ENTREGABLE PARAL·LELISME

SEGON ENTREGABLE

Username: par4105
24 d’Octubre de 2018
Tardor del 2018

Àlex Valls
Roger Guasch

Taula de continguts

Task decomposition and granularity analysis

Explain the different task decomposition strategies and granularities explored with Tareador for the Mandelbrot code, including in your deliverable the task dependence graphs obtained and clearly indicating the most important characteristics for them (use the small test case -w 8 for this analysis). Explain which section of the code is causing the serialization of all tasks in *mandeld-tar* and how this section of code should be protected when parallelizing with OpenMP. Reason when each strategy/granularity should be used.

In this part of the deliverable, we have explored two different task decomposition strategies: point and row decomposition. In the following pages you are going to see different graphs obtained through Tareador. Below every graph there is a little explanation to contextualize each figure.

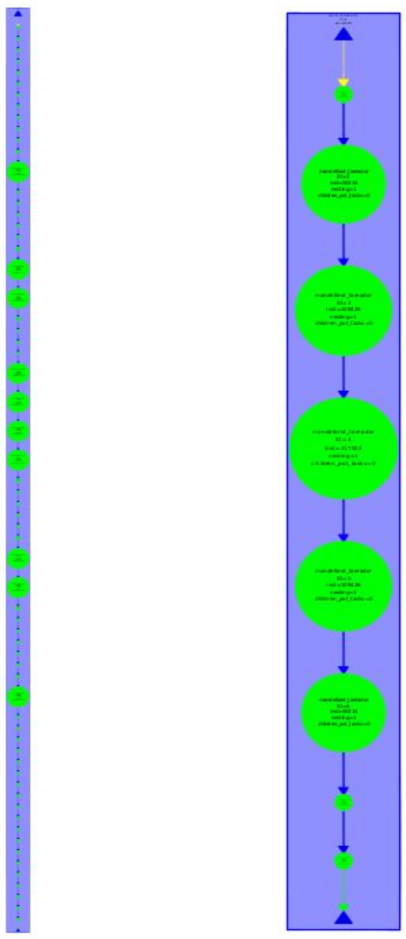


Figure 2.1 Mandelbrot display dependency graph, using point decomposition strategy, on the left of the figure. On the right we can see the dependency graph of the Mandelbrot display with row decomposition strategy.

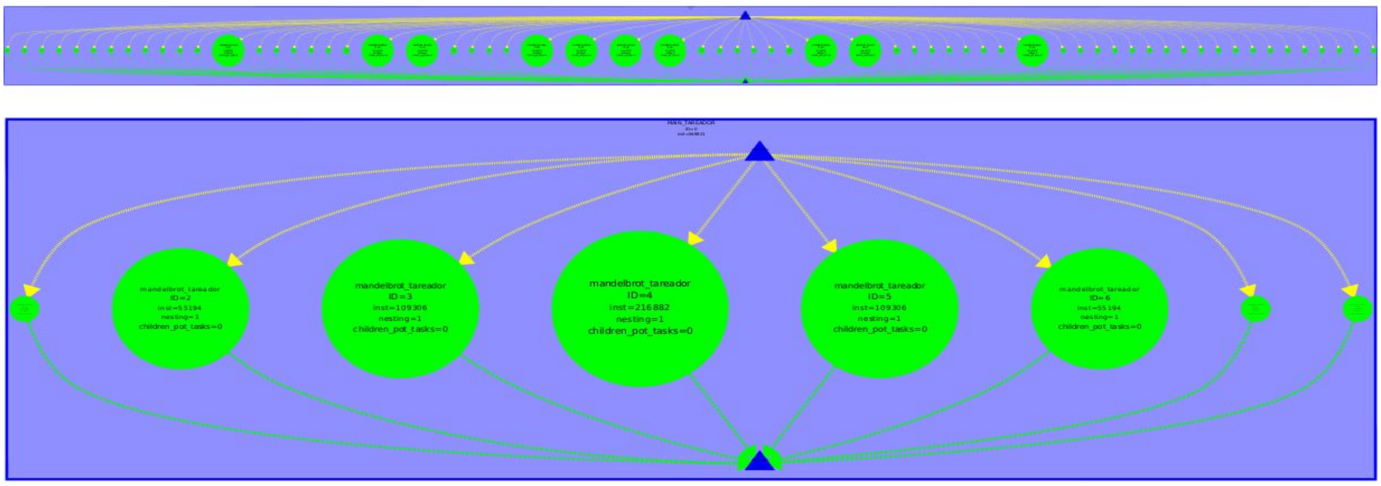


Figure 2.2 Mandelbrot dependency graph, using point decomposition strategy, on the top. At the bottom we can see the dependency graph of the Mandelbrot program with row decomposition strategy.

Looking carefully we can spot the difference between Mandelbrot and Mandelbrot display. Clearly we can state that Mandelbrot display isn't parallelizable, even though we tried with point and row decomposition. All tasks depends of the previous one because anyone can start until the previous one has finished its execution.
Comparing two different graphs from Figure 2.1 we can say that applying point decomposition we are adding more granularity to the problem, creating more tasks that with even finer size.

Analyzing now the Figure 2.2 it is not difficult to affirm that Mandelbrot program is parallelizable, indistinctly of which parallelization strategy we choose. The results of the granularity are not really different from Figure 2.1. In this case, point decomposition, adds more granularity to the tasks that are created, whereas row strategy creates less tasks but with bigger size. This will cause that, every task, takes longer to finish its execution, when using row strategy.

Another thing worth mentioning is that there will be an imbalance of the execution load, due to the different in sizes of the tasks. While one are going to take more time to be executed, the others, are going to need less time, causing this imbalance.

Before finishing the answer, we provide, in the Figure 2.3, the code of Mandelbrot. To avoid using more space than necessary we only have copied the code that is causing the serialization of all tasks in mandeld-tar.

```
/* Scale color and display point  */
/* In this part a variable named color is being created. We execute an store operation to initialize the variable */
        long color = (long) ((k-1) * scale_color) + min_color;
        if (setup_return == EXIT_SUCCESS)  {
          /* We access to the value of color to determine which one is needed to paint a concrete pixel */
          /* In conclusion: color variable creates dependence */
 /* Solution: Make sure any process accesses the value of the variable during the execution of this function */
```

```
                #pragma omp critical
                {
                    XSetForeground (display, gc, color);
                    XDrawPoint (display, win, gc, col, row);
                }
        }
```
Figure 2.3 Mandelbrot piece of code that is causing the serialization.

The variable that is causing the serialization is color. As we can see, every time we need to paint a pixel, an access to the variable color is needed. If we want to parallelize this code we need to make sure no more than one process access to this variable at the same time.

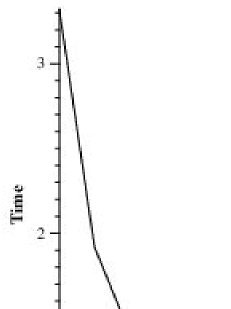
Point decomposition in OpenMP

For the Point strategy implemented in OpenMP, describe and reason about how the performance has evolved for the three task versions of the code that you have evaluated, using the speed–up plots obtained and Paraver captures. After that, explain the influence of the granularity control available in the taskloop construct, showing how the execution behaves when setting the number of tasks or iterations per task to 800, 400, 200, 100, 50, 25, 10, 5, 2 and 1, for example. Include the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000), again including Paraver captures to help you in the reasoning. 9 Optional: taskgroup vs. taskwait: Include the parallel version that makes use of the taskgroup construct. Explain the two fundamental differences with taskwait and if they are relevant in this code. Also include the strong scalability analysis and use tracing to show the differences with taskwait.

For gathering all the data needed for this report we followed the steps described below. After the study some modifications of the code were needed, followed by the compilation of each version in order to obtain different binaries. For the **first version**, the code can be found in the Figure 2.4.

```
for (row = 0; row < height; ++row) {
  #pragma omp parallel
  #pragma omp single
    for (col = 0; col < width; ++col) {
      #pragma omp task firstprivate(col) {
      ...
      do {
        temp = z.real*z.real - z.imag*z.imag + c.real;
        z.imag = 2*z.real*z.imag + c.imag; z.real = temp;
        lengthsq = z.real*z.real + z.imag*z.imag;
        ++k;
      } while (lengthsq < (N*N) && k < maxiter);
      output[row][
```

Figure 2.4 Code modified for the first version of the analyzed program.



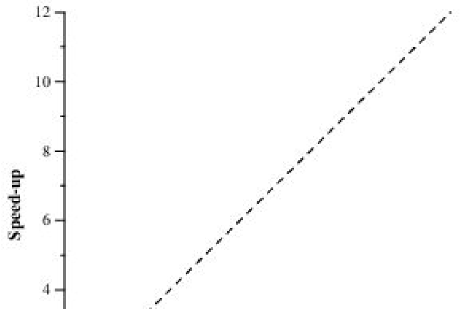


Figure 2.5 Execution and speed up of the first version of the program, using point granularity.

The execution and speed-up time of the first version can be seen in Figure 2.5. As we can appreciate, the execution time is reduced as long as we add new CPUSs. The problem is that, if we analyze the Speed-up plot, we will see that this is not the best paralel version we can obtain considering that the line is far below from the linear speed-up shown as a discontinuous line.

We also used the Tareador in order to obtain the task execution and task instantiation plots. The results can be seen in Figure 2.6. We can affirm that exists a considerable amount of code that is being executed sequentially, either at the beginning as well as the end. This reduces considerably the gain we can obtain as well as it acts as a barrier for our parallel version. The amount of control dependencies is considerable while the parallel part is being executed. There are not clear variable dependencies in this version, as we can assume with the results of the Tareador program.

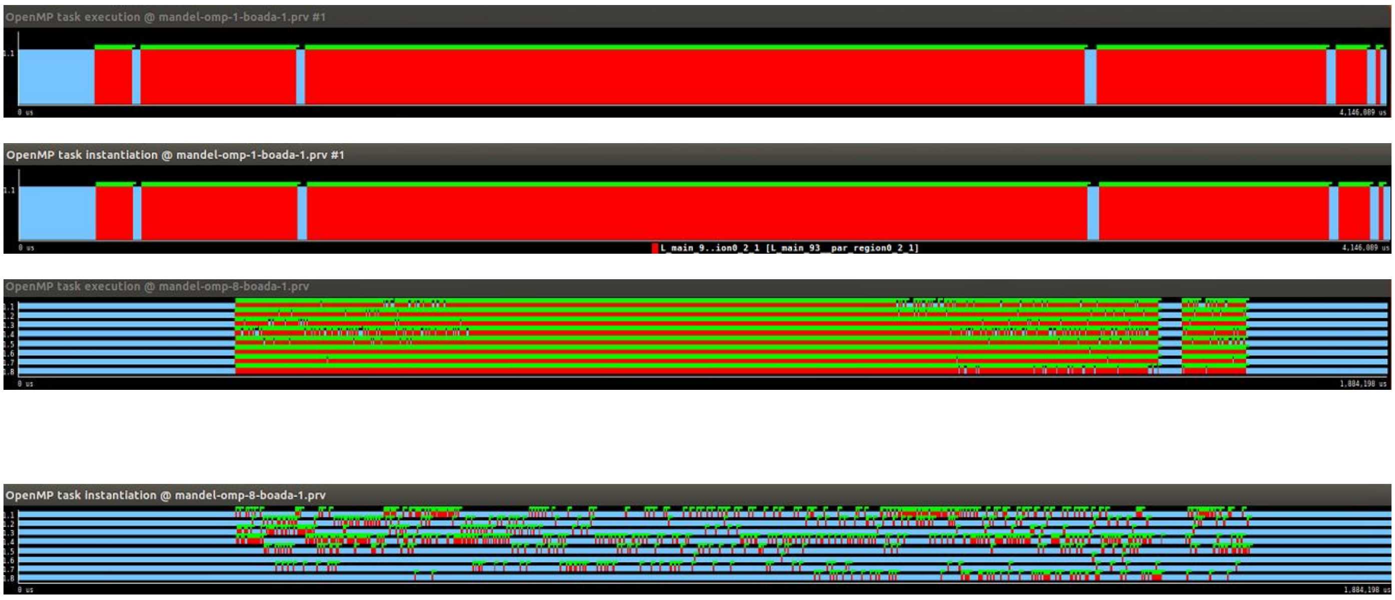
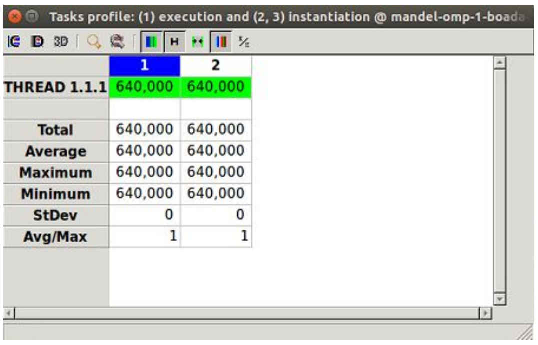
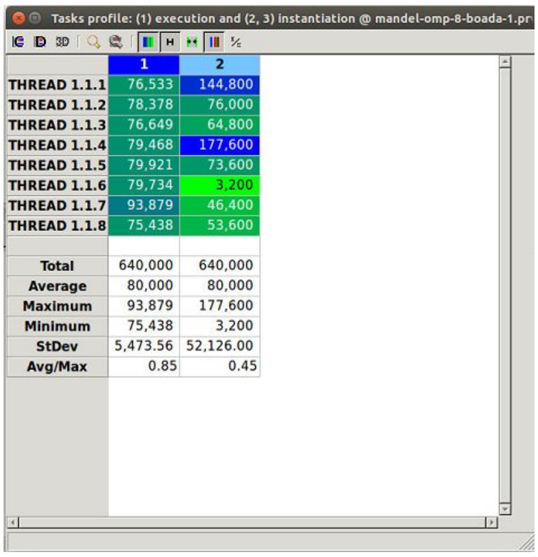

OpenMP task execution @ mandel-omp-1-boada-1.prv #1
0 us
4,146,089 us
OpenMP task instantiation @ mandel-omp-1-boada-1.prv #1
L_main_9..ion0_2_1 [L_main_93__par_region0_2_1]
4,146,089 us
OpenMP task execution @ mandel-omp-8-boada-1.prv
1,884,198 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv
1,884,198 us

Figure 2.6 Tareador task execution and task instantiation results for the first version of the program studied with 1 and 8 CPUs.

Although which point version we analyze with tareador, the results obtained with the sequential version will be the same. The tareador task execution and instantiation of the sequential program will be omitted in the following versions.


Tasks profile: (1) execution and (2, 3) instantiation @ mandel-omp-1-boad

	1	2
THREAD 1.1.1	640,000	640,000
Total	640,000	640,000
Average	640,000	640,000
Maximum	640,000	640,000
Minimum	640,000	640,000
StDev	0	0
Avg/Max	1	1


Tasks profile: (1) execution and (2, 3) instantiation @ mandel-omp-8-boada-1.pr

	1	2
THREAD 1.1.1	76,533	144,800
THREAD 1.1.2	78,378	76,000
THREAD 1.1.3	76,649	64,800
THREAD 1.1.4	79,468	177,600
THREAD 1.1.5	79,921	73,600
THREAD 1.1.6	79,734	3,200
THREAD 1.1.7	93,879	46,400
THREAD 1.1.8	75,438	53,600
Total	640,000	640,000
Average	80,000	80,000
Maximum	93,879	177,600
Minimum	75,438	3,200
StDev	5,473.56	52,126.00
Avg/Max	0.85	0.45

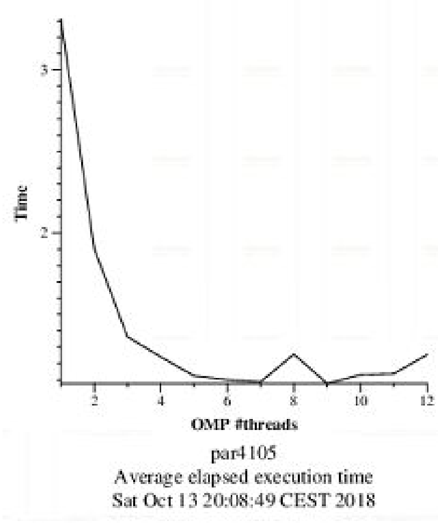
Figure 2.7 Task profiles for the first version of the program studied with 1 and 8 CPUs.

In the **second version** of the program we have explored more point parallelization strategies, using *taskwait* instruction. What *taskwait* does is that it specifies a wait on the completion of child tasks of the current task as it stated in the OpenMP reference manual. What it means is that every child waits in that point until every tasks have finished it's execution. In this version we are also using the *firstprivate* clause, that specifies that each thread should have its own instance of a variable and should be initialized with the value of the variable, because it exists before the parallel construct. In other words: the variable inside the clause will be initialized with the value that it contained prior to the parallel region. The code used for this particular version can be read in the Figure 2.8, whereas the execution time and speed up plots can be seen in Figure 2.9.

```
#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 taskwait // waiting point for all child tasks
}
```

Figure 2.8 Code modified for the second version of the analyzed program.

Going more in detail of the code described above we can conclude that there will be one *taskwait* for every column of the matrix. Also the *row* and *col* variables will have the value that they had prior to the parallel region.



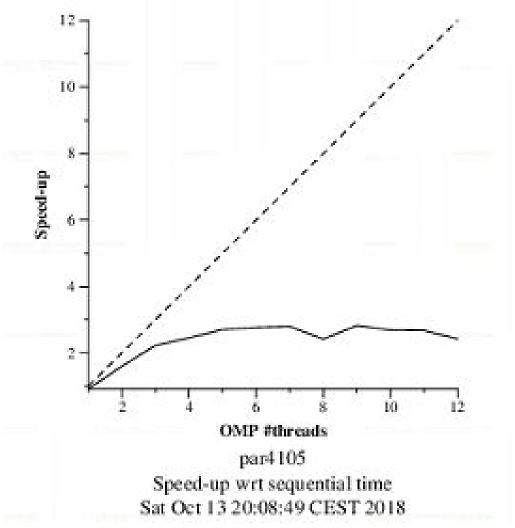
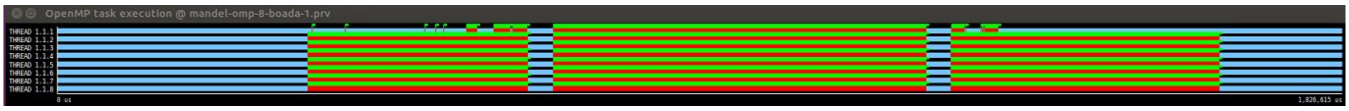


Figure 2.9 Execution and speed up of the second version of the program, using point granularity with taskwait and firstprivate.

Comparing the plots obtained in this version with the plots of the first one -Figure 2.5- we can see that execution time plot decreases until or near 0 y-axis faster than the first version. Although that, the speed up plot is not substantially improved respect its predecessor version .



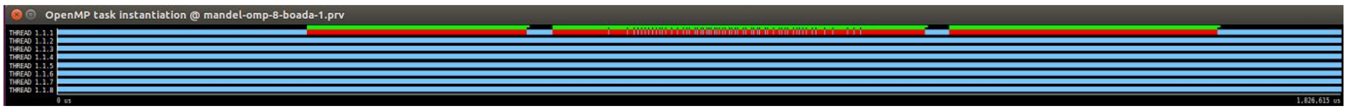
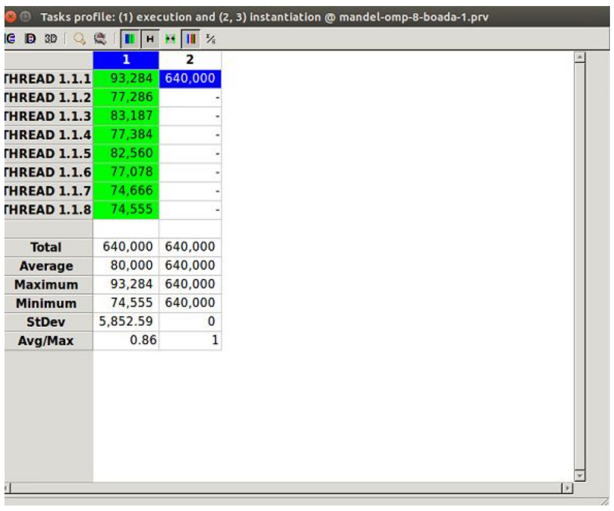


Figure 2.10 Tareador task execution and task instantiation results for the second version of the program studied with 8 CPUs.

We can see that there is a considerable difference between the results of this paraver and the ones obtained while executing the program with one CPU but there is no difference between the first version and the second parallel version if we compare both execution times.

Comparing now the task instantiation we can appreciate that this version is a little bit faster than the previous one, shown in Figure 2.6, although this second version has more instantiations.


Tasks profile: (1) execution and (2, 3) instantiation @ mandel-omp-8-boada-1.prv

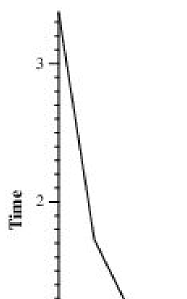
	1	2
THREAD 1.1.1	93,284	640,000
THREAD 1.1.2	77,286	-
THREAD 1.1.3	83,187	-
THREAD 1.1.4	77,384	-
THREAD 1.1.5	82,560	-
THREAD 1.1.6	77,078	-
THREAD 1.1.7	74,666	-
THREAD 1.1.8	74,555	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	93,284	640,000
Minimum	74,555	640,000
StDev	5,852.59	0
Avg/Max	0.86	1

Figure 2.11 Task profile for the first second version of the program studied with 8 CPUs.

For the **third version** of this program, for this specific part of the deliverable, we have explored the use of the taskloop constructs. This construct specifies that the iterations of one or more associated loops will be executed in parallel, using OpenMP tasks. We have to consider that the iterations are going to be distributed across tasks created by the construct and scheduled to be executed. In other words: every iteration will be a possible parallel task and will wait inside a pool of parallel instructions until one CPU is able to execute it. The process that CPUs follow in order to choose one task inside the task pool is called **scheduling**.

```
#pragma omp parallel
#pragma omp single
  for (row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) num_tasks(64) // grainsize(width/64)
    for (col = 0; col < width; ++col) {
      ...
    }
}
```
Figure 2.12 Code modified for the third version of the analyzed program.

We have to consider, reading the code in the Figure 2.12, that with taskloop construct one can use other instructions such as firstprivate, described in the previous version, or stablish the number of tasks and their granularity.



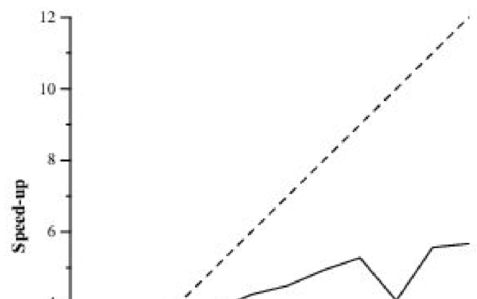
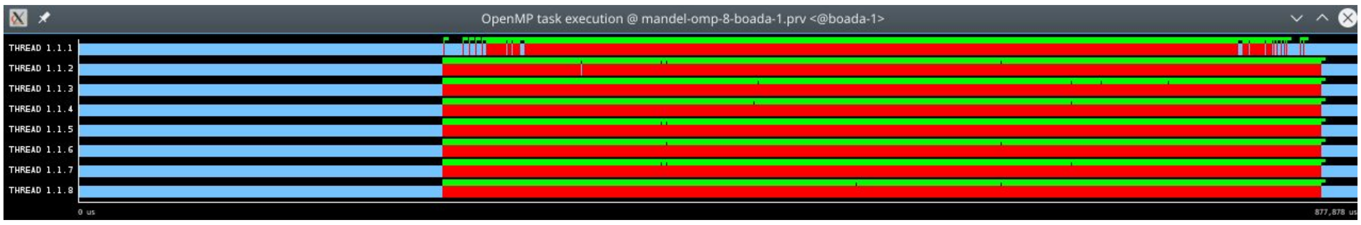


Figure 2.13 Execution and speed up of the third version of the program, using point granularity with taskloop.

Comparing the execution time plot with the second version one can state that the time line descends slower than its predecessor. Although that it looks like in previous plots, using 12 or more threads will cause an increment of the execution time whereas the plot for this third version clearly shows that adding more threads will reduce even more the average elapsed execution time. Seeing the speed up plot there is no doubt that this version, compared with the previous one, is better because the parallelization plot is closer to the linear speed up than previous versions.



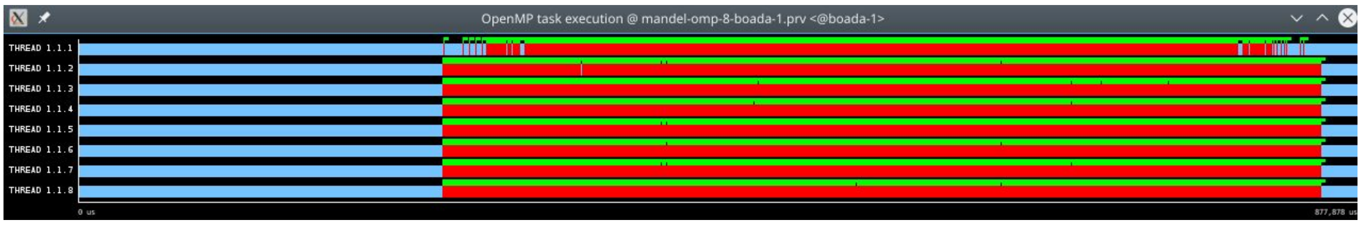
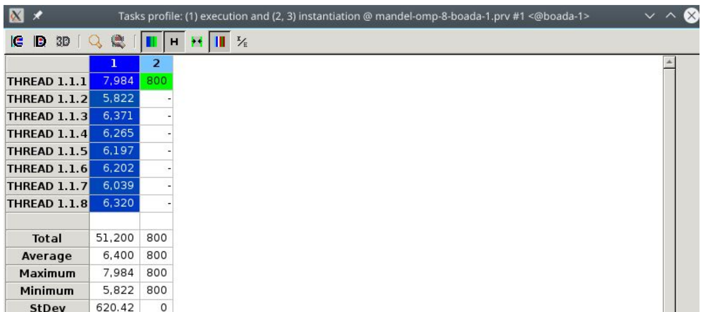


Figure 2.14 Tareador task execution and task instantiation results for the third version of the program studied with 8 CPUs.

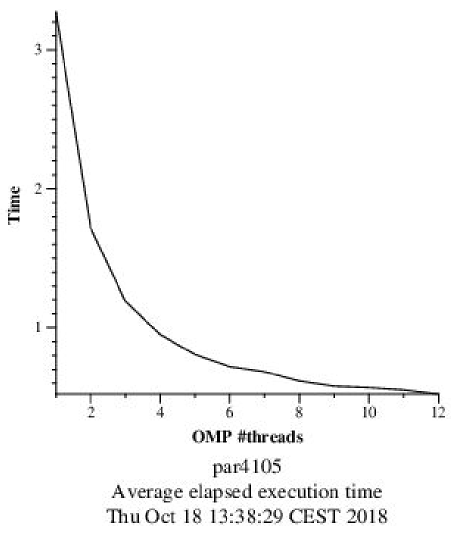
Now we can clearly see that there is a 950.000ns execution time improvement between this version and the previous ones. In the other hand the task instantiation time is being multiplied almost by three in this version, if we do the same comparison as described before.

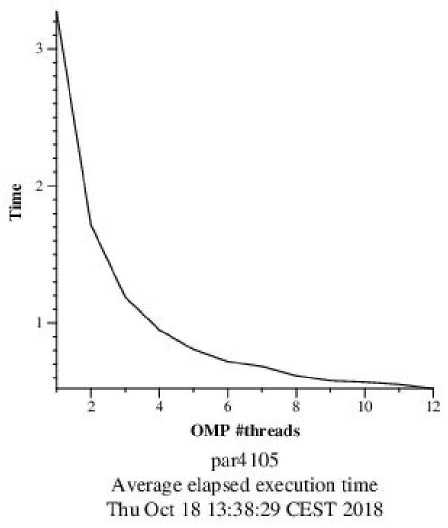

Tasks profile: (1) execution and (2, 3) instantiation @ mandel-omp-8-boada-1.prv #1 <@boada-1>

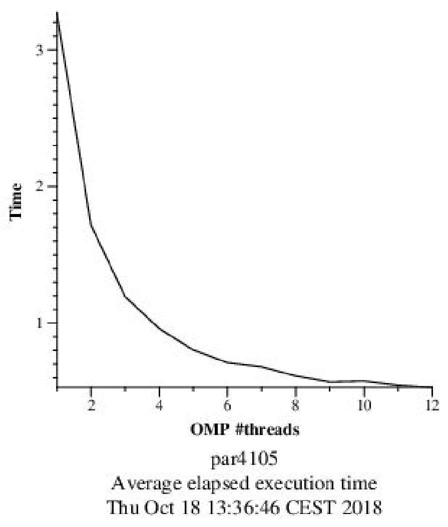
	1	2
THREAD 1.1.1	7,984	800
THREAD 1.1.2	5,822	-
THREAD 1.1.3	6,371	-
THREAD 1.1.4	6,265	-
THREAD 1.1.5	6,197	-
THREAD 1.1.6	6,202	-
THREAD 1.1.7	6,039	-
THREAD 1.1.8	6,320	-
Total	51,200	800
Average	6,400	800
Maximum	7,984	800
Minimum	5,822	800
StDev	620.42	0

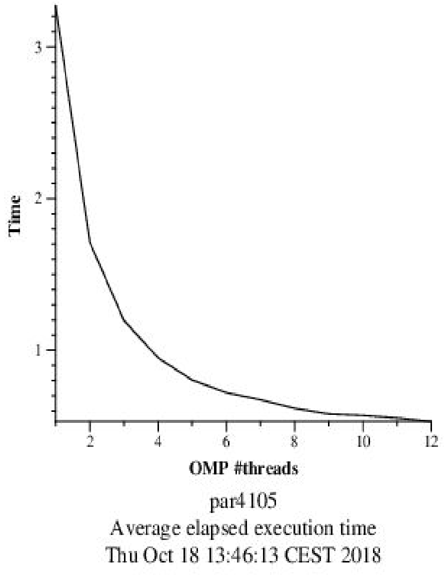
Figure 2.15 Task profile for the first third version of the program studied with 8 CPUs.

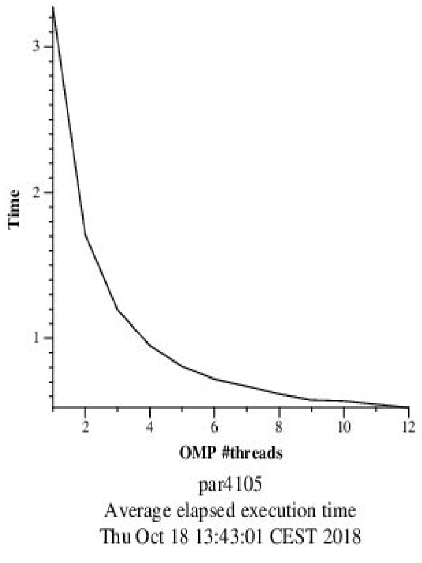
Before finishing this part we have studied how the average execution time and speed up plots change while increasing or decreasing *grainsize*. The results can be seen in Figure 2.16 for the average execution time and in Figure 2.17 for the speed up.

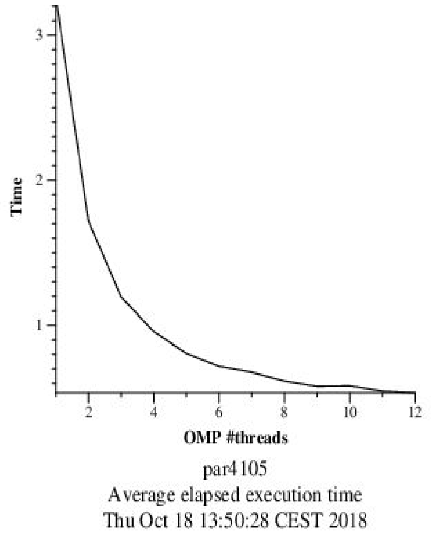


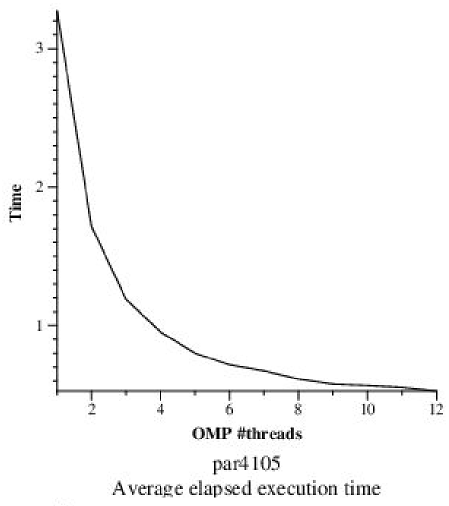


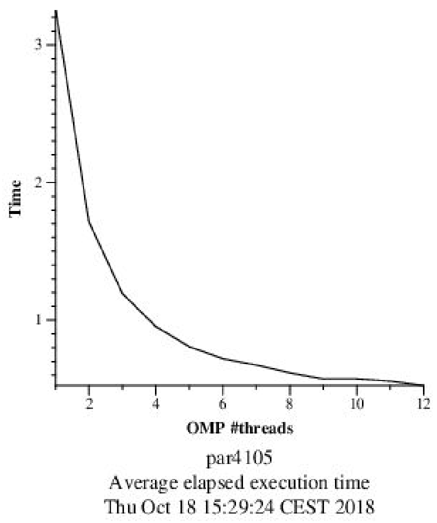


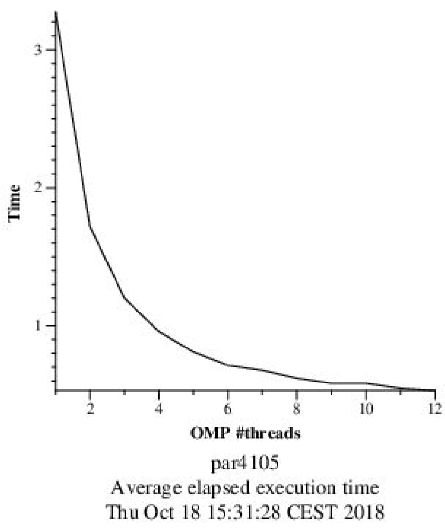












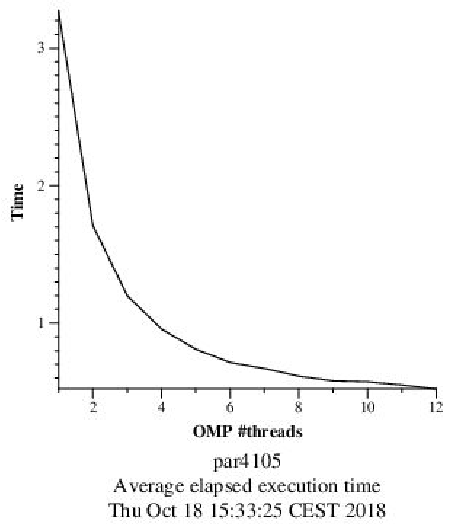
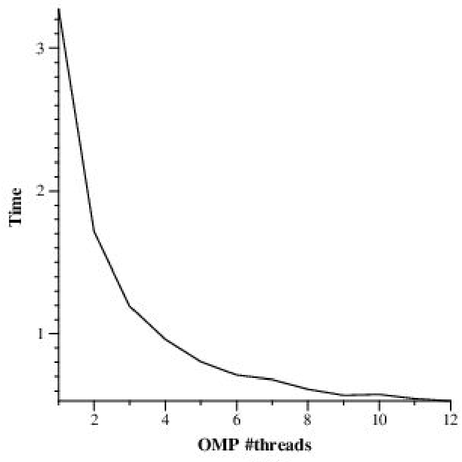
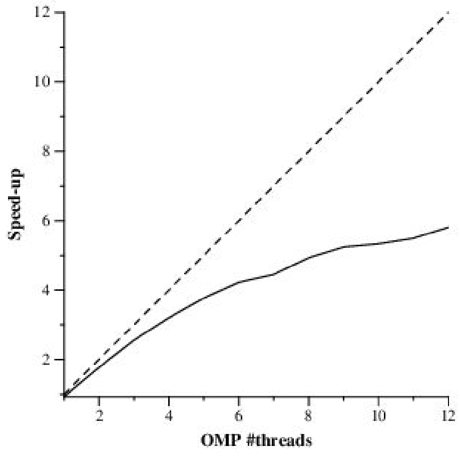


Figure 2.16 Average execution time while modifying grainsize value. From left to right the plot is obtained with 800, 400, 200, 100, 50, 25, 10, 5, 2 and 1 grainsize.

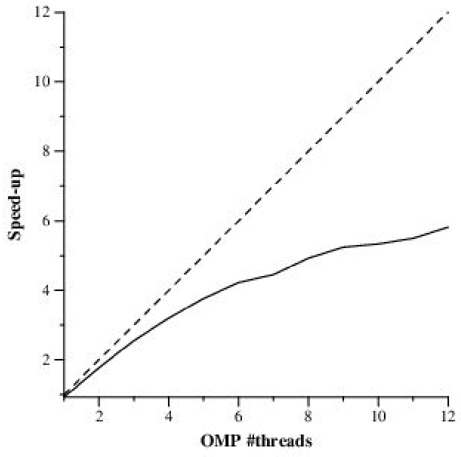
It’s really difficult to see any appreciable difference between the different plots in Figure 2.12. We can conclude that increasing or decreasing grainsize does not affect the average elapsed time.



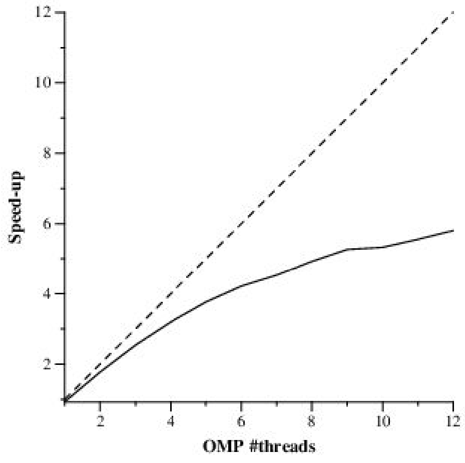
par4105
Average elapsed execution time
Thu Oct 18 13:36:46 CEST 2018



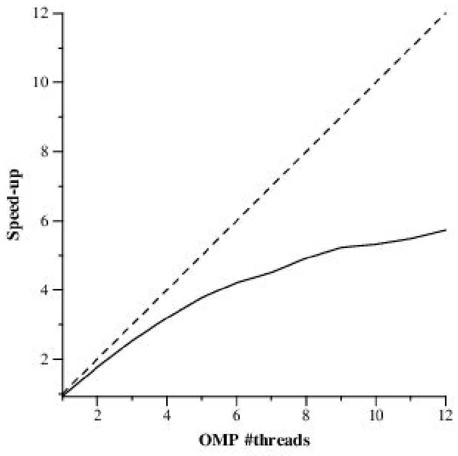
par4105
Speed-up wrt sequential time
Thu Oct 18 13:38:29 CEST 2018



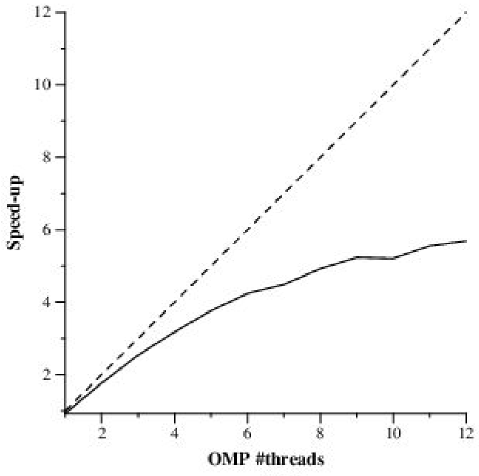
par4105
Speed-up wrt sequential time
Thu Oct 18 13:38:29 CEST 2018



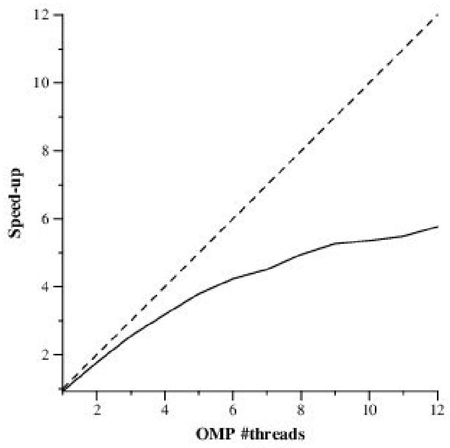
par4105
Speed-up wrt sequential time
Thu Oct 18 13:43:01 CEST 2018



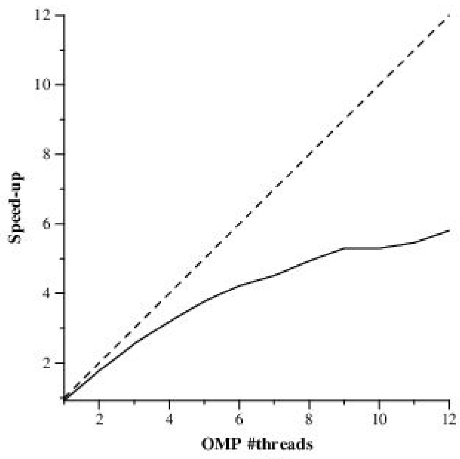
par4105
Speed-up wrt sequential time
Thu Oct 18 13:46:13 CEST 2018



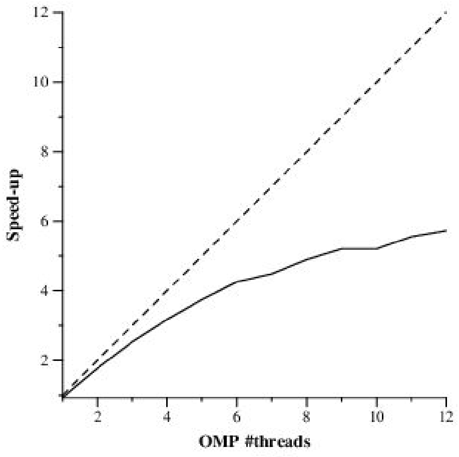
par4105
Speed-up wrt sequential time
Thu Oct 18 13:50:28 CEST 2018



par4105
Speed-up wrt sequential time
Thu Oct 18 13:53:27 CEST 2018



par4105
Speed-up wrt sequential time
Thu Oct 18 15:29:24 CEST 2018



par4105
Speed-up wrt sequential time
Thu Oct 18 15:31:28 CEST 2018

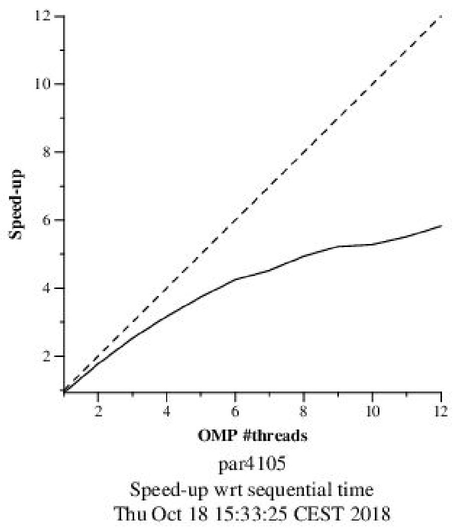


Figure 2.17 Average speed up time while modifying grainsize value. From left to right the plot is obtained with 800, 400, 200, 100, 50, 25, 10, 5, 2 and 1 grainsize.

While doing this experiment we couldn't spot any difference between the speed up plot between selecting 800 grainsize and 1 grainsize.

Optional: taskgroup vs. taskwait:

Include the parallel version that makes use of the taskgroup construct. Explain the two fundamental differences with taskwait and if they are relevant in this code. Also include the strong scalability analysis and use tracing to show the differences with taskwait.

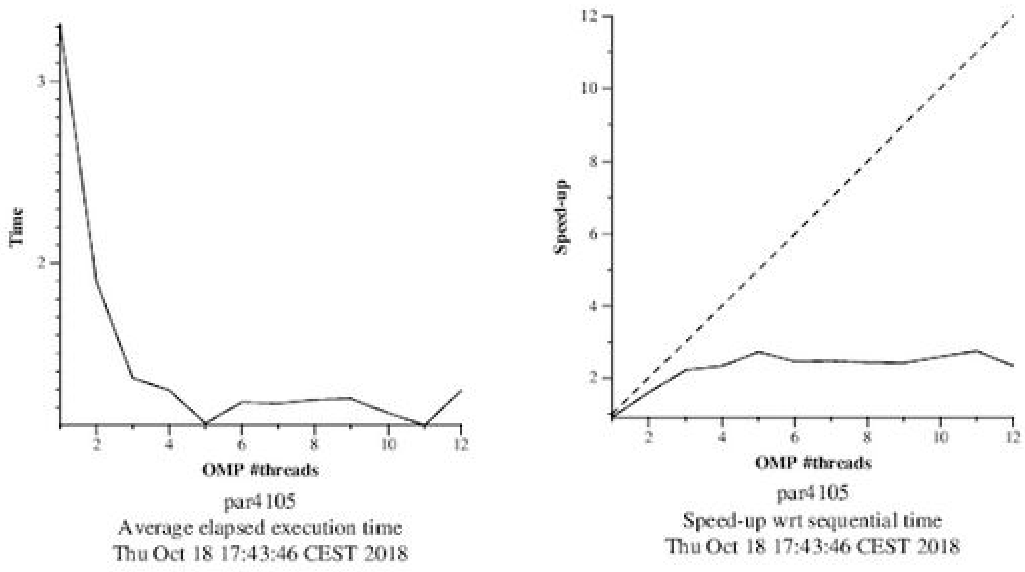


Figure 2.18 Execution time and Speed-up plots with taskgroup

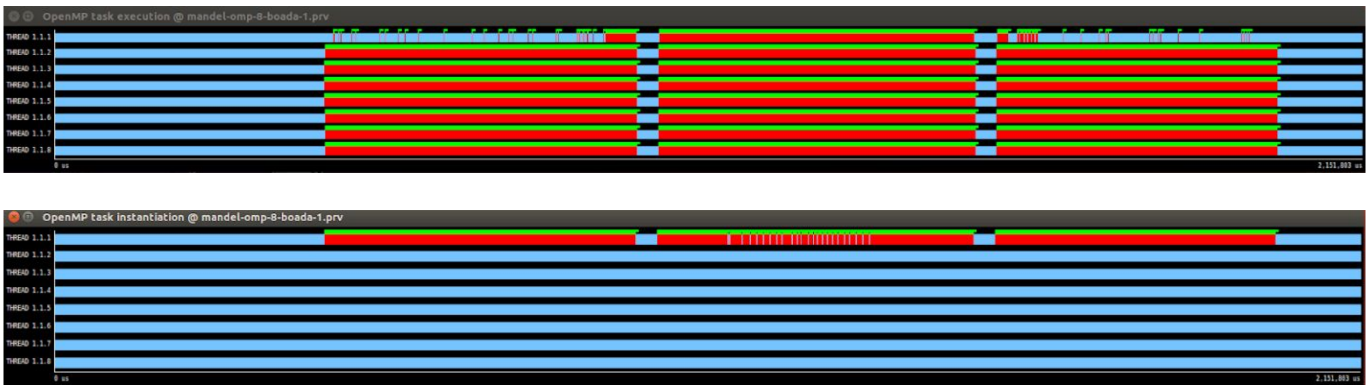
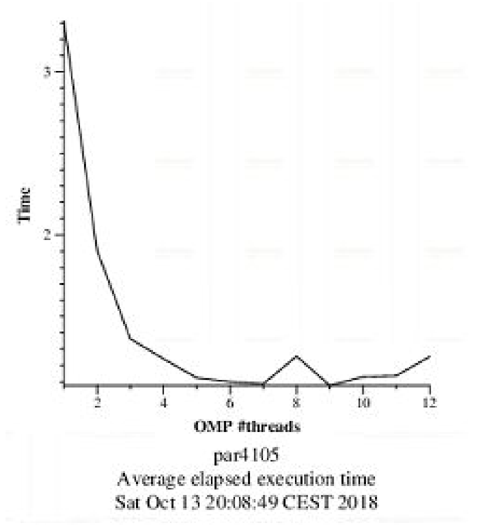


Figure 2.19 Tareador task execution and task instantiation results for taskgroup version of the program studied with 8 CPUs.

In this part we can see the analysis of the parallel version using taskgroup. The main difference between *taskgroup* and *taskwait* is that, in the first one, the current task is suspended until all child and descendant tasks, that have been generated, finished the execution. In *taskgroup*, the task only suspends its execution until all child tasks have finished their execution.



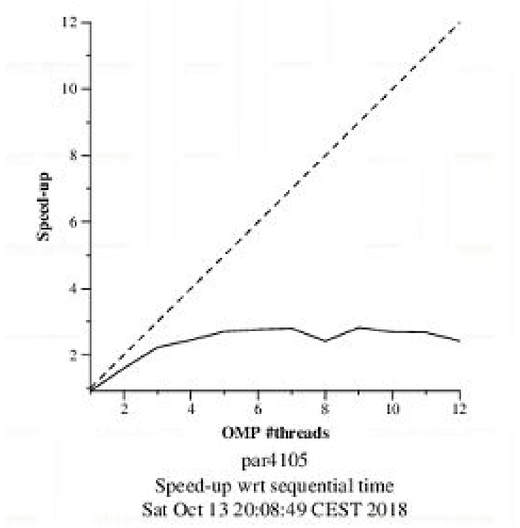


Figure 2.20 Execution time and Speed-up plots with taskwait

Now we can compare the *taskwait* and the *taskgroup* plots. The taskwait program has a small improvement on its execution time, because it has in almost every thread, a very low value. In contrast, the *taskgroup* speedup, is better than the *taskwait* speedup plot, because it is more consistent over threads. The differences between the *taskwait* and *taskgroup* are almost undetectable, causing some difficulties in spotting the little difference in the final analysis.

Row decomposition in OpenMP

For the Row strategy implemented in OpenMP, describe the parallelization strategy that you have decided to implement. Reason about the performance that is obtained comparing with the results obtained for the best Point implementation, including the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000) and Paraver captures that help you to better explain the results that have been obtained.

After some discussion, for the row decomposition study, we implemented the *taskloop* strategy. This construct what it does is that it specifies that one or more loops will be executed in parallel. Each iteration is distributed across the tasks that have been created and scheduled to be executed. We also need to detail that the variable row has to be firstprivate, in order to save it's value and avoiding that every thread executes all the tasks. The code that we implemented it can be seen in Figure.

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop firstprivate(row) num_tasks(64) // grainsize(width/64)
for (int row = 0; row < height; ++row) {
  for (int col = 0; col < width; ++col) {
  {
    .
    .
    .
  }
}
```

Figure 2.21 Code modified for the row parallelization using taskgroup strategy.

Taking advantage of the fact that we can declare more than one OpenMP directive per line we also detail that we don't want to have more than 64 tasks with the num_tasks directive. Also grainsize is explicitly detailed with width divided by 64, controlling how many tasks are being assigned to every thread.

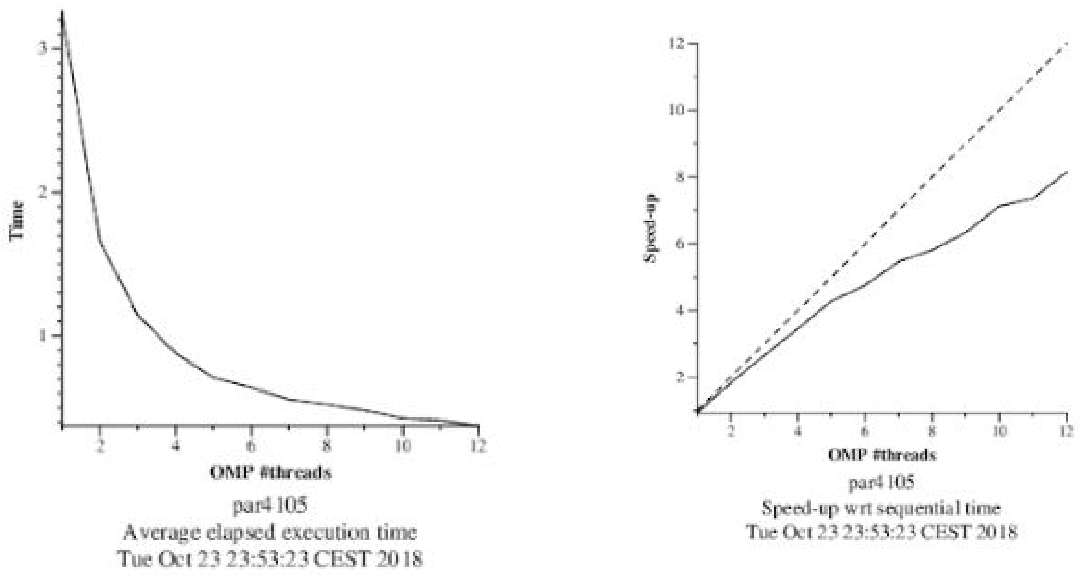


Figure 2.22 Execution time and Speedup plots of the taskgroup parallelization strategy.

The execution time plot shown above is similar to an negative exponential function. While the number of threads increases the execution time is reduced until reaching almost 0 seconds with 12 threads. The speedup plot is one of the best plots obtained in this laboratory assignment because, as we can see, the line is very close to the theoretical linear execution. In the end, the plot separates a little bit more but, although that, it preserves an almost linear form until 6 threads.

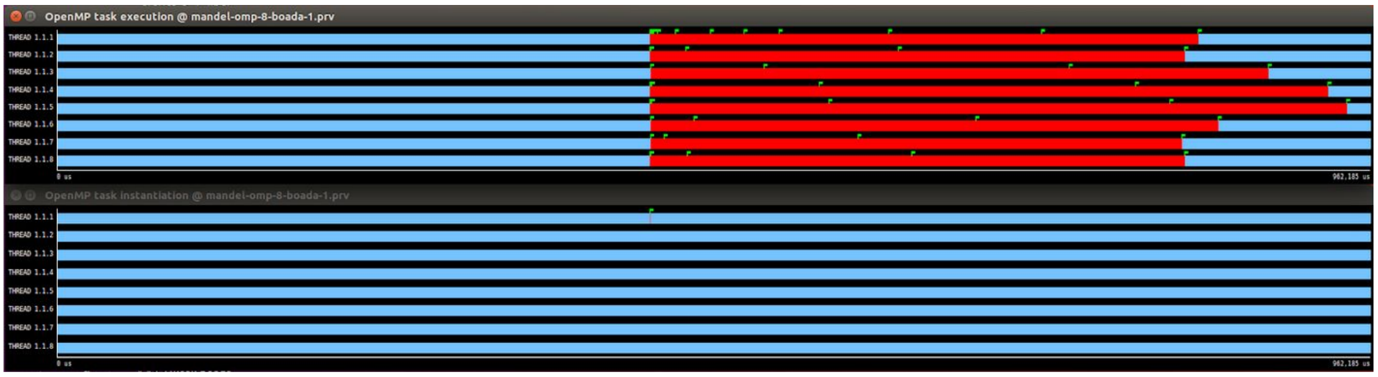


Figure 2.23 Execution time and Speedup plots of the taskgroup parallelization strategy.

The results obtained with tareador shows that the number of tasks instantiations are almost nonexistent. This parallelization is useful because it avoids unnecessary overheads while achieving impressives results as shown in the Figure (plot exec time).

Optional: for–based parallelization

If you decided to implement the for-based parallelization, Include the relevant portion of the OpenMP code commenting whatever necessary. Also include the execution time and speed–up plots that have been obtained for different loop schedules when using for example 8 threads (with -i 10000). Reason about the performance that is observed.

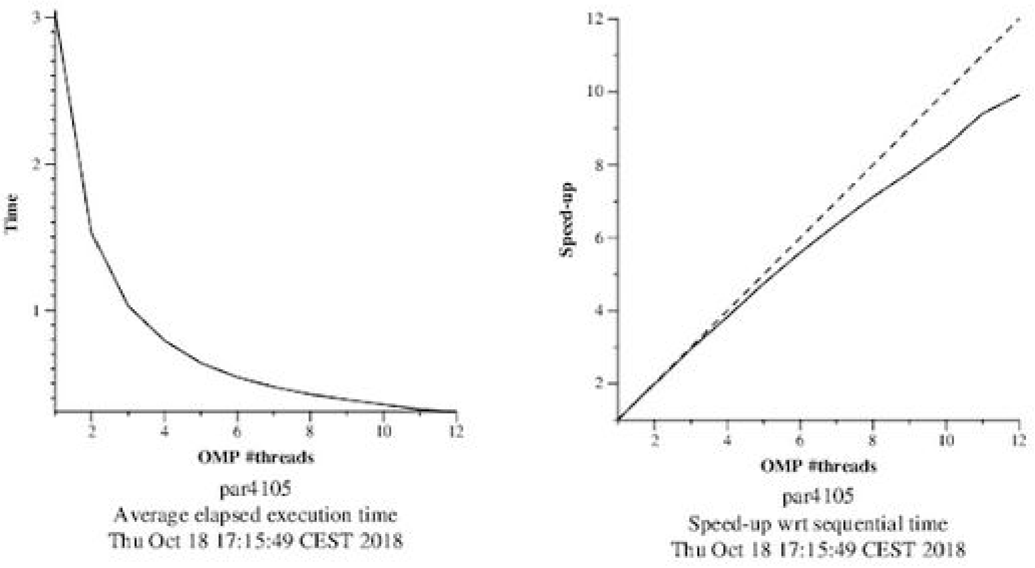


Figure 2.24 Execution time and Speedup plots in dynamic schedule

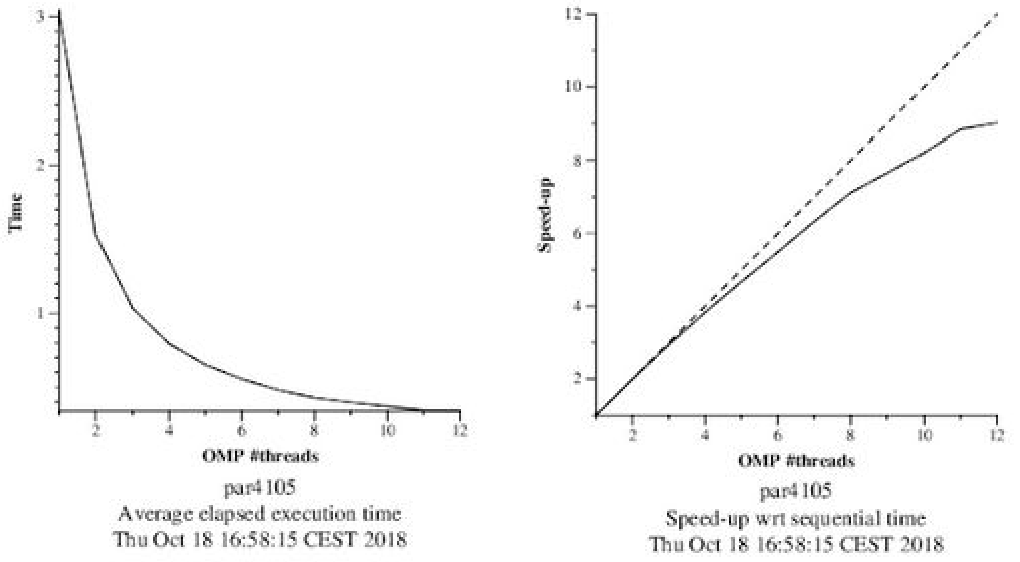


Figure 2.25 Execution time and Speedup plots in static schedule

```
#pragma omp parallel
#pragma omp for schedule(static,1)
for (int row = 0; row < height; ++row) {
  for (int col = 0; col < width; ++col) {
    {
      .
      .
      .
    }
  }
}
```

Figure 2.26 Fragment of code showing the static version of the program.

```
#pragma omp parallel
#pragma omp for schedule(dynamic,1)
for (int row = 0; row < height; ++row) {
  for (int col = 0; col < width; ++col) {
    {
      .
      .
      .
      }
    }
  }
```

Figure 2.27 Fragment of code showing the dynamic version of the program.

In this part, we have changed the parallel portion of the code as you can see in the images above. We have used the for-based parallelization for testing static and dynamic schedule. As we can see in the graphics, the execution time in both cases is almost identical. Otherwise, if we compare the speedup graphics we can see that, as the number of threads increase, the dynamic schedule is better than the static one. It occurs because the dynamic schedule assigns all the tasks among the threads available in runtime. In conclusion, we can say that it is preferable to choose the dynamic schedule although it doesn't exist a big difference between both.