Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

par4111
Adrià Cabeza, Xavier Lacasa
Departament d' Arquitectura de Computadors

April 10, 2019
2018 - 19 SPRING

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Contents

1 Introduction

This second laboratory assignment consists on a series of tests and modifications on a program called Mandelbrot. The program computes a particular set of points belonging to the complex domain,whose boundary generates a distinctive and recognisable two-dimensional fractal shape.

2 Task decomposition and granularity analysis

In this section we will explain the different task decomposition strategies and granularities explored using Tareador and the Mandolbrot program.

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.

2.1 Point granularity

Firstly, we will do a task at each single point (row,col) level. Please look at the Figure 1 to see the dependency graph obtained at that level of granularity. The code associated to the tasks defined at point granularity level is defined is as it follows:

```
/* Calculate points and save/display */
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
        tareador_start_task("point");
      // calculation of a single point
        tareador_end_task("point");
    }
}
```

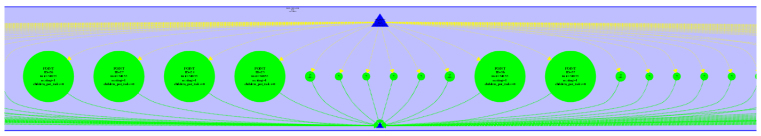


Figure 1: Part of the dependency graph for point decomposition

2.2 Row granularity

The second one consists on doing the tasks at the row level. Please look at the Figure 2 to see the dependency graph obtained at that level of granularity.

The code associated to the tasks defined at point granularity level is defined is as it follows:

```
/* Calculate points and save/display */
for (row = 0; row < height; ++row) {
   tareador_start_task("row");
    for (col = 0; col < width; ++col) {
      // calculation of a single point
    }
    tareador_end_task("row");
}
```

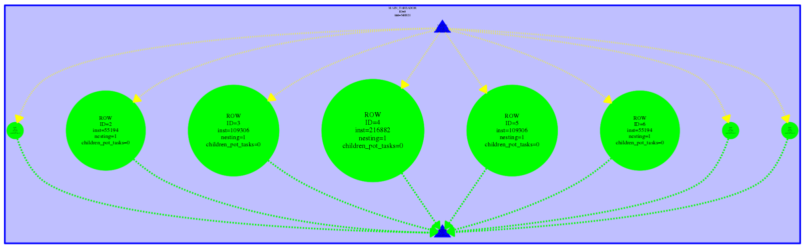


Figure 2: Dependency graph for row decomposition

If we compare each dependency graph we can observe that the load between different tasks is not equally distributed: the sizes of the tasks have very different values. For this reason we decided that to create tasks in the *static mode* is not the best option. We should use *dynamic mode* instead. Also we can observe that there is no shared data between tasks so there are no dependencies.

This inspection leads us to the question of which task granularity is the more appropriate to apply to implement a parallel version of the Mandelbrot code. We do think that depends on the system we are executing the program.

In boada, for example, we are not going to have a computer with thousands of processors, so the point granularity is a bit useless since we will never get any profit from it and the overhead is bigger. We believe that the best granularity is the row one.

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?

The previous analysis was made using the *mandel-tar* program which does not show the fractal on the screen. If we analyse the program that shows the image (*mandeld-tar*, we get the following dependency graph (Figure 3).

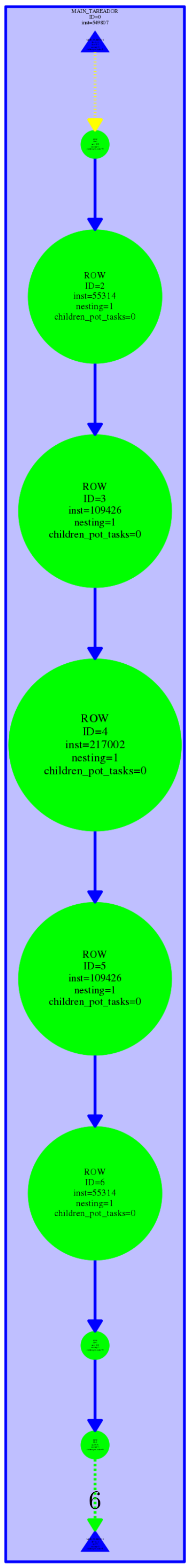


Figure 3: Dependency graph for *mandeld* program

It is easy to conclude that we have obtained a sequential version of the program with a lot of dependencies that do not allow to parallelize. The reason for this behaviour is the way the results are printed in the screen (we need to print the dots line by line so parallelism is useless here). So we can conclude that the window that is used to draw the Mandelbrot Set (X11) is the reason for the dependencies. In order to parallelize the code we should modify it as follows:

```
#pragma omp critical
{
   XSetForeground(display,gc,color);
   XDrawPoint(display,win,gc,col,row);
}
```

In this solution we have created a critical section to avoid the variable access that creates the conflict in order to avoid the possible data race and delete those dependencies. However, we also thought about some other options like calculating first the points and when you have all of them calculated paint them. But we decided to go with the critical zone option, since it is more efficient and implement it would take more effort and a lot of changes in the code.

3 *Point* decomposition in *OpenMP*

In this section we will analyse different parallelization schemes applied to the point-granularity and tasking model.

The way tasking works is a thread that generates a task pool and assigns, at execution time, the different tasks to available threads.

There are different tasking schemes: simple-tasking, tasking using the *taskwait* constructor and tasking using *taskloop*.

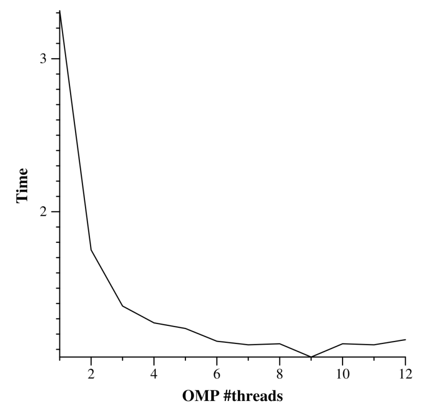
Describe and reason about how the performance has evolved for the three task versions of the code that you have evaluated.

3.1 Simple tasking

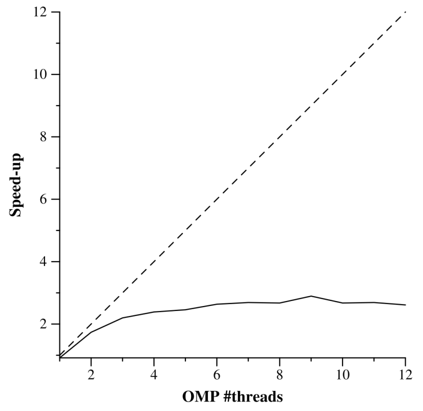
Firstly we have used the *simple-tasking* method. In this first version of the program the team of threads is created in each iteration of the row loop, by using the **parallel** construct; immediately after that, a single task generator is specified by using the **pragma omp single** distributor. Using this version we obtain the speed-up plot below (Figure 6). The code to parallelize using this method is the following:

```
#pragma omp parallel
#pragma omp single
for(row= 0; row<height; ++row){
    for (col = 0; col < width; ++col){
        #pragma omp task firstprivate(col)
        {
            ...
        }
    }
}
```

The private clause exists in order to avoid problems of data races, and the firstprivate is because it is necessary to have the old value.



par4113
Average elapsed execution time
Wed Apr 10 01:21:07 CEST 2019



par4113
Speed-up wrt sequential time
Wed Apr 10 01:21:07 CEST 2019

Figure 4: Simple-tasking plots

The traces look like these when examined with Paraver, with normal (top) and task (mid and bottom) configurations:

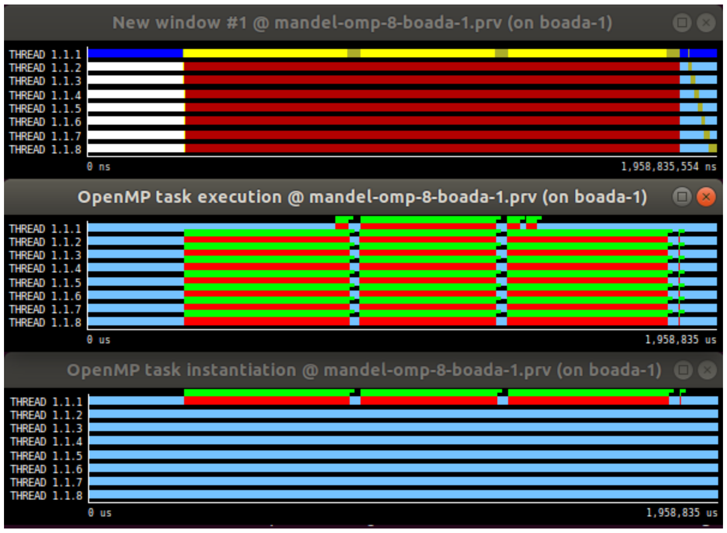

New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
1,958,835,554 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,958,835 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,958,835 us

Figure 5: Simple-tasking paraver traces

The top graphic indicates thread executions, the middle one shows task executions in green/red and the bottom one indicates task instantiation in green/red. It is very clear that only thread 1 creates tasks in the bottom one, and while doing so, all other threads execute them in the graphic in the middle.

Here we can have a closer look at the task execution traces:

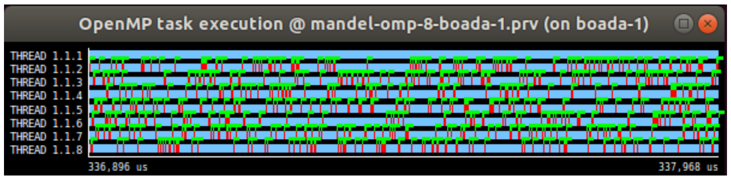
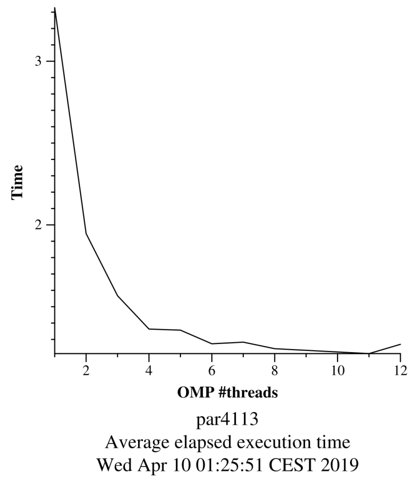


Figure 6: Simple-tasking task execution paraver traces

3.2 Taskwait tasking

Secondly, we have used *taskwait* method, which defines a point in the program to wait for the termination of all child tasks generated up to that point. In this version we obtained an speed-up plot (Figure 9) that is very similar to the simple-tasking one. The main difference between both options is that *taskwait* has a more defined positives growth rate. The code to parallelize using this method is the following:

```
#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
}
```



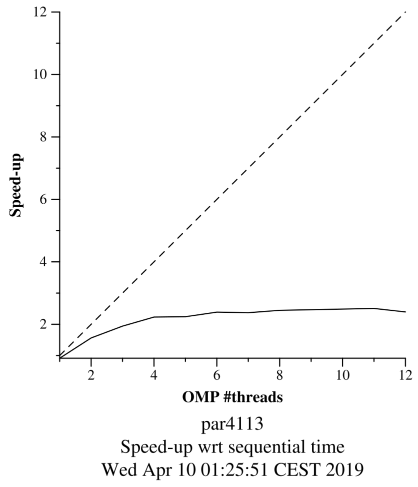


Figure 7: Taskwait method plots

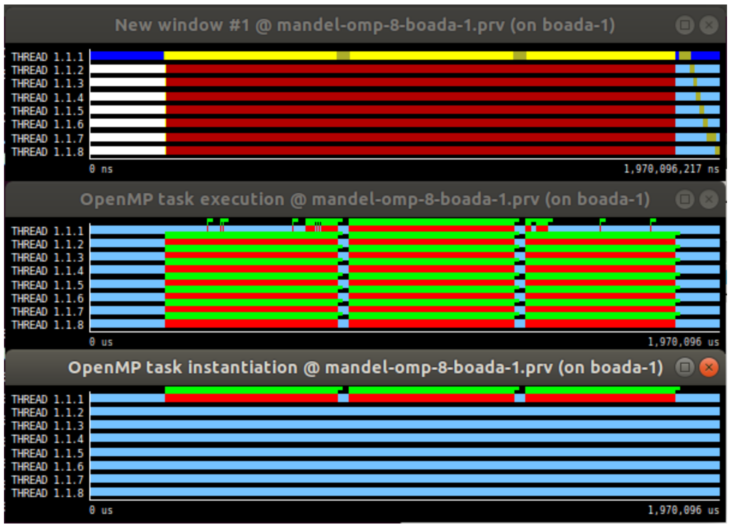

New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
1,970,096,217 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,970,096 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,970,096 us

Figure 8: Taskwait method Paraver

3.3 Taskloop tasking

Finally, we have used the version *using taskloop*.

```
#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){
        ...
    }
}
```

For taskloop we have tried different numbers of tasks, meaning different numbers of iterations per task (grainsize = total_Iterations/num_tasks). The numbers we have tried are 800,400,200,100,50,25,10,5,2,1. The following is the time difference among them:

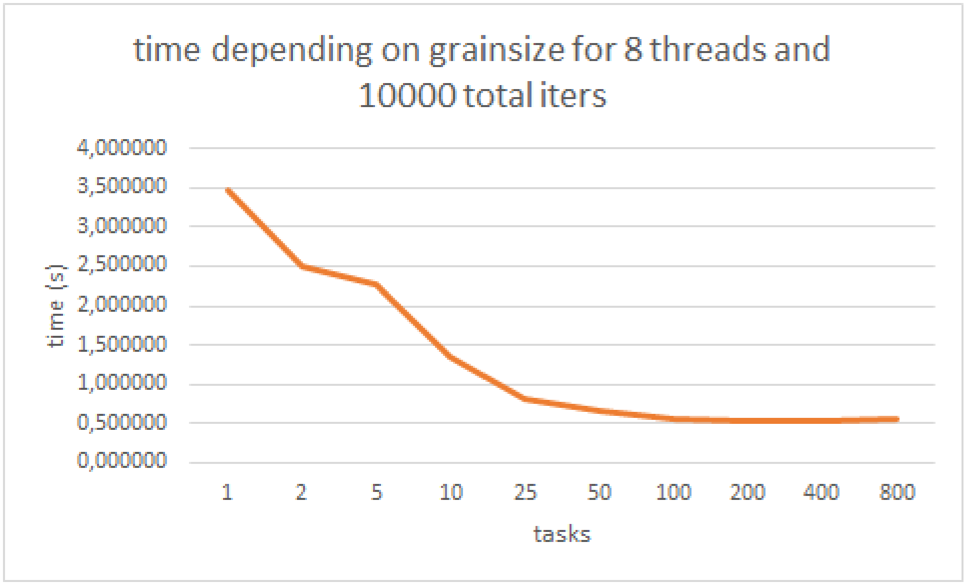


Figure 9: Taskloop time variations for different grainsizes

We have taken Paraver captures of every single number of tasks as well as strong scalability plots, here we will show some of them, please refer to the annex to see them all.

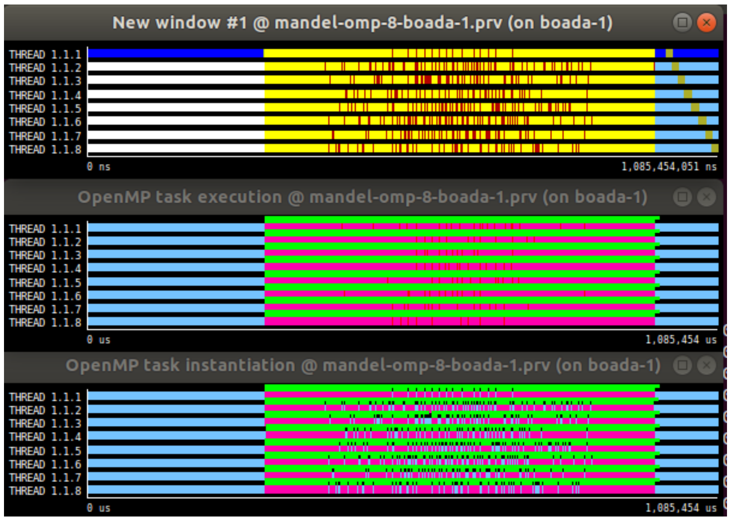

New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
1,085,454,051 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,085,454 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,085,454 us

Figure 10: Taskloop Paraver with 800 tasks

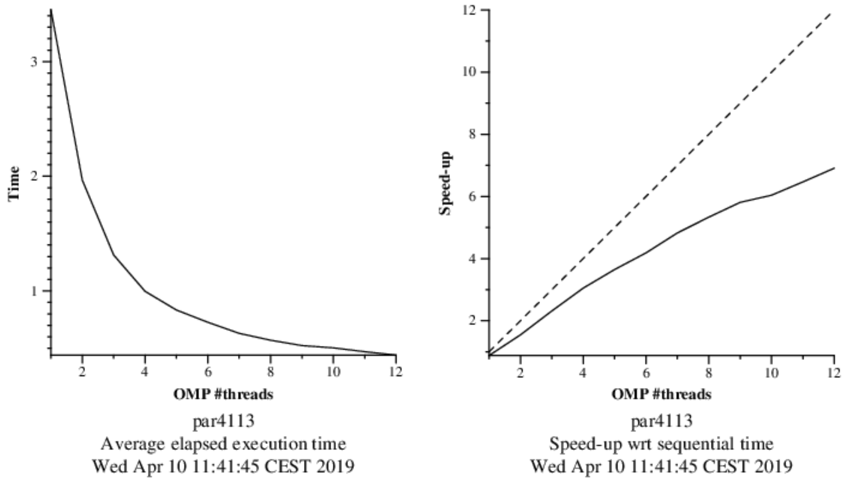


Figure 11: Taskloop Strong scalability plot with 800 tasks

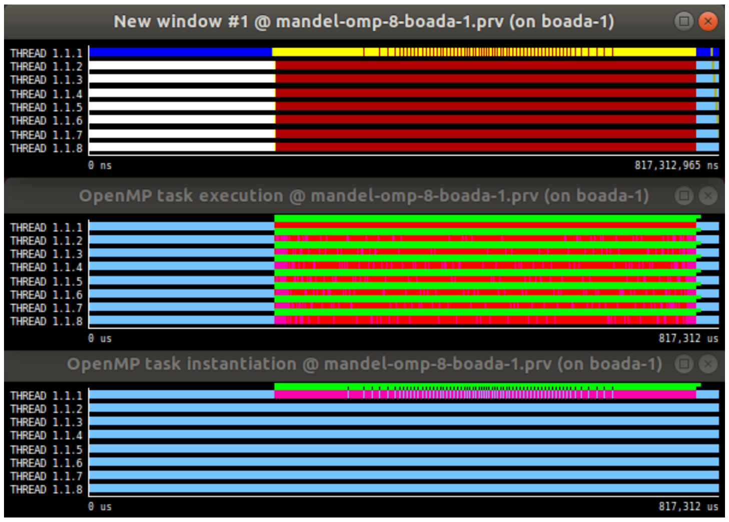

New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.8
0 ns
817,312,965 ns
0 us
817,312 us

Figure 12: Taskloop Paraver with 100 tasks

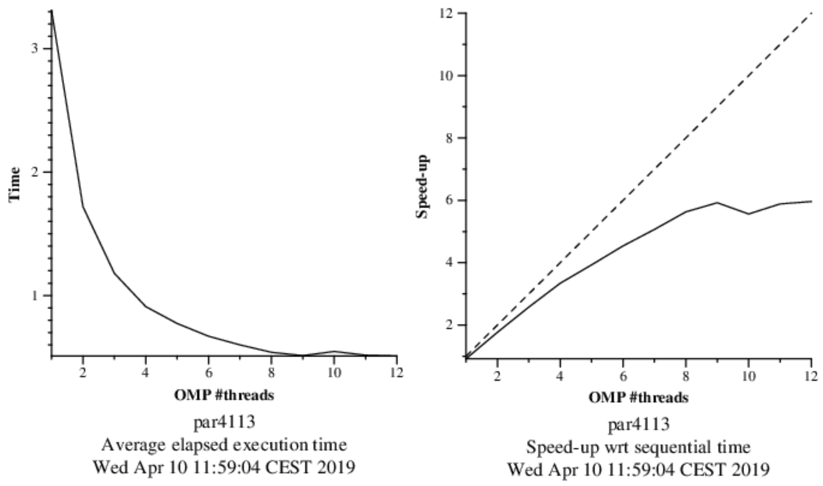


Figure 13: Taskloop Strong scalability plot with 100 tasks

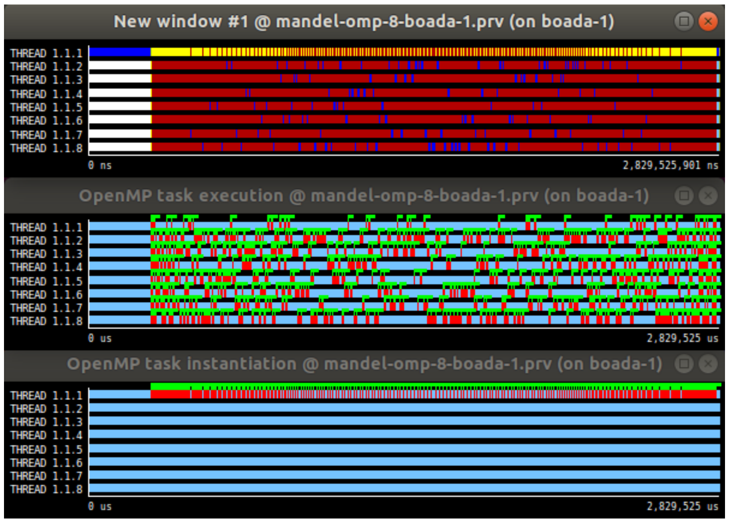

New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
2,829,525,901 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
2,829,525 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
2,829,525 us

Figure 14: Taskloop Paraver with 2 tasks

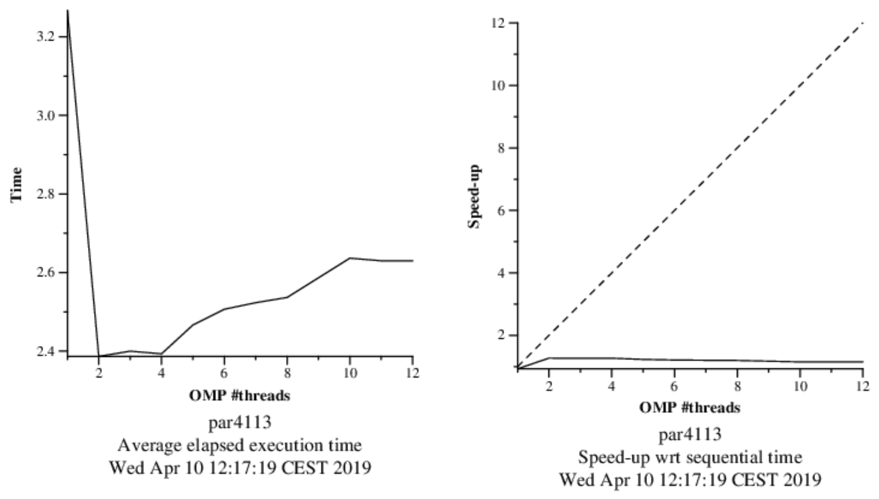


Figure 15: Taskloop Strong scalability plot with 2 tasks

With these previous graphics we can appreciate how task creation is only parallelized when there are a lot of tasks. This might happen because the more tasks there are, the smaller they each are, and so threads have time to execute them and then create more because there aren't more tasks in the queue. With larger tasks, however, threads are busy executing tasks most of the time and when they finish one there's already another waiting, so only 1 thread has time to create tasks.

Also, looking at the strong scalability plots we observe that for a big number of tasks the speedup keeps increasing even for ¿ 8 threads, which makes sense given that they can all be busy as there are many tasks. For smaller numbers of tasks, however, the speedup decays quickly as there is no real advantage to creating more tasks if there aren't threads there to execute them.

With this we conclude that out of the 3 methods, taskloop with a high number of tasks is the best as it gives lower execution times than the others.

OPTIONAL:

We have also tried taskgroup as an alternative to putting taskwait at the end of the inner for loop to wait for tasks to end before creating more. The difference is taskwait waits for tasks created in the region to end, but it won't wait for its children tasks. Instead, taskgroup waits for the tasks created inside the group, not all tasks, but it will wait also for the children tasks of the tasks in the group. This differences are not relevant for this code as no task is generating children tasks, so we expect a very similar performance.

```
#pragma omp parallel
#pragma omp single
for(row= 0; row<height; ++row){
    #pragma omp taskgroup
    for (col = 0; col < width; ++col){
        #pragma omp task firstprivate(row,col)
        {
            ...
        }
    }
}
```

Here are the Paraver capture and Strong scalability plot, which confirm the similarity:

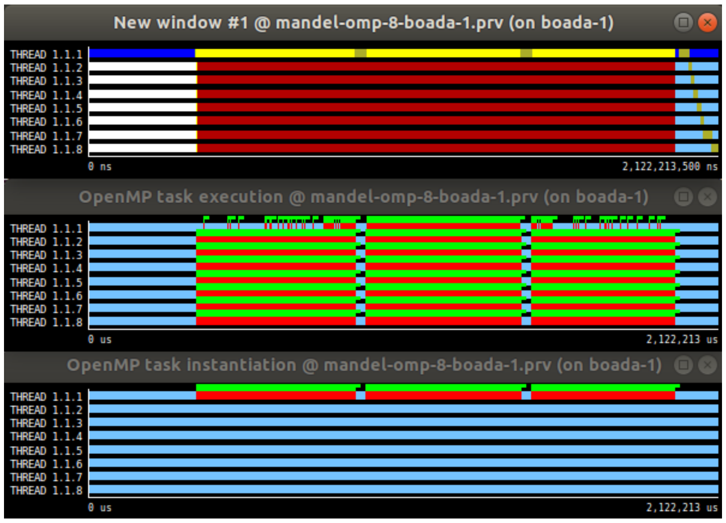
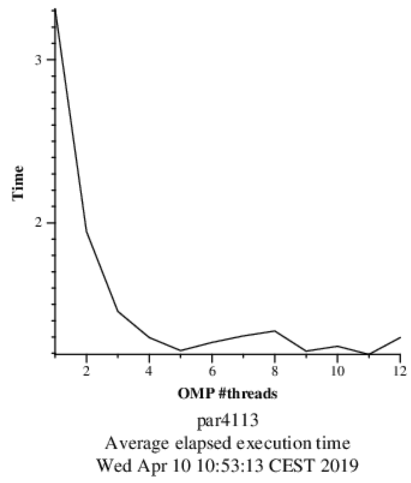

New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
2,122,213,500 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
2,122,213 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
2,122,213 us

Figure 16: Taskgroup Paraver with 2 tasks



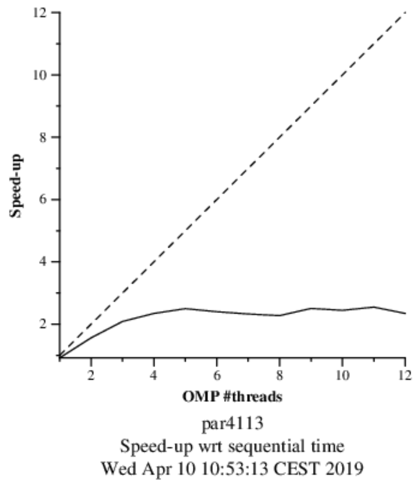


Figure 17: Taskgroup Strong scalability plot with 2 tasks

4 *Row* decomposition in *OpenMP*

From the point of view of granularity, the other option of parallelism we have seen in the assignment is the *row* decomposition. For the *row* decomposition, we have the following code:

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop nogroup num_tasks(800)
```

```
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        ...
    }
}
```

When executed, it gives us the following results:

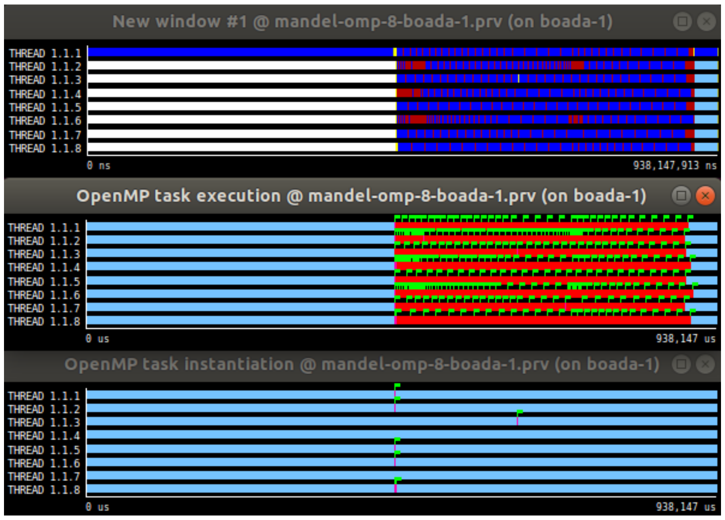
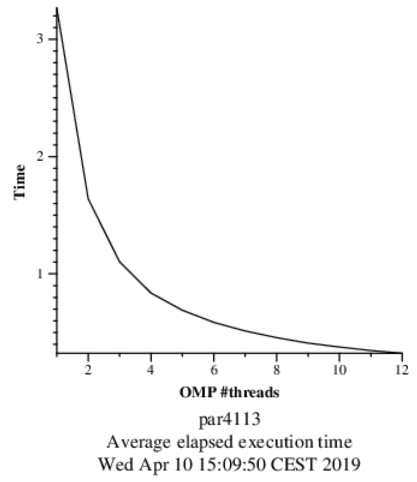

New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
938,147,913 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
938,147 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
938,147 us

Figure 18: Row decomposition with taskloop (Paraver traces)



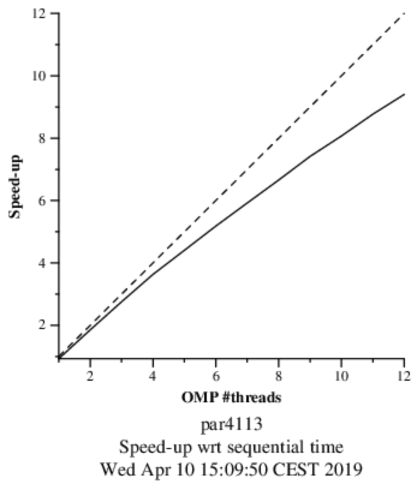


Figure 19: Row decomposition with taskloop (Strong scalability plot)

As it can be seen in the previous plots, it is actually faster than taskloop with point decomposition instead of row. This is probably due to the row decomposition creating less tasks and thus producing less overhead, but still creating enough tasks to have all threads busy working without wasting time. Of all the methods explored, Row decomposition with taskloop and a high number of tasks is the best parallelization method.

5 For-based parallelization

We have decided to also try the for parallelization method instead of creating tasks, as we hope this is going to have a smaller overhead since each iteration is directly assigned to a thread instead of creating a task and then assigning it. The schedule of the *for* directive in OpenMP decides how everything is going to be distributed along the threads.

In this section we are going to study the behaviour for the Mandelbrot set for each type of schedule: **static, dynamic and guided**.

5.1 Static

The code to parallelize using the **static** for method is the following:

```
    /* Calculate points and save/display */
    #pragma omp parallel for schedule(static,10) private(col,row)
    for (row = 0; row < height; ++row) {
        for (col = 0; col < width; ++col) {
            complex z, c;
            ...
        }
    }
```

The result is the following:

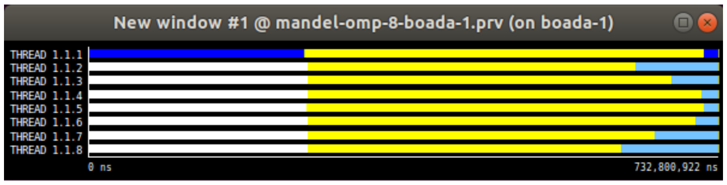


Figure 20: For parallelization (Paraver)

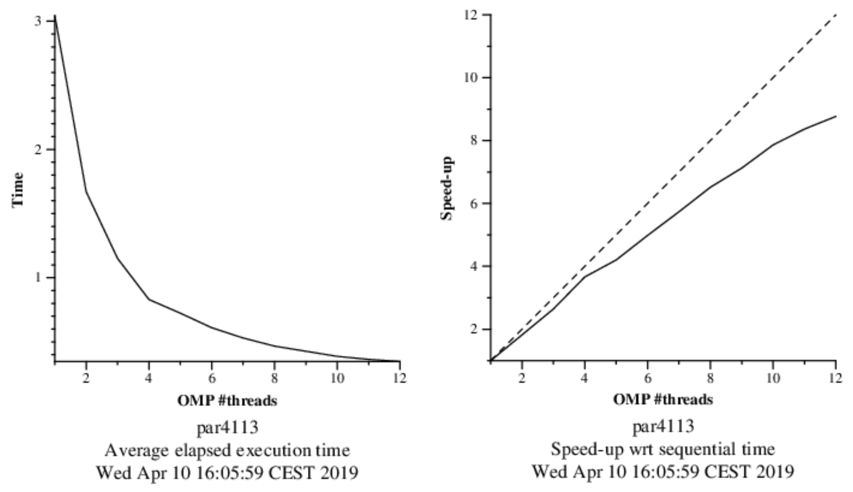


Figure 21: For parallelization (Strong scalability)

5.2 Dynamic

The code to parallelize using the **dynamic** for method is the following:

```
/* Calculate points and save/display */
  #pragma omp parallel for schedule(dynamic,10) private(col,row)
  for (row = 0; row < height; ++row) {
      for (col = 0; col < width; ++col) {
          complex z, c;
          ...
      }
  }
```

The result is the following:

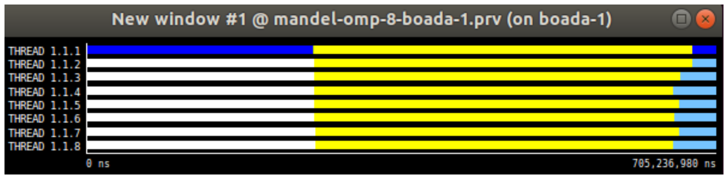
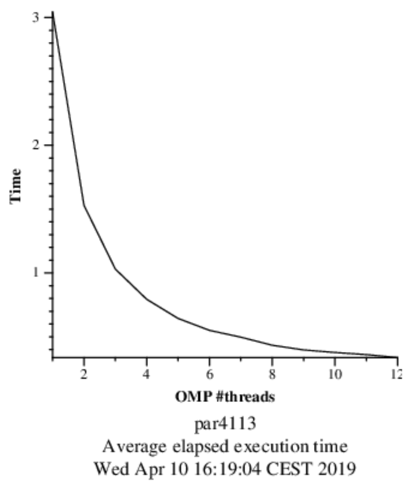


Figure 22: For parallelization (Paraver)



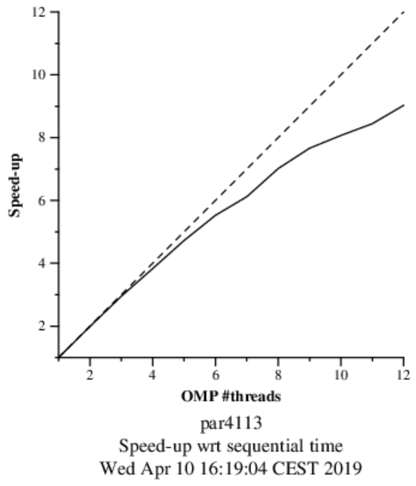


Figure 23: For parallelization (Strong scalability)

5.3 Guided

The code to parallelize using the **guided** for method is the following:

```
/* Calculate points and save/display */
#pragma omp parallel for schedule(guided,10) private(col,row)
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
        complex z, c;
        ...
    }
}
```

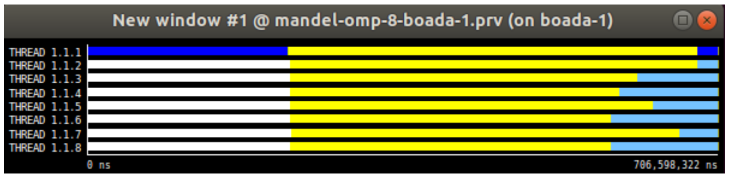


Figure 24: For parallelization (Paraver)

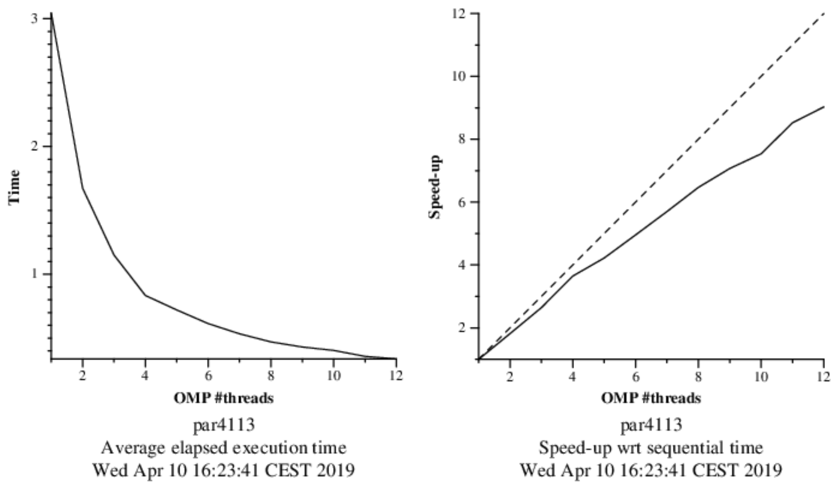


Figure 25: For parallelization (Strong scalability)

As you can see we have chosen a **chunk size of 10**. The main reason for that was because we did not want to have ”tasks” too large (as we saw that small tasks are faster previously, we wanted to simulate this for the ”for parallelization”) and we decided that 10 was a good value for this approach.

The plots we obtained by executing each of the schedules showed that static is slightly slower than dynamic and guided, and this is probably because there is an unbalanced load between threads. Guided is slower than dynamic by a matter of 1 ms or so, we expected to see a clearer difference but they behave very similarly time wise.

Finally, comparing them to the tasks versions, it is clear that for loop parallelization without tasks (especially dynamic and guided scheduling) have better performance than tasks.

6 Conclusion

The main goal of this laboratory was to learn and compare different parallelization methods. After completing the different experiments we can say that we have learned how a sequential code can be distributed into different tasks of different grainsize and we have seen different task generation strategies to parallelize the code. We have also been able to compare them to for loop parallelization without tasks, with different schedulings.

Also, we have been able to observe that not always the bigger the number of tasks, the greater the results, and analyse *a posteriori* the possible reasons for this behaviour. There are several techniques to deal with tasks and each one has a different use case. For the most part, however, in this report we have found using taskloop with a small grainsize is the best task strategy, but parallel for is even faster in time, specially with dynamic and guided scheduling.

7 Annex: Pictures

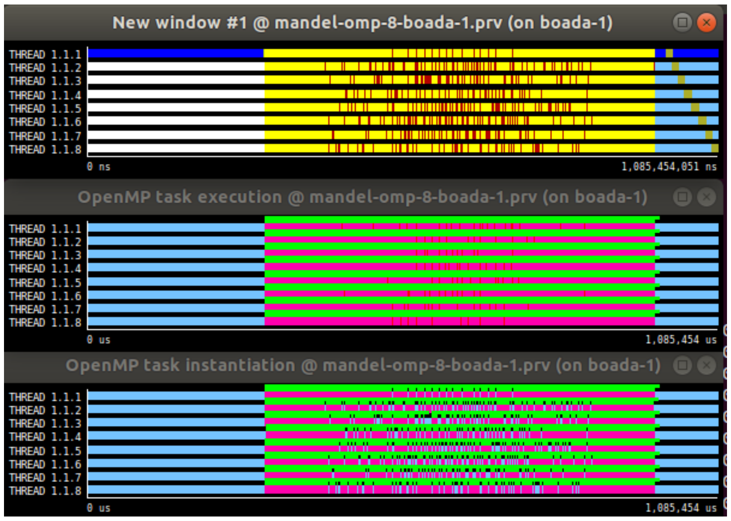

New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
1,085,454,051 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,085,454 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,085,454 us

Figure 26: Taskloop Paraver with 800 tasks

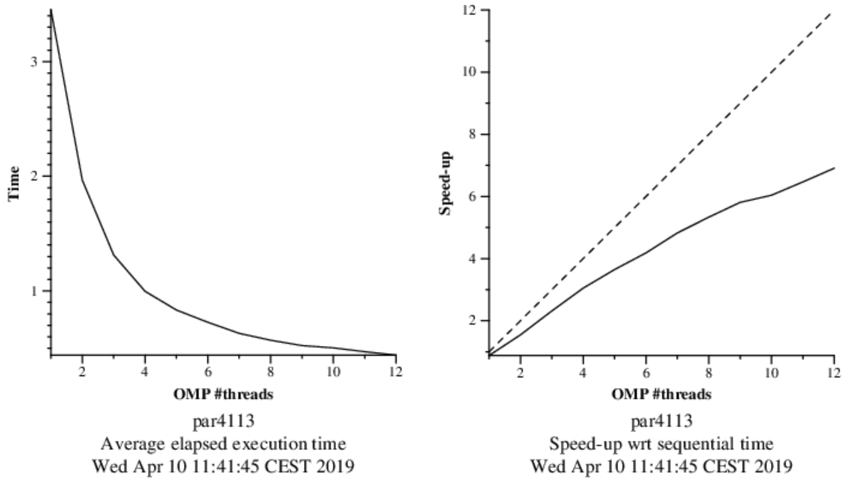


Figure 27: Taskloop Strong scalability plot with 800 tasks

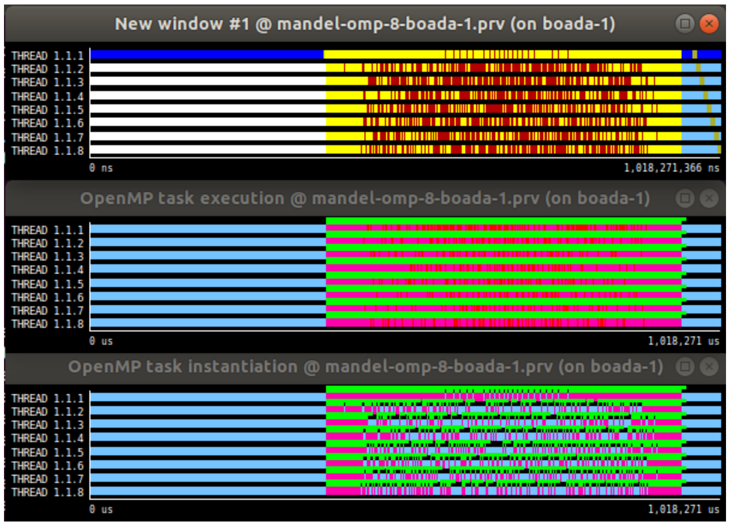

New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
1,018,271,366 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,018,271 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,018,271 us

Figure 28: Taskloop Paraver with 400 tasks

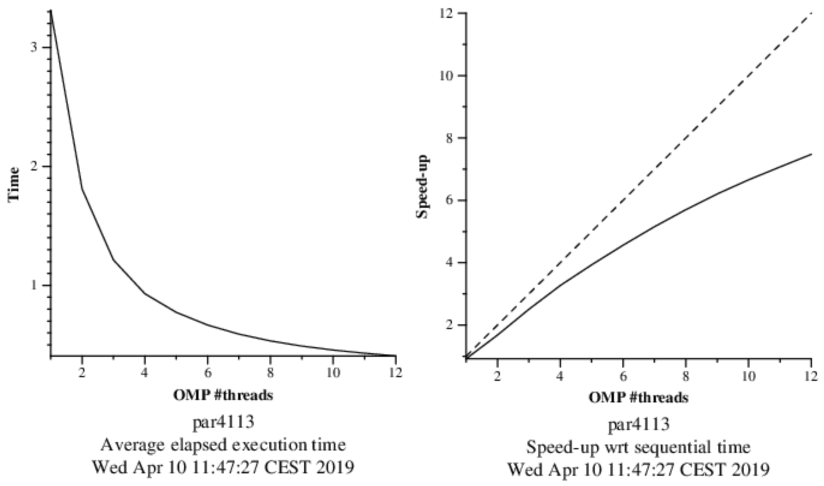
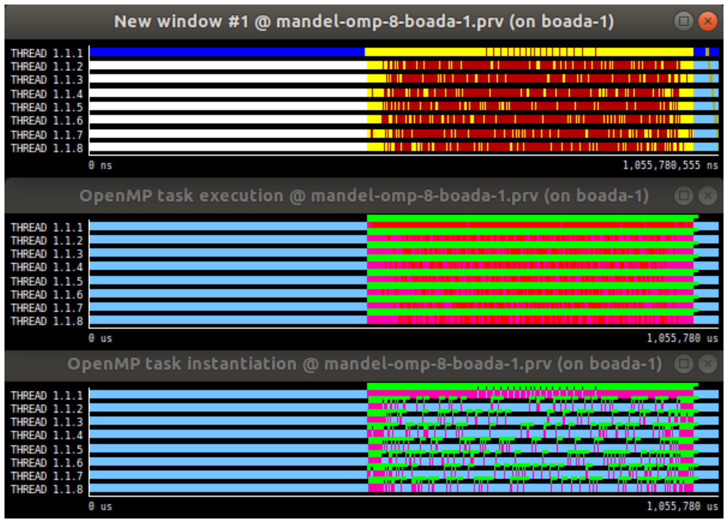


Figure 29: Taskloop Strong scalability plot with 400 tasks



New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
1,055,780,555 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,055,780 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,055,780 us

Figure 30: Taskloop Paraver with 200 tasks

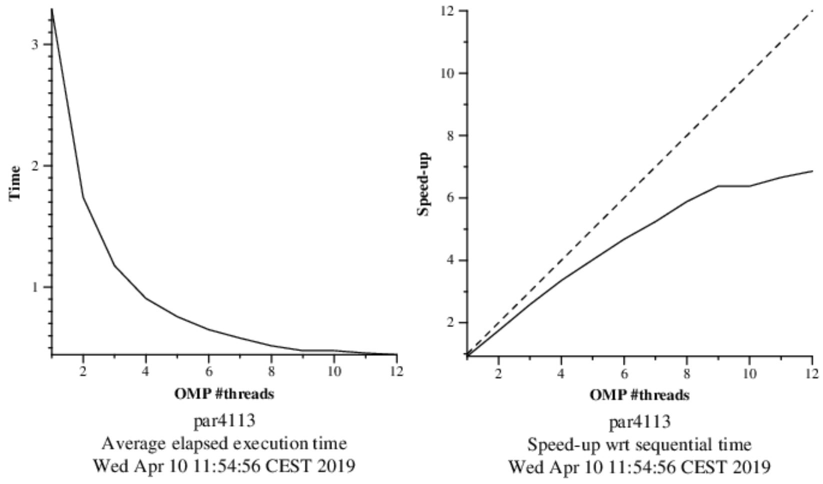


Figure 31: Taskloop Strong scalability plot with 200 tasks

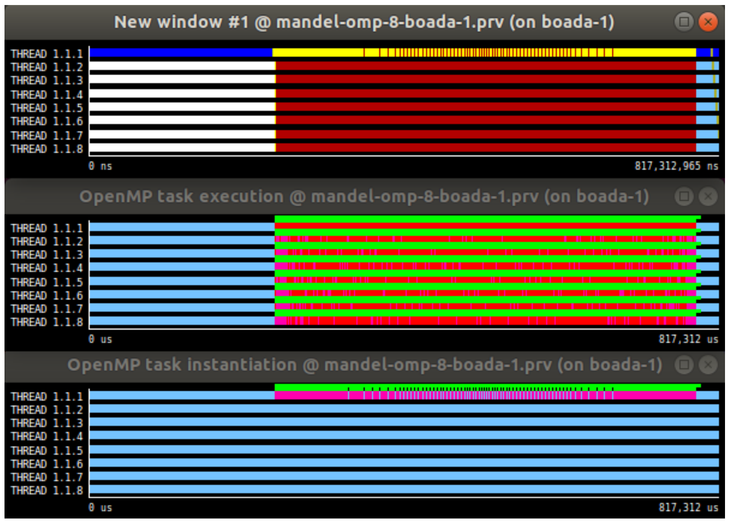

New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
817,312,965 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
817,312 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
817,312 us

Figure 32: Taskloop Paraver with 100 tasks

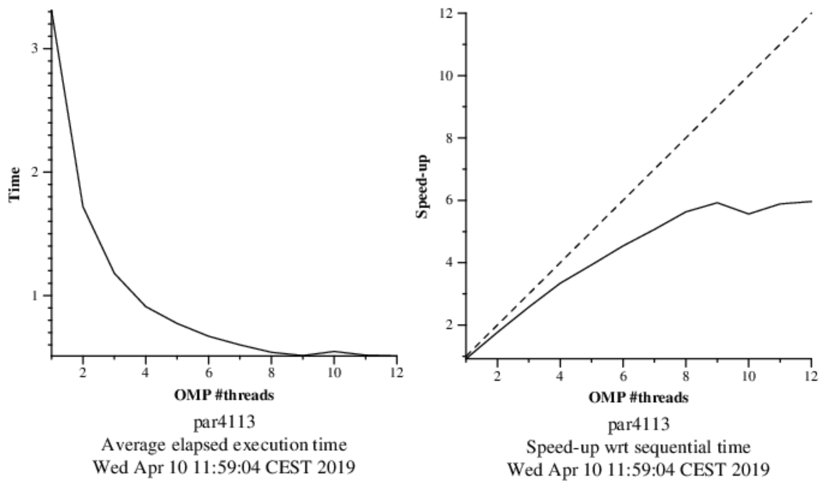


Figure 33: Taskloop Strong scalability plot with 100 tasks

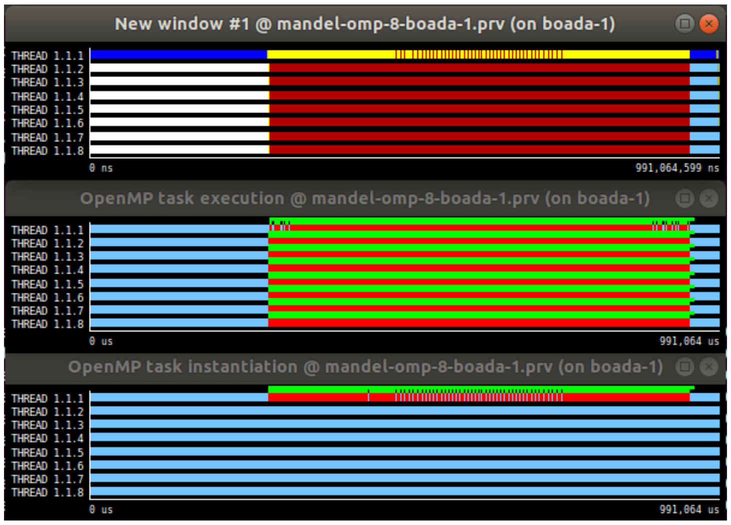


Figure 34: Taskloop Paraver with 50 tasks

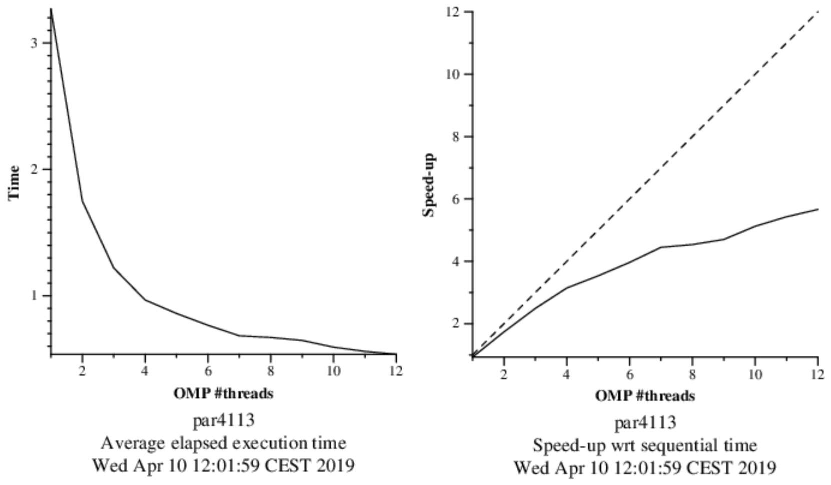
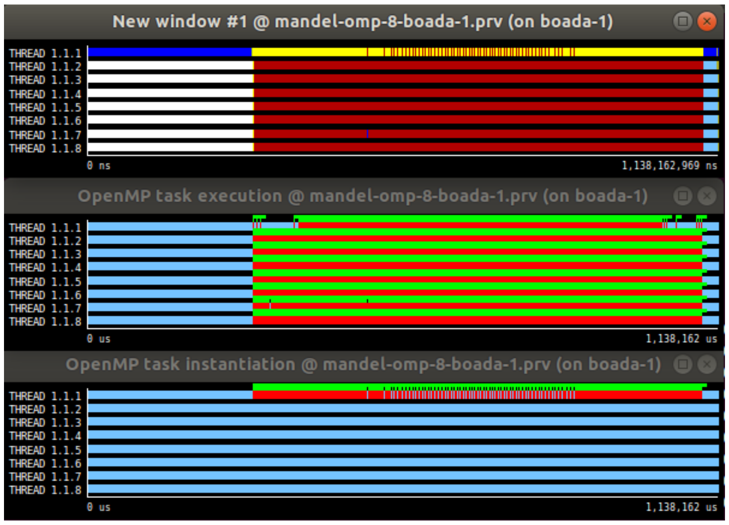


Figure 35: Taskloop Strong scalability plot with 50 tasks



New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
1,138,162,969 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,138,162 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,138,162 us

Figure 36: Taskloop Paraver with 25 tasks

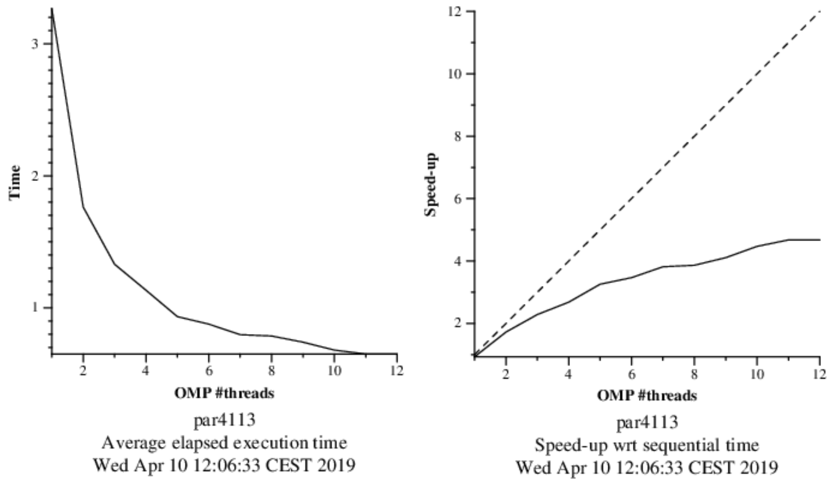
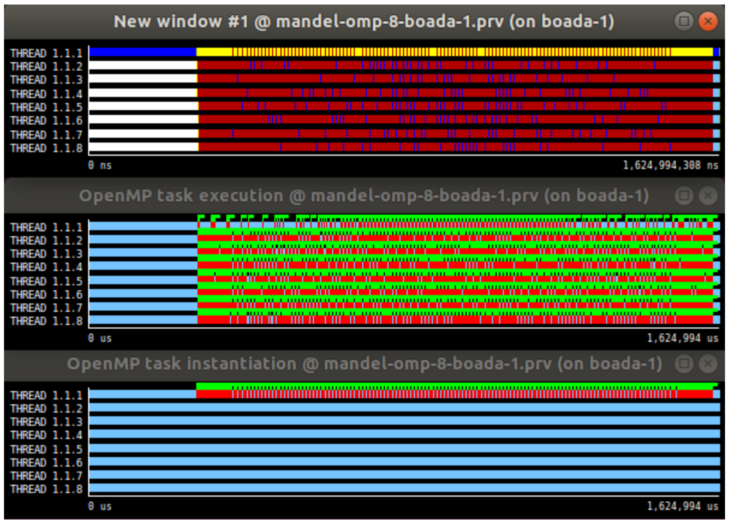


Figure 37: Taskloop Strong scalability plot with 25 tasks



New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
1,624,994,388 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,624,994 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
1,624,994 us

Figure 38: Taskloop Paraver with 10 tasks

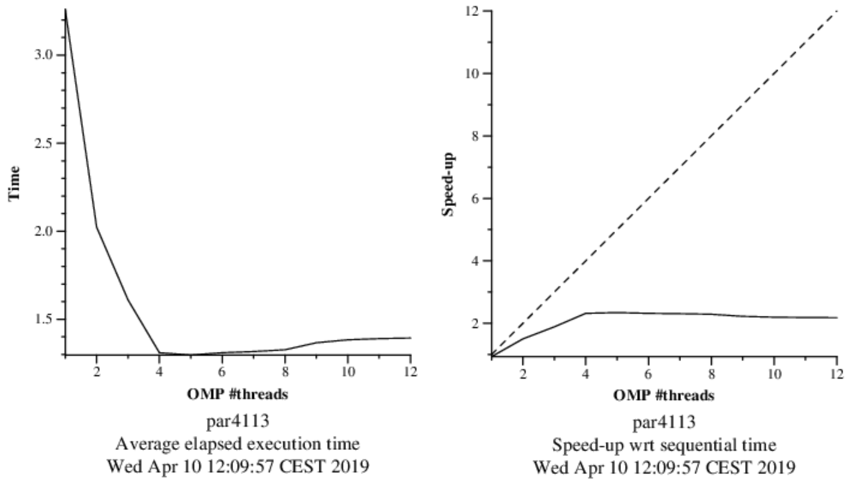


Figure 39: Taskloop Strong scalability plot with 10 tasks

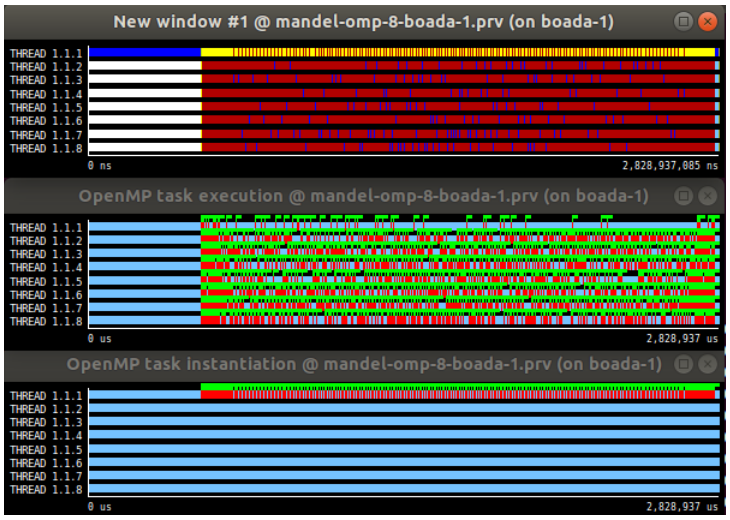

New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
2,828,937,085 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
2,828,937 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
2,828,937 us

Figure 40: Taskloop Paraver with 5 tasks

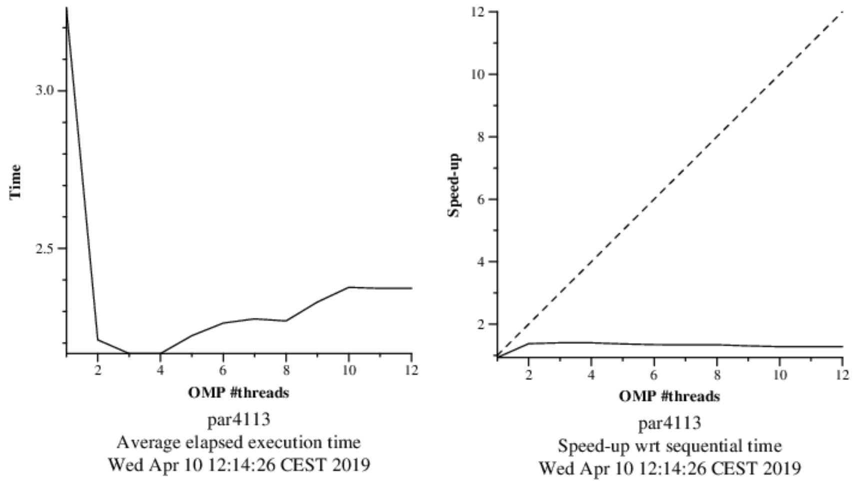


Figure 41: Taskloop Strong scalability plot with 5 tasks

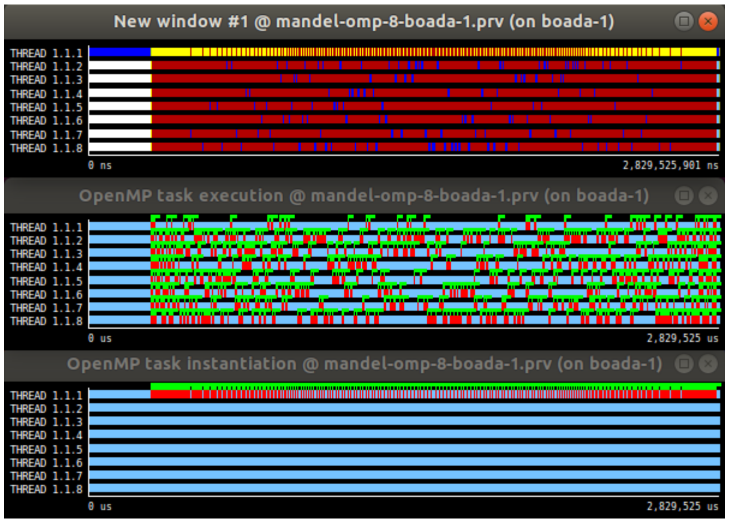

New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
2,829,525,901 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
2,829,525 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
2,829,525 us

Figure 42: Taskloop Paraver with 2 tasks

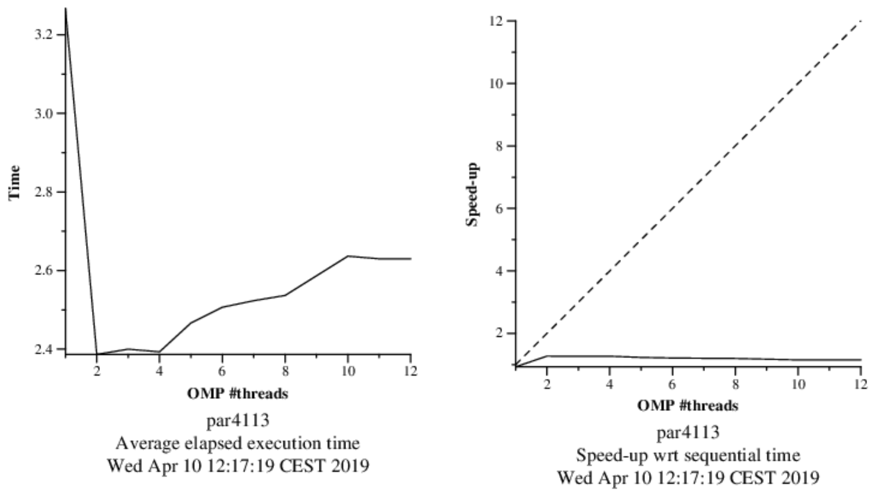


Figure 43: Taskloop Strong scalability plot with 2 tasks

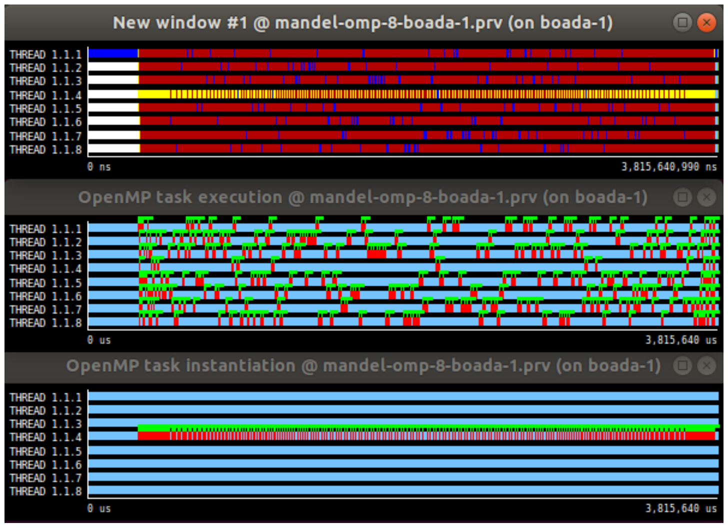

New window #1 @ mandel-omp-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
3,815,640,990 ns
OpenMP task execution @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
3,815,640 us
OpenMP task instantiation @ mandel-omp-8-boada-1.prv (on boada-1)
0 us
3,815,640 us

Figure 44: Taskloop Paraver with 1 tasks