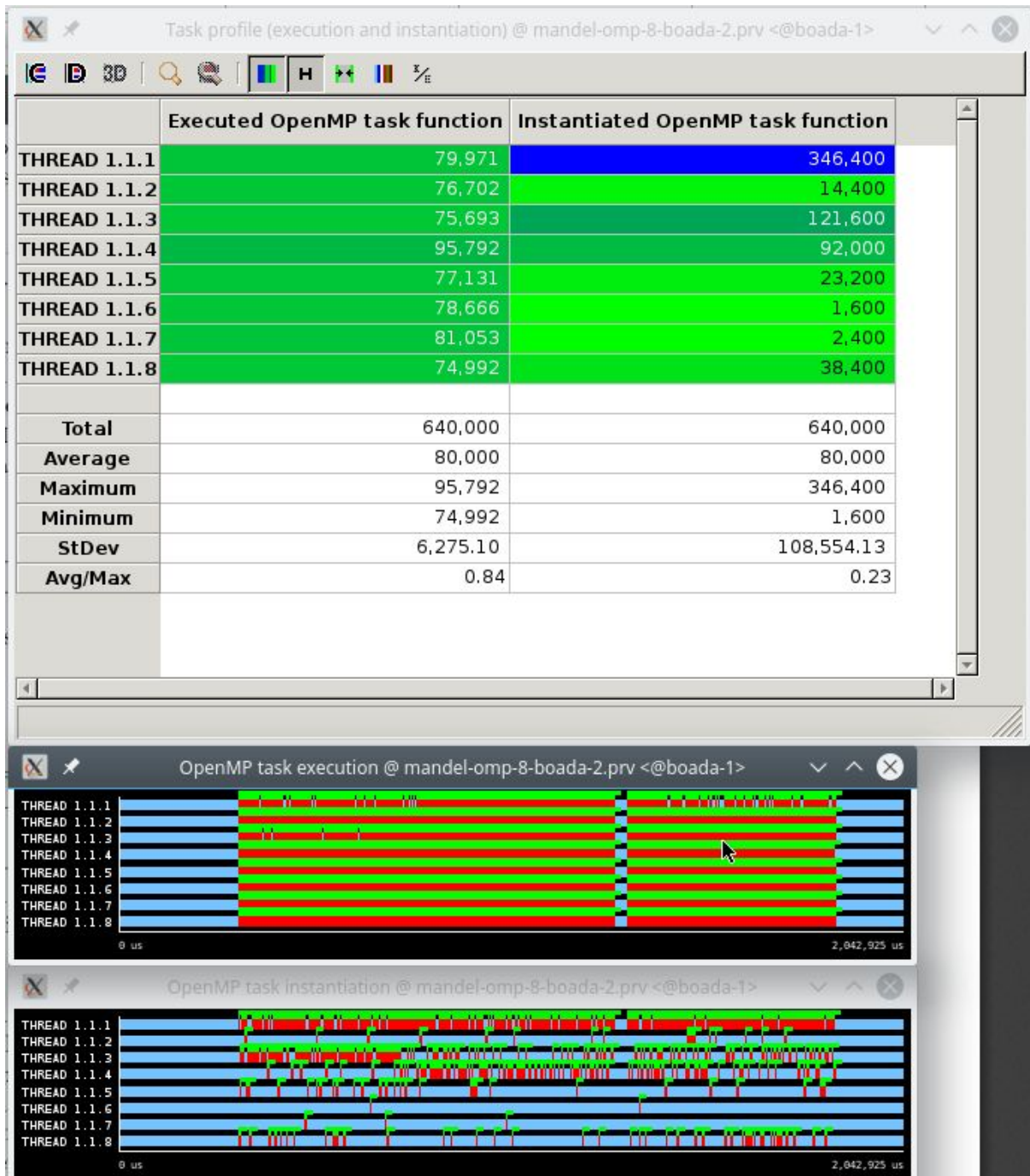


Figure 2.4 Table and tasks timelines of first version

As we can see in the table, 640.000 tasks are being created and executed. The creation of tasks are very unbalanced but the execution are more adjusted (with the values of maximum and minimum).

## 2.Second Version (First version + taskwait)

```
/* Calculate points and save/display */
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {

        #pragma omp task firstprivate(row, col)
        {
            ...
        }
    }
    #pragma omp taskwait
}
```

Figure 2.5 Code of second version

In this version of code, we have only added the taskwait clause at the end of column's loop that waits for the children threads to be finished.



Figure 2.6 Timeline of second version

This timeline shows us that the first thread create the parallel region and creates all of the tasks.

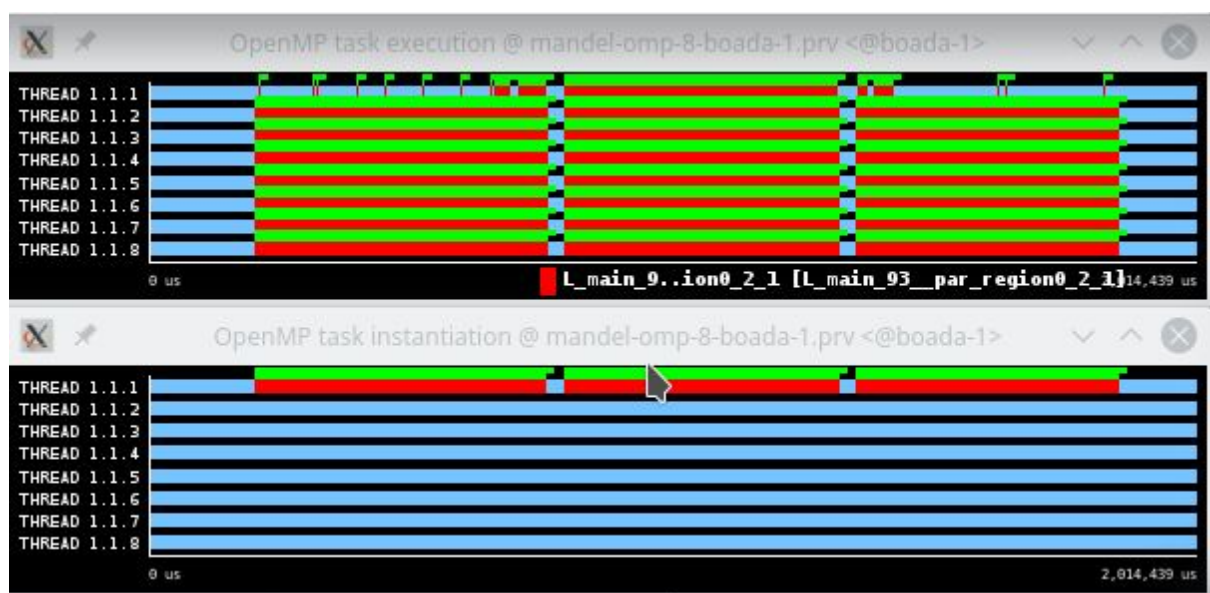
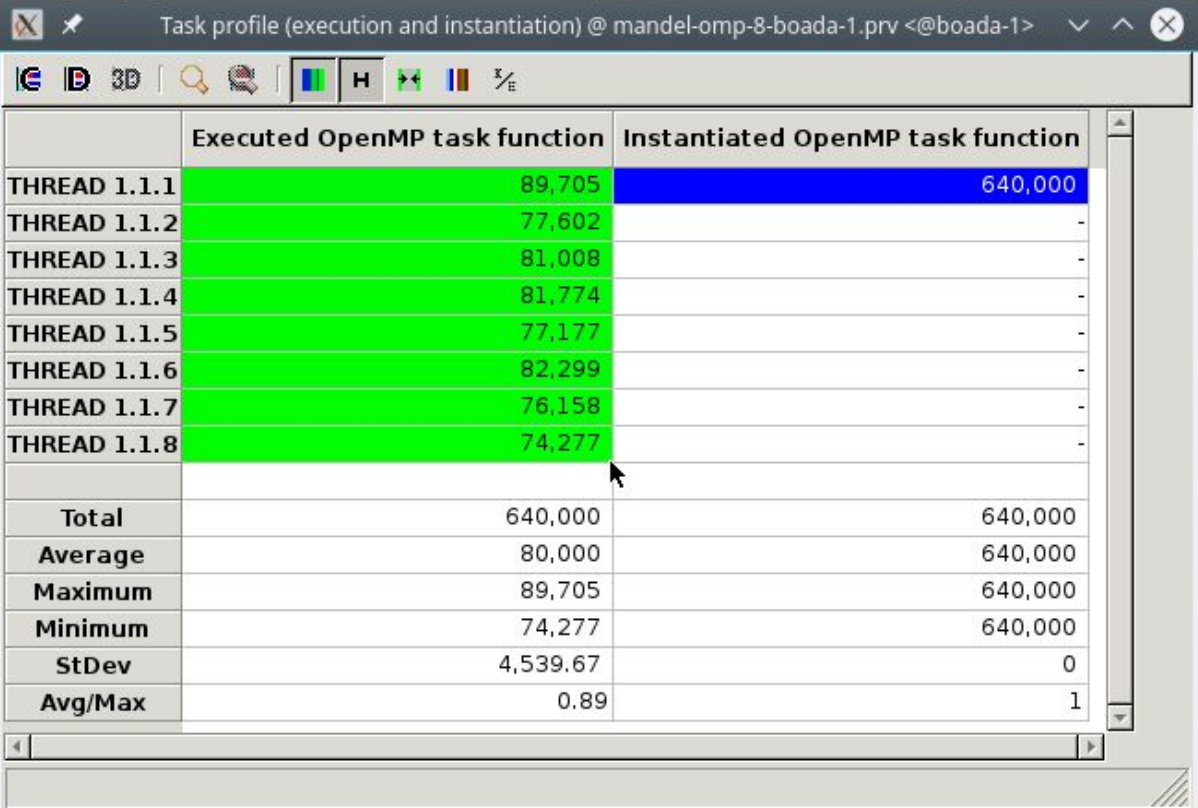


Figure 2.7 Tasks timelines of second version

These timelines shows how are created the tasks by only one thread. The execution is made by all the threads.

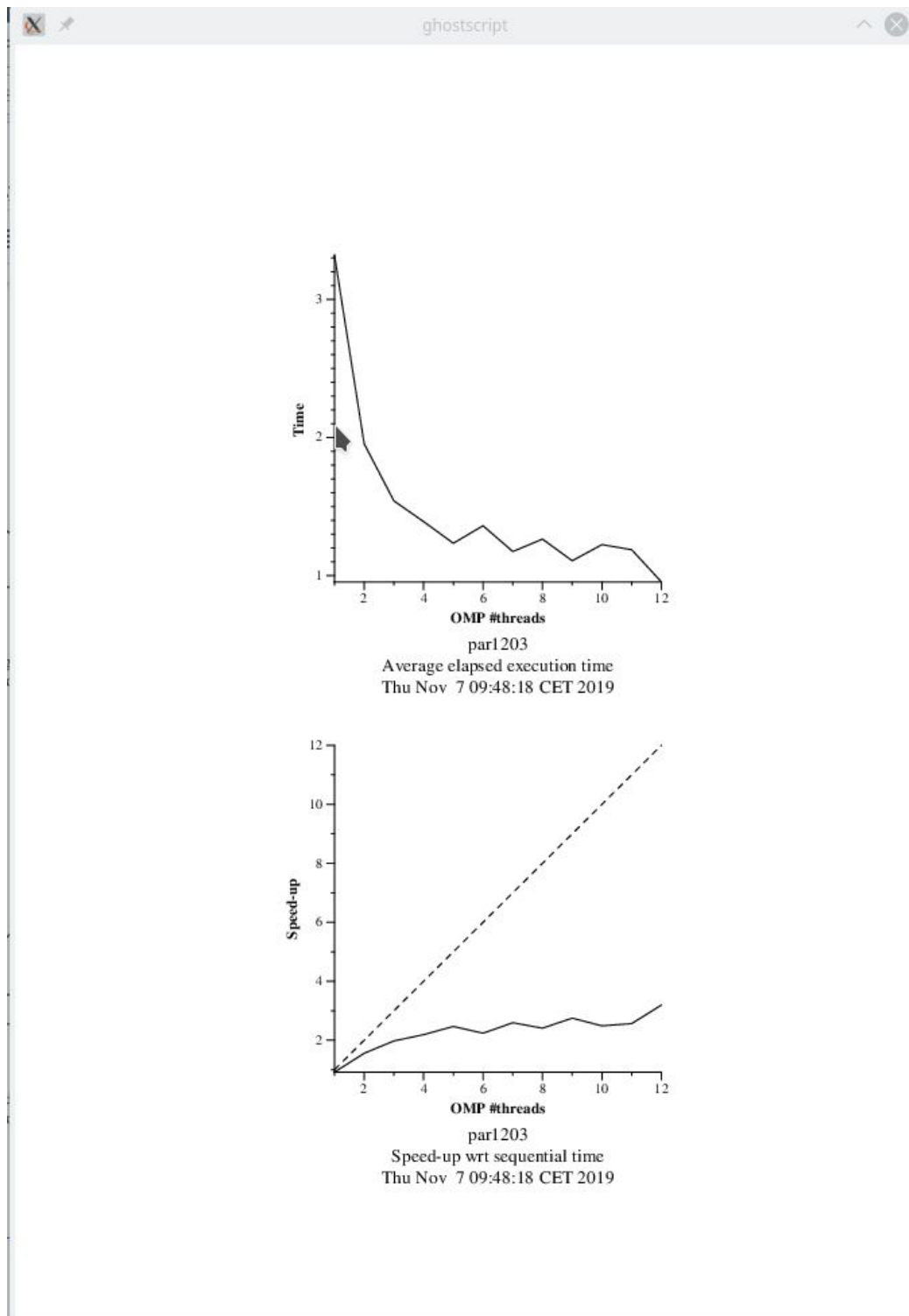


The screenshot shows a window titled "Task profile (execution and instantiation) @ mandel-omp-8-boada-1.prv <@boada-1>". It contains a table with two main columns: "Executed OpenMP task function" and "Instantiated OpenMP task function". The table lists data for eight threads (THREAD 1.1.1 to 1.1.8) and a summary section with rows for Total, Average, Maximum, Minimum, StDev, and Avg/Max. Thread 1.1.1 has a maximum value of 89,705 in the execution column and 640,000 in the instantiation column. All other threads have a maximum value of 74,277 in the execution column and 0 in the instantiation column. The summary rows show a total of 640,000 for both execution and instantiation, with an average of 80,000, a standard deviation of 4,539.67, and an average/maximum ratio of 0.89.

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	89,705	640,000
THREAD 1.1.2	77,602	-
THREAD 1.1.3	81,008	-
THREAD 1.1.4	81,774	-
THREAD 1.1.5	77,177	-
THREAD 1.1.6	82,299	-
THREAD 1.1.7	76,158	-
THREAD 1.1.8	74,277	-
<b>Total</b>	640,000	640,000
<b>Average</b>	80,000	640,000
<b>Maximum</b>	89,705	640,000
<b>Minimum</b>	74,277	640,000
<b>StDev</b>	4,539.67	0
<b>Avg/Max</b>	0.89	1

*Figure 2.8 Table of second version*

640.000 tasks are created by Thread 1.1.1 and are executed by all threads. The number of tasks and the size is the same but the difference is the creation of these.



*Figure 2.9 Strong scalability plots for version 2*

We can see that the speedup is very bad and it can't reach the ideal speedup. Also, the time is reduced too much with more threads.

### 3.Third Version (Second version without taskwait and with taskgroup)

```
/* Calculate points and save/display */
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskgroup
    for (int col = 0; col < width; ++col) {
        #pragma omp task firstprivate(row, col)
    }
}
```

Figure 2.10 Code of third version

In this situation, we removed the taskwait clause and we have introduced a taskgroup sentence between row and column's loops.



Figure 2.11 Timeline of third version

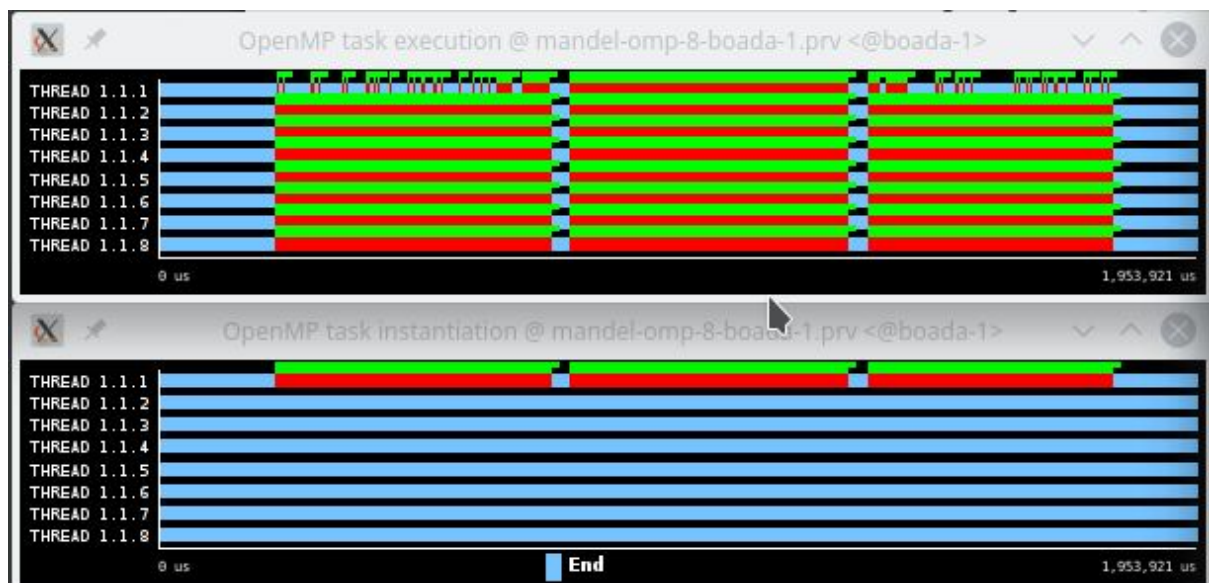
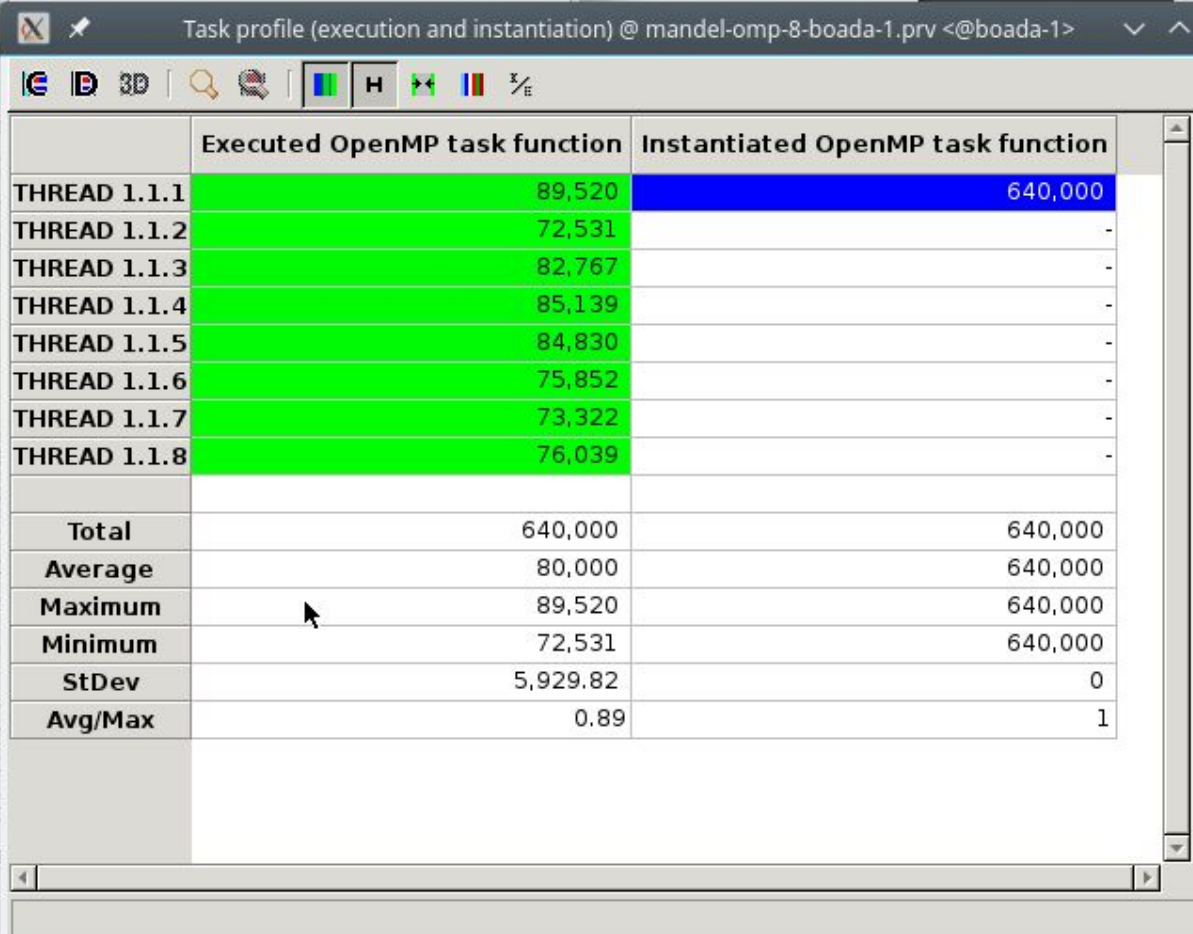


Figure 2.12 Tasks timelines of third version

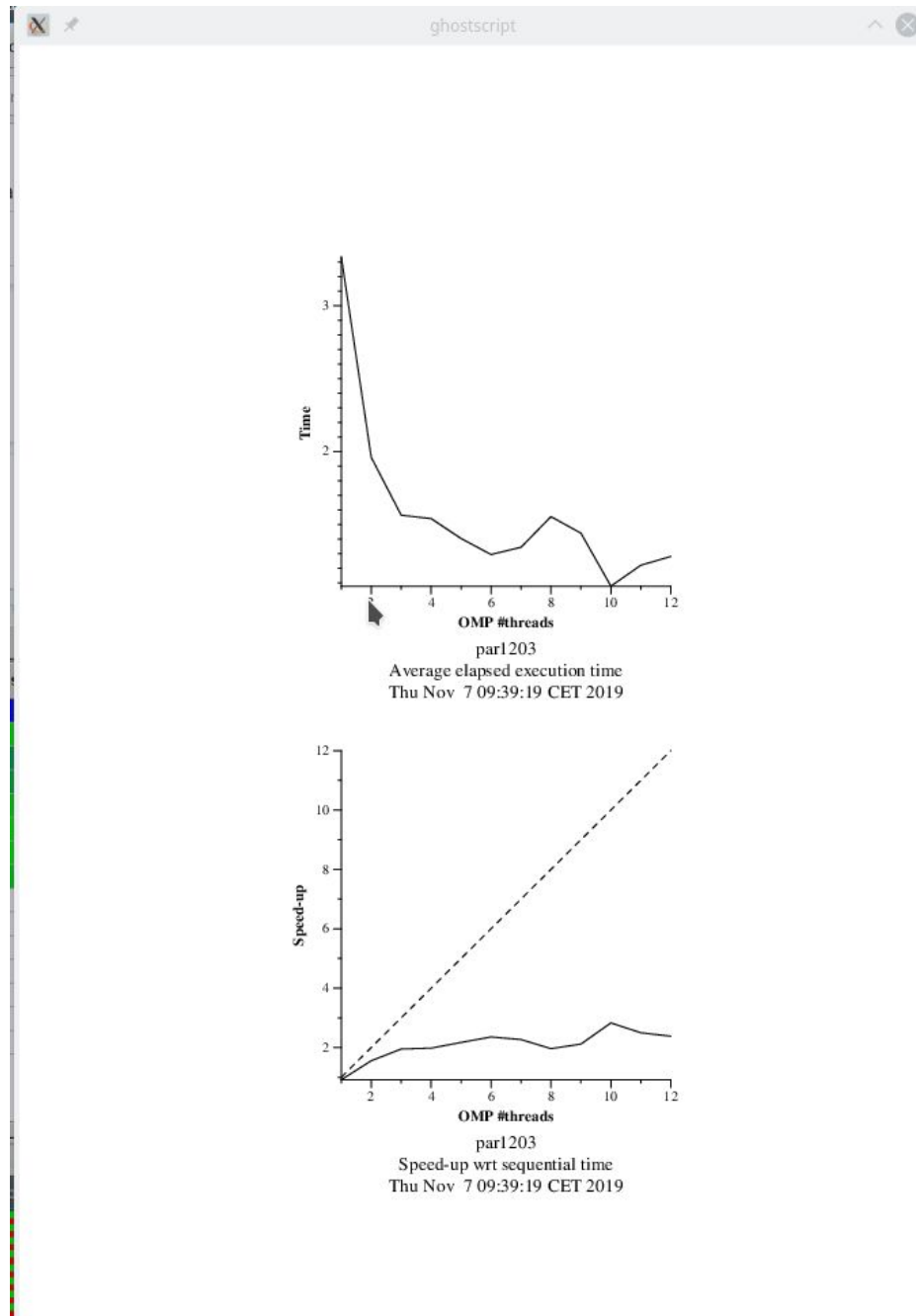
Related to timelines, the difference between second and third version is that this one, the first thread have more small regions to execute program regions, in second version there are fewer parts.



	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	89,520	640,000
THREAD 1.1.2	72,531	-
THREAD 1.1.3	82,767	-
THREAD 1.1.4	85,139	-
THREAD 1.1.5	84,830	-
THREAD 1.1.6	75,852	-
THREAD 1.1.7	73,322	-
THREAD 1.1.8	76,039	-
<b>Total</b>	640,000	640,000
<b>Average</b>	80,000	640,000
<b>Maximum</b>	89,520	640,000
<b>Minimum</b>	72,531	640,000
<b>StDev</b>	5,929.82	0
<b>Avg/Max</b>	0.89	1

*Figure 2.13 Table of third version*

The table is almost the same compared with the second version. The StDev is a little bit bigger than the previous table.



*Figure 2.14 Strong scalability plots for version 3*

There is almost no improvement between the codes of version 3 and version 2. The taskgroup clause has an implicit barrier at the end of the execution making unnecessary the taskwait clause.



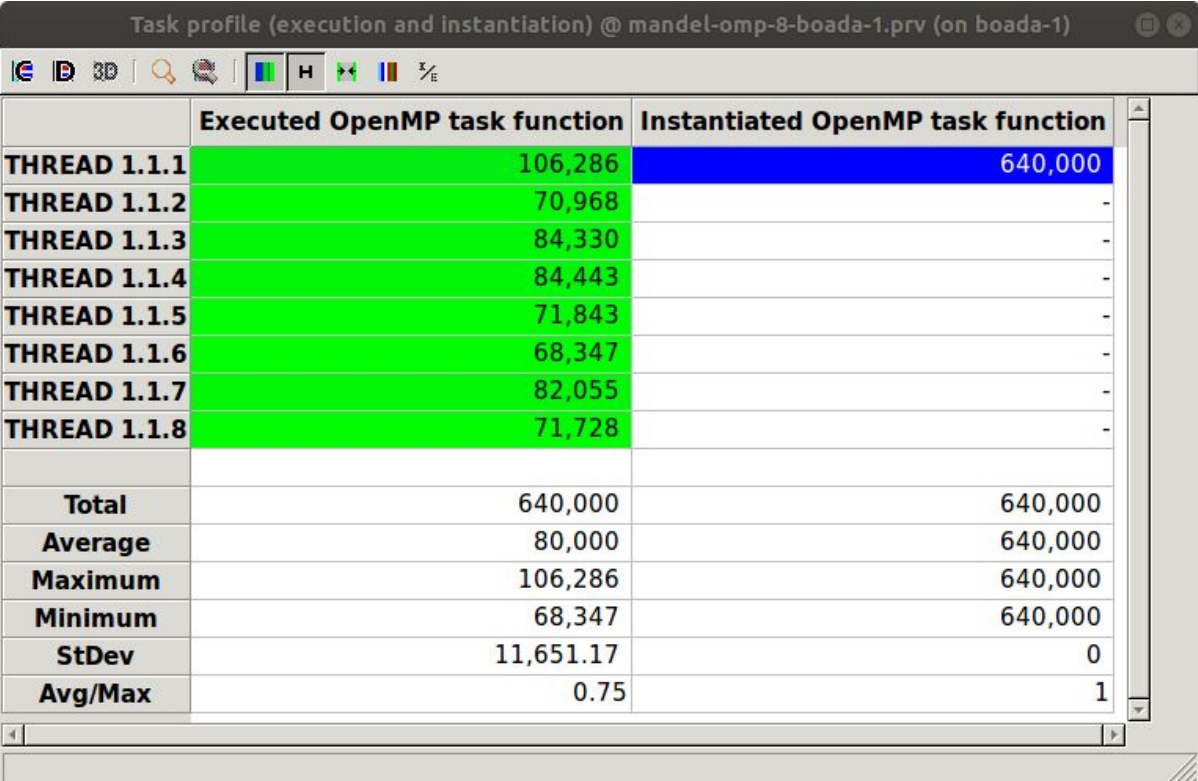
#### 4.Fourth Version (Second version without taskwait)

```
/* Calculate points and save/display */
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {

        #pragma omp task firstprivate(row, col)
    }
}
```

Figure 2.15 Code of fourth version

These code is practically the same as in the second version, but removing the taskwait.



Task profile (execution and instantiation) @ mandel-omp-8-boada-1.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	106,286	640,000
THREAD 1.1.2	70,968	-
THREAD 1.1.3	84,330	-
THREAD 1.1.4	84,443	-
THREAD 1.1.5	71,843	-
THREAD 1.1.6	68,347	-
THREAD 1.1.7	82,055	-
THREAD 1.1.8	71,728	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	106,286	640,000
Minimum	68,347	640,000
StDev	11,651.17	0
Avg/Max	0.75	1

Figure 2.16 Table of fourth version

The tables are almost the same, they just change a little in the execution of the tasks, the thread 1.1.1 executes a little more of the account, apart from the StDev that is much larger in this version compared to the second.

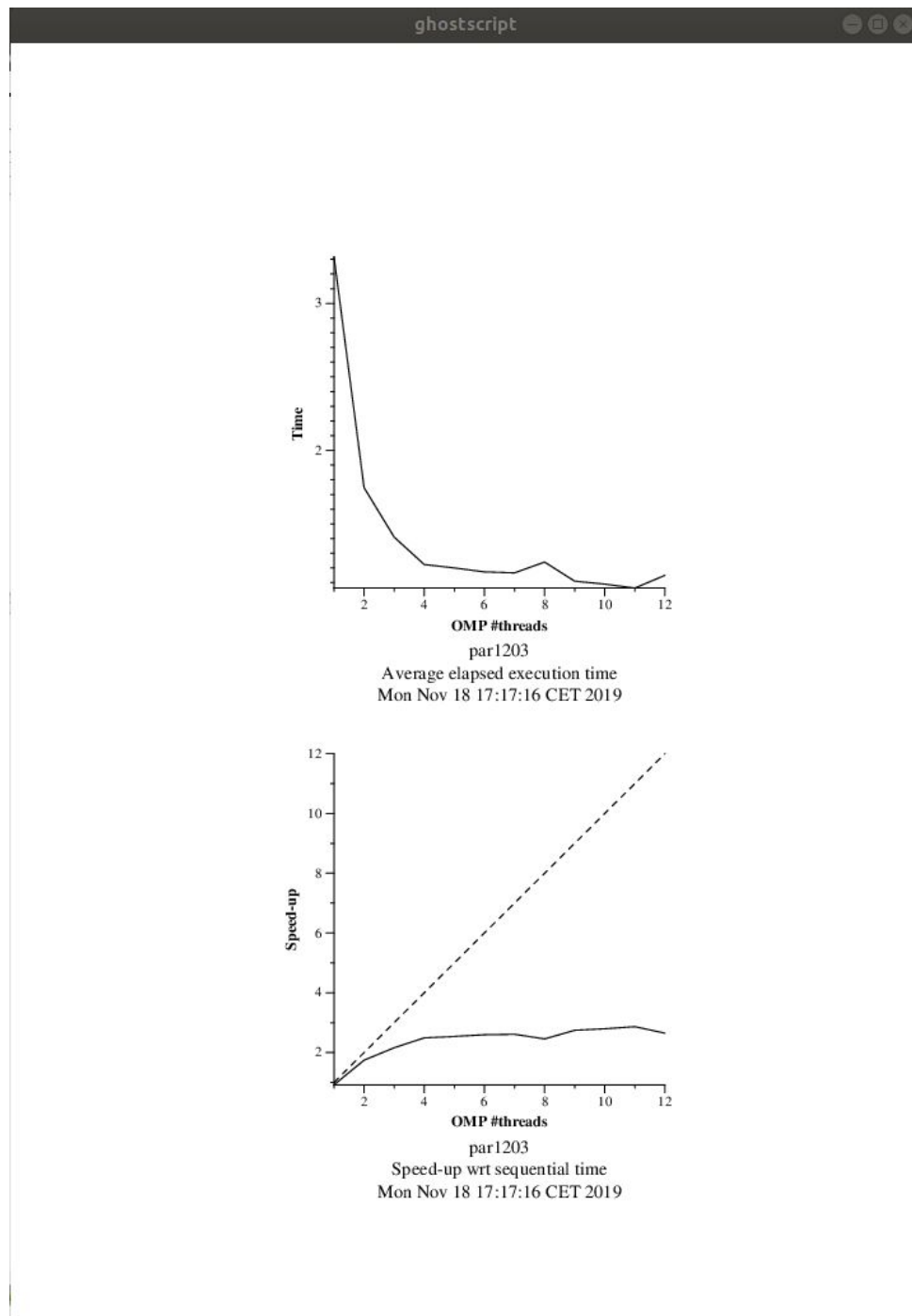


Figure 2.17 Tasks timelines of fourth version



Figure 2.18 Timelines of fourth version

We can see in the timelines that the difference is in the execution of tasks of thread 1.1.1, which has fewer regions over time, but are larger, so it executes more tasks that we see in more detail in the table.



*Figure 2.19 Strong scalability plots for version 4*

These plots show us the relation with ideal speedup, they show a very bad correlation and the real speedup can't reach that ideal level.

## 5.Fifth Version (Taskloop with graincontrol)

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) num_tasks(800)
    for (int col = 0; col < width; ++col) {
```

Figure 2.20 Code of the fifth version

This code is different than the others because it incorporates the pragma sentence taskloop with the definition of the number of tasks. Changing this value, we can get a graphics with a lot of values of num\_tasks and the behaviour of the execution time.

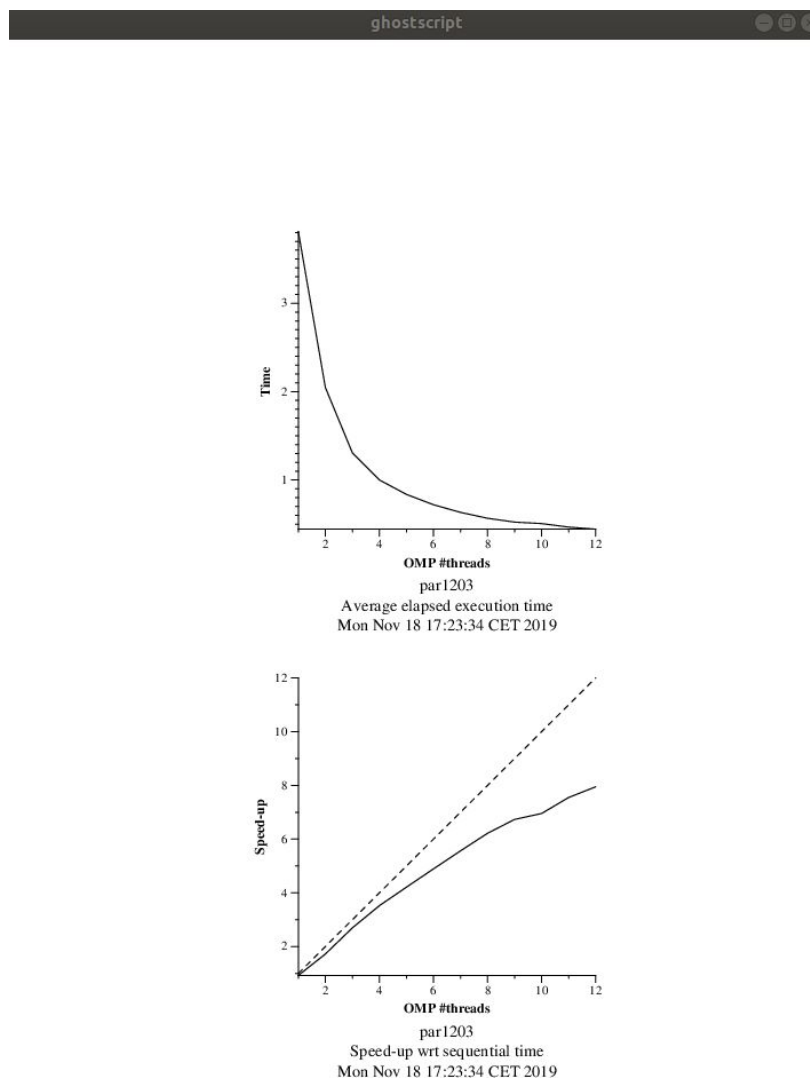
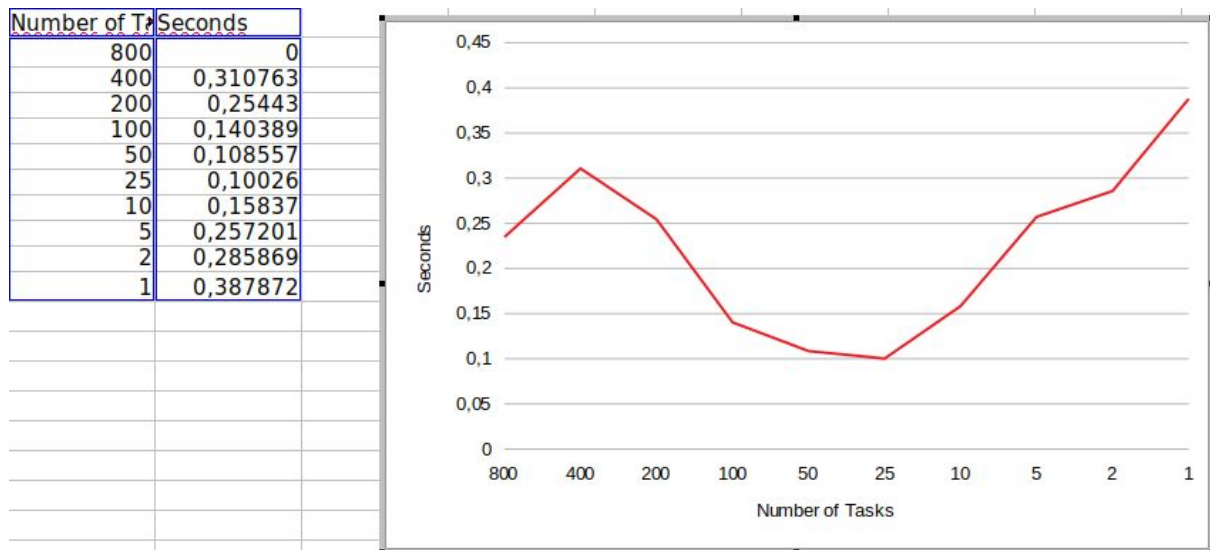


Figure 2.21 Strong scalability plots for version 5

In this version, the speedup is very high and it almost reach the perfect speedup. It is explained due to a less overhead between tasks and it gains a lot of execution time.



*Figure 2.22 Num Tasks - Seconds graphics of fifth version*

This plot is the relation between the number of tasks and the execution time, it makes a curve reaching the minimum in 25 number of tasks and the maximum with the 1 task.

## 4.3 Row decomposition in OpenMP

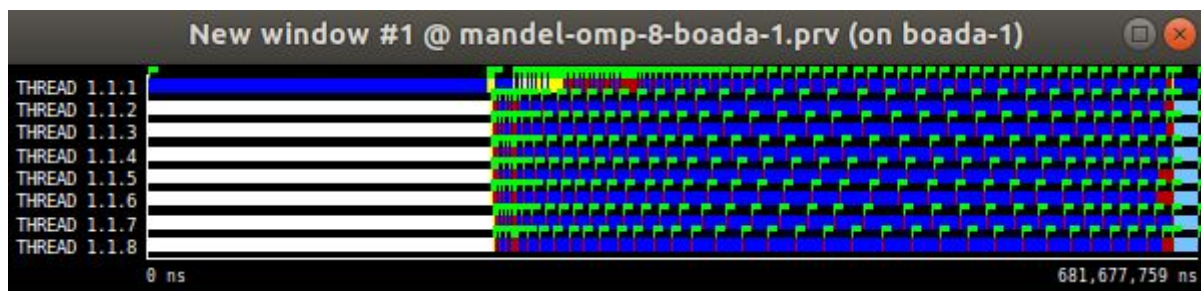
At this point we are questioned about row decomposition strategy, which consists of, as previously said, creating a task every time we visit a new row.

```
/* Calculate points and save/display */
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp task firstprivate(row) private(col)
    for (int col = 0; col < width; ++col) {
        .
    }
    complex z, c;
```

*Figure 3.1 Row decomposition source code*

The code is parallelized with the parallel and single directives before the beginning of the outer loop, then inside the loop a task is created.

Now, we will observe how this code works watching the tasks being created and synchronised between them.



*3.2 Figure 2.2 Timeline of first version*

As we can see all the threads are executing the code. The thread 1.1.1 forks the tasks and the others are synchronised to execute those tasks



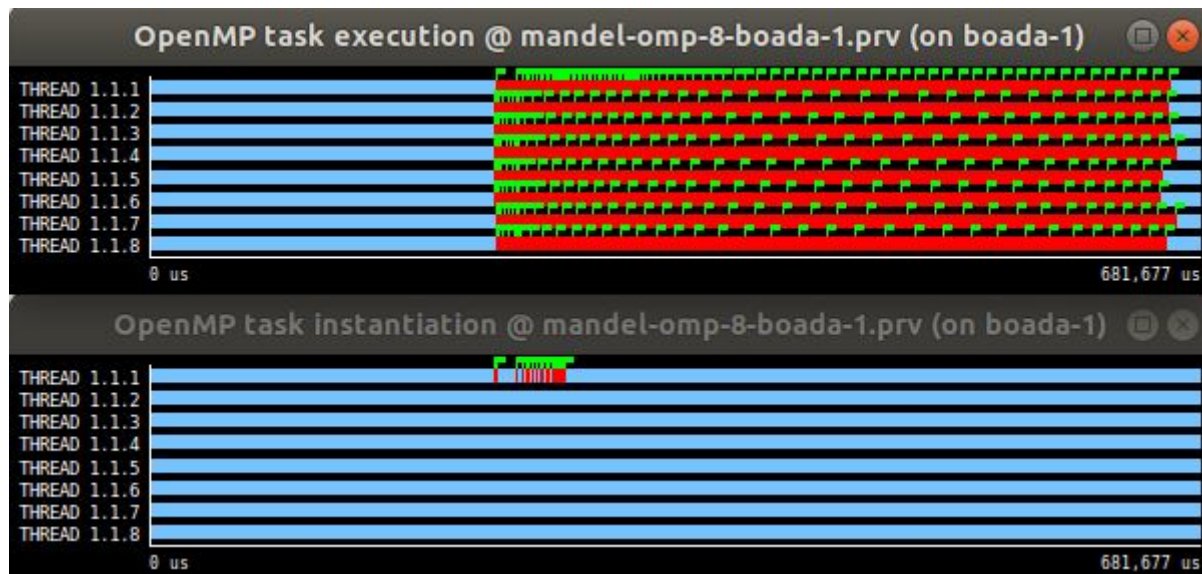


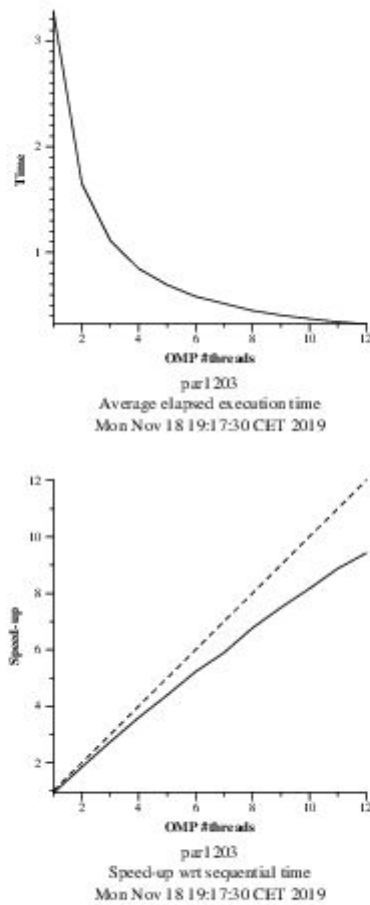
Figure 3.3 Tasks timelines

The workload among threads in the task execution is balanced. The activity in task instantiation is that little because the granularity is not as fine as the point decomposition strategy.

Task profile (execution and instantiation) @ mandel-omp-8-boada-1.prv (on boada-1)		
	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	315	800
THREAD 1.1.2	69	-
THREAD 1.1.3	79	-
THREAD 1.1.4	72	-
THREAD 1.1.5	69	-
THREAD 1.1.6	62	-
THREAD 1.1.7	74	-
THREAD 1.1.8	60	-
Total	800	800
Average	100	800
Maximum	315	800
Minimum	60	800
StDev	81.46	0
Avg/Max	0.32	1

Figure 3.4 Table of row decomposition

We can see in this table that the amount of tasks created is very little and the thread 1.1.1 covers all the creation overhead.



*Figure 3.5 Strong scalability plots.*

We can observe that as the number of threads is increasing the improvement is less than with the early increases. However, the execution performance is quite good and the speed-up line is similar to the ideal one



## 4.4 Conclusions

To conclude, we can determine that, taking into account the speedup plots, versions 1 to 4 of the point decomposition strategy are quite bad in terms of performance compared to the ideal speedup relationship due to the excessive overhead that the threads suffer and the time that there is no work being done.

However, in each of the four versions there are significant changes in both the declaration of the parallel region and in the creation of tasks, accomplishing little improvement that does not reach a performance that can be compared to the ideal one.

On the other hand, the latest version of the point decomposition and the row decomposition are much better and have a more optimal performance, since its plot is much closer to the ideal line. What makes us understand that the fact of manually managing the grainsize and putting the creation of tasks in the row's loop is much better for the performance of this particular program.