PAR Laboratory Assignment

Lab 3: Iterative task decomposition with OpenMP: the computation of the Mandelbrot set

20/10/2021

Fall 2021-22

Index

1. Task decomposition and granularity analysis

1.1. Row strategy

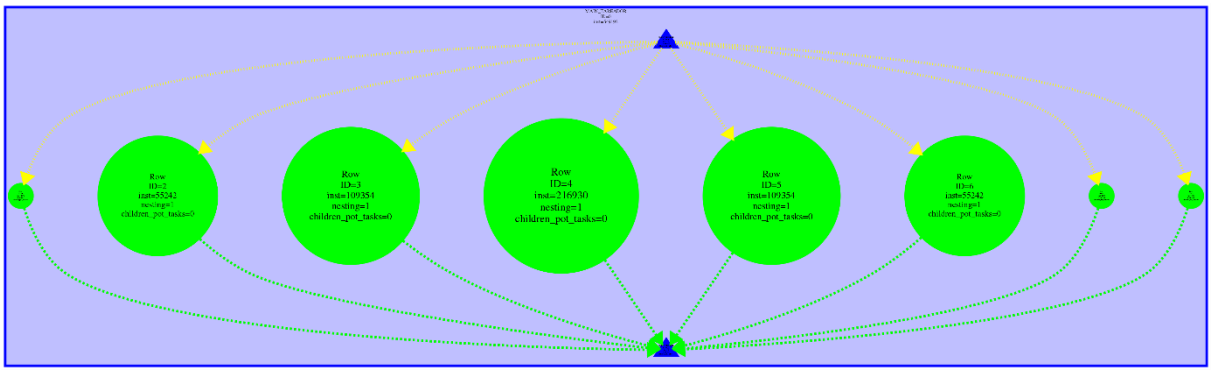


Figure 1.1.1: Dependency graph with row granularity and without additional options

The two main characteristics we observe are:

- There is no dependency between tasks
- Not all the tasks have the same amount of work.

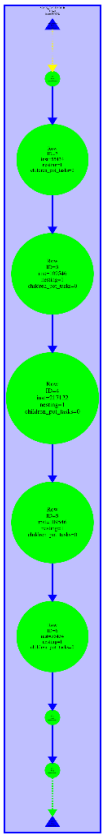


Figure 1.1.2: Dependency graph with row granularity with the -d option

The two main characteristics we observe are:

- There is dependency between all tasks, that's the difference with the previous execution.
- As same as the previous execution not all the tasks have the same amount of work.

The part of the code that is making big difference with the previous case is:

```
if (output2display) {
   /* 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);
   }
}
```

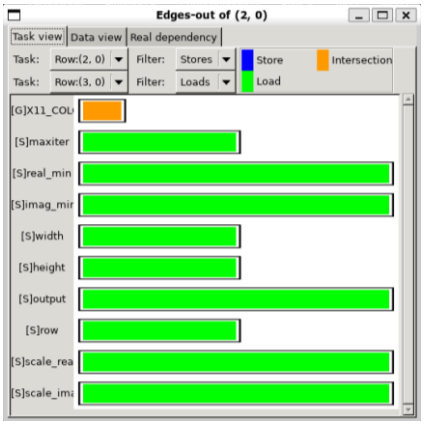


Figure 1.1.3: Dependency variables of -d option

At the above image we can observe that the dependency between tasks is because a variable named X11_COLOR_fake, that corresponds to color variable from the above code.

We can protect this section of code in the parallel OpenMP using the command #pragma omp critical, to ensure exclusive access to variable color.

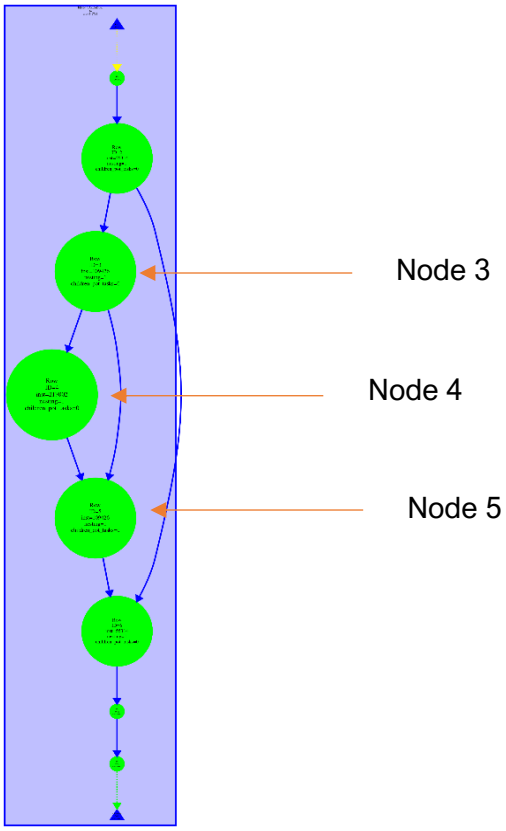


Figure 1.1.4: Dependency graph with row granularity and with -h option

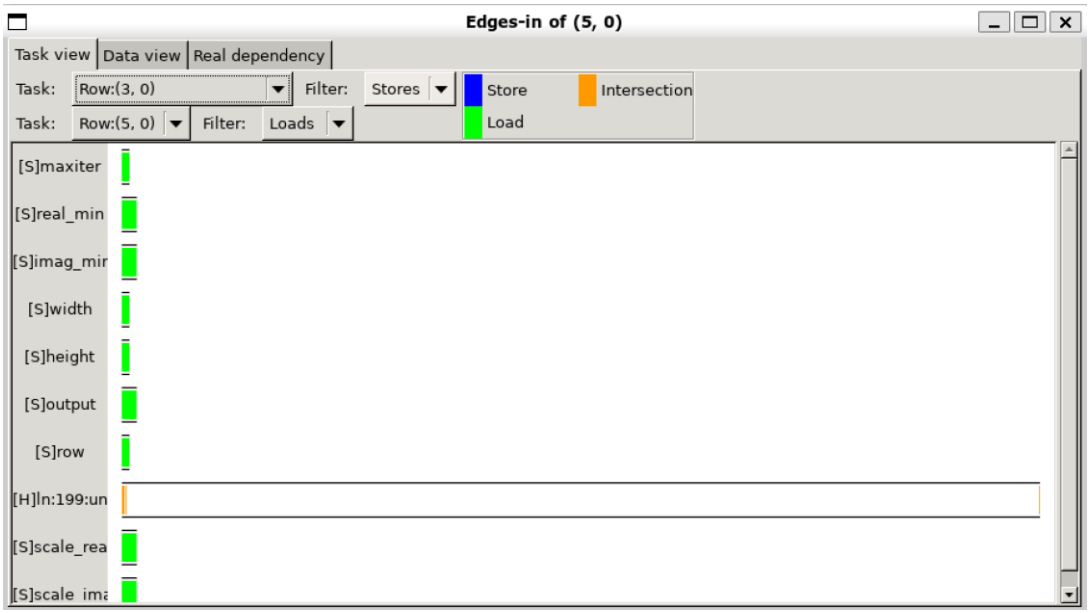


Figure 1.1.5: Dependency variables of -h option

The above figure shows us the dependency between tasks, if we look at the orange bar, we deduce that this is the cause of dependency between nodes 3 and 5 and for the rest of the nodes. This dependency corresponds to a variable named ln:199:uncast(llvm_internal), that corresponds to a unknown variable.

The part of the code that is making big difference with the previous case is:

```
if (output2histogram) histogram[k-1]++;
```

Finally, to protect this part of the code when using OpenMP we will add #pragma omp atomic.

1.2. Point strategy

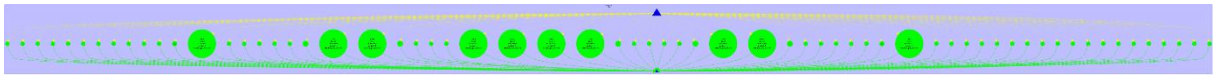


Figure 1.2.1: Dependency graph with point granularity and without additional options

Notice that the number of tasks increases with this version compared to the row strategy, due we replace first loop definition with a finer grained tasks defined inside the second loop. As result of that the dependency between tasks does not change and not all the tasks have the same amount of work.

If we use the -d option we obtain the exact same result as the row strategy, but in this case with a fine-grained task.

Finally, when we use the -h option there are two differences between the row strategy and point strategy:

- The number of tasks increases because of the task is generated inside the second loop not the first.
- The tasks remain with different amount of work, but the dependency between the nodes changes, on one hand we have that all the nodes with a bigger amount of work depends on each other and for the other hand the tasks with less amount of work also depends on each other, but a node with a bigger amount of work do not depends on a node that has a lower amount of work, or vice versa.

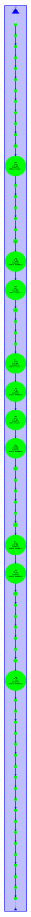


Figure 1.2.2: Dependency graph with point granularity with the -d option

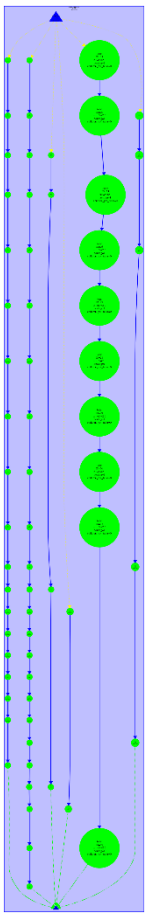


Figure 1.1.3: Dependency graph with point granularity and with -h option

2. *Point* decomposition in OpenMP

2.1 Point strategy implementation using task

The first parallelizable version of this program consists of creating a task for each point to be calculated (see Figure 2).

```
#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: First parallel version with task construct

The *firstprivate* clause is important to avoid dating races.

After executing the program for 1 and 8 threads we obtain the following data:

For 1 thread total execution time (in seconds): 3.974946

For 8 thread total execution time (in seconds): 1.321656

The resulting speed-up is the one shown in Figure 3.

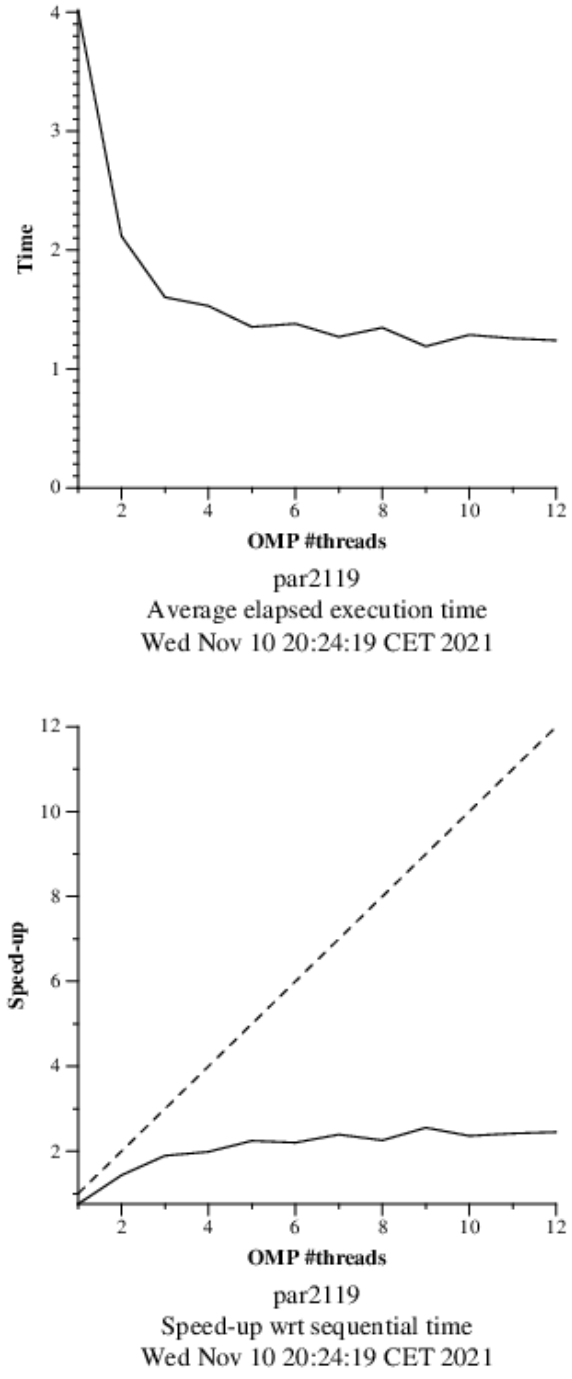


Figure 3: Speed-up for task version

Notice the improvement in Speed-up by increasing the number of threads, we can also notice that the improvement is getting lower because of we are getting closer to the maximum parallelization for these tasks.

In this execution, thread 1 is the one in charge of creating the tasks while the others (2-8) will execute those tasks. We can see that the task execution threads (2-8) spend most of the time

in synchronization (Figure 4), if we expand the timeline we can see that they are executing tasks (Figure 5).

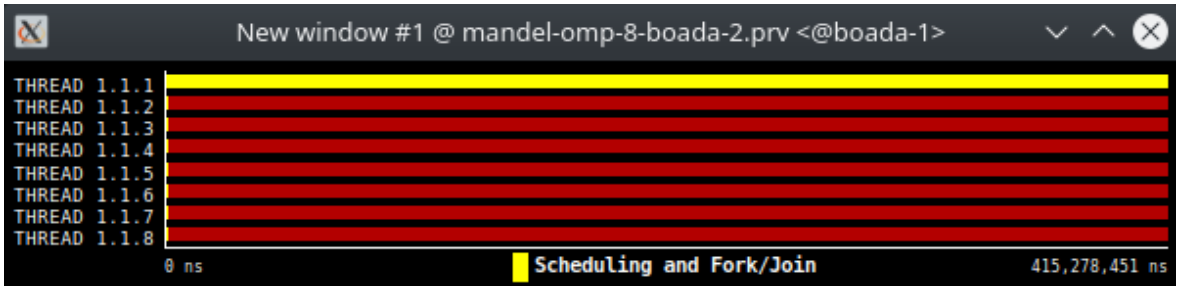


Figure 4: Timeline window

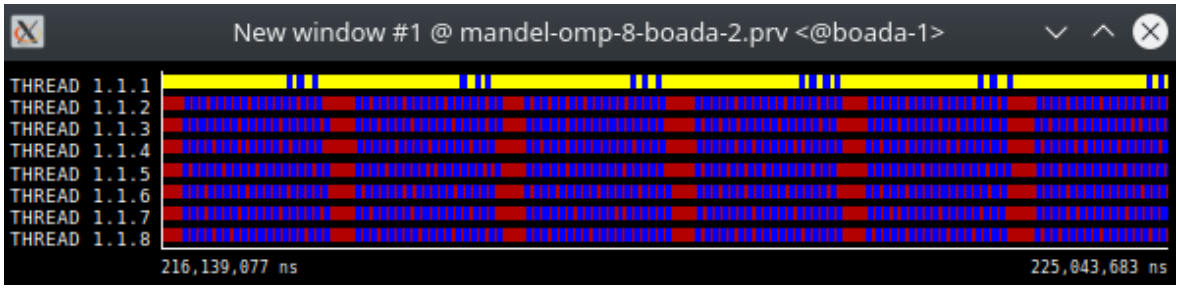


Figure 5: Timeline window extended

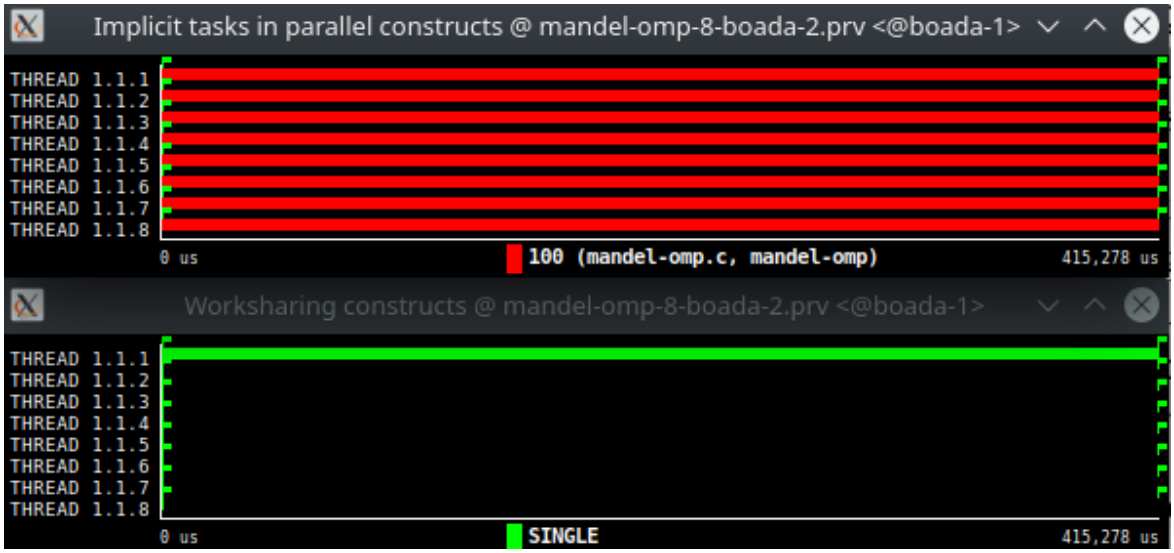
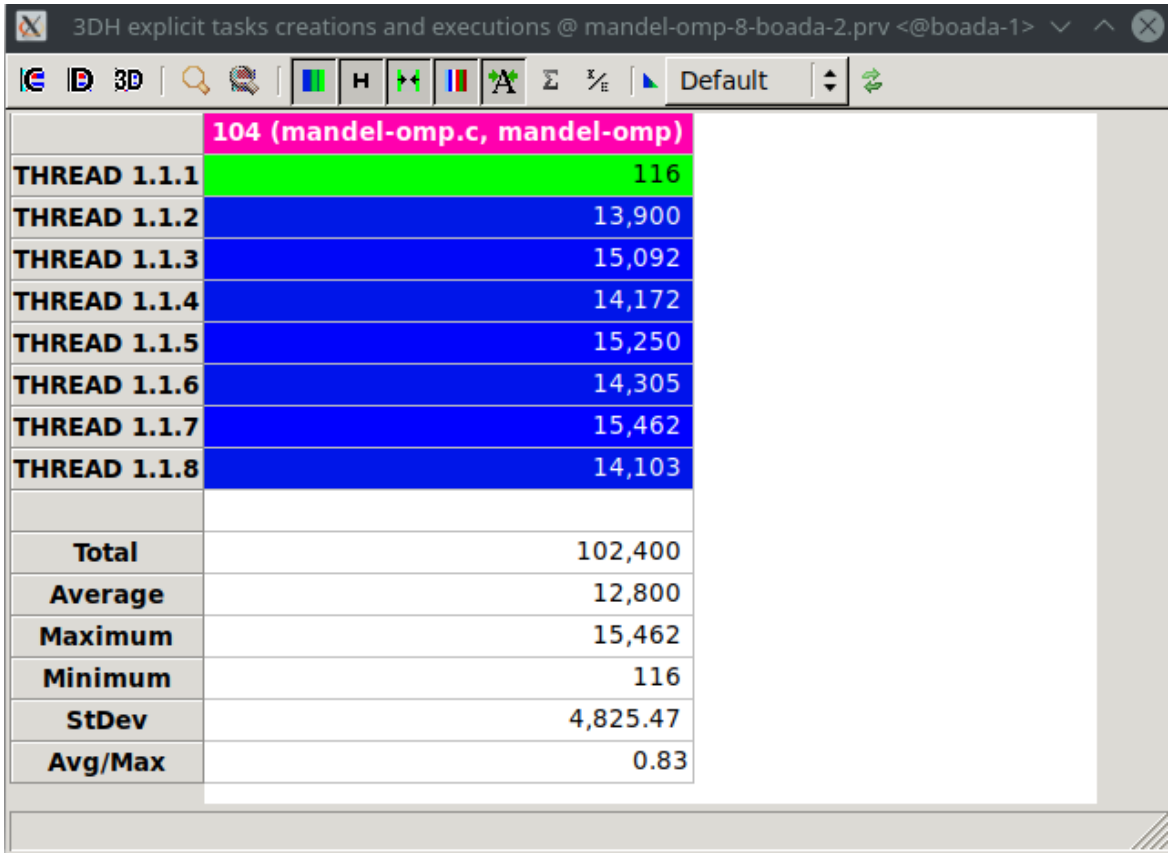


Figure 6: implicit tasks in parallel constructs (top) and work-sharing construct (bottom)

In Figure 7 we can see that the first thread oversees creating the tasks while the other threads (2-8) are executing them.

Figure 7: Explicit tasks function created and executed

In figure 8 we can see the number of tasks that each thread executes and other data on the number of tasks.



	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	116
THREAD 1.1.2	13,900
THREAD 1.1.3	15,092
THREAD 1.1.4	14,172
THREAD 1.1.5	15,250
THREAD 1.1.6	14,305
THREAD 1.1.7	15,462
THREAD 1.1.8	14,103
Total	102,400
Average	12,800
Maximum	15,462
Minimum	116
StDev	4,825.47
Avg/Max	0.83

Figure 8: Profile of explicit tasks creation and execution

With the histograms we can see the high dispersion of the tasks, the amount of tasks executed in a specific task duration (Figure 9) and the percentage of executing tasks with a certain task duration (Figure 10).

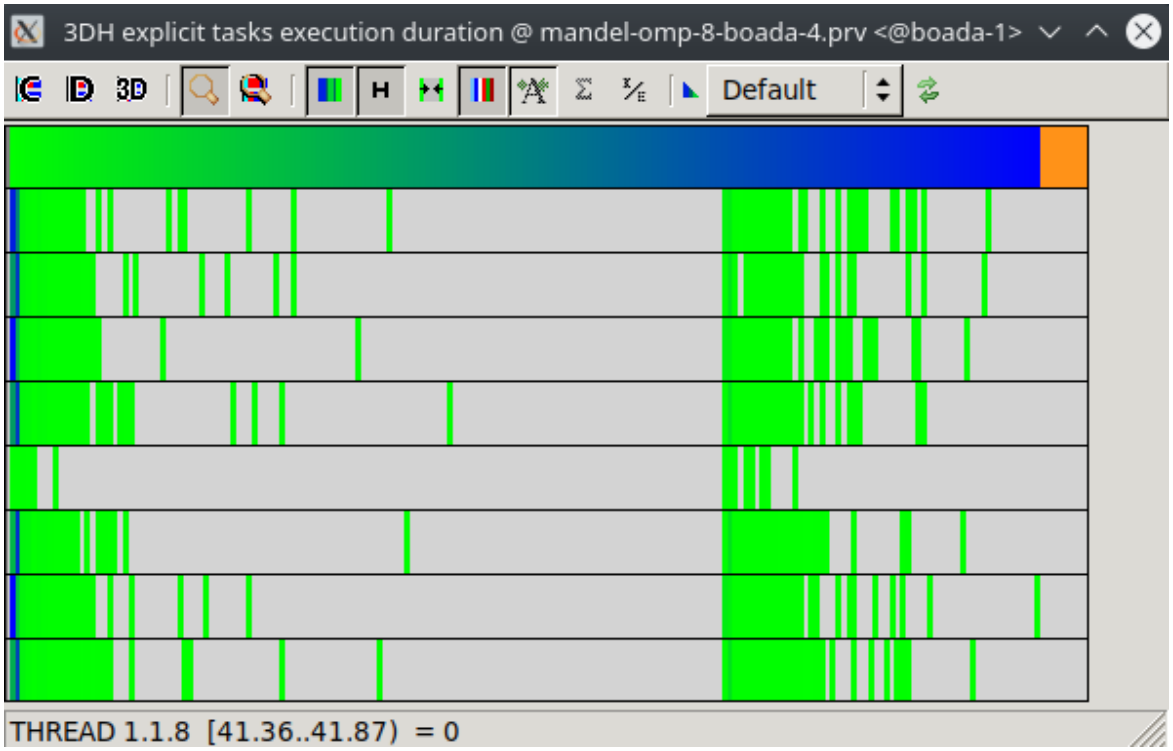


Figure 9: Histogram of explicit tasks execution duration (num bursts)

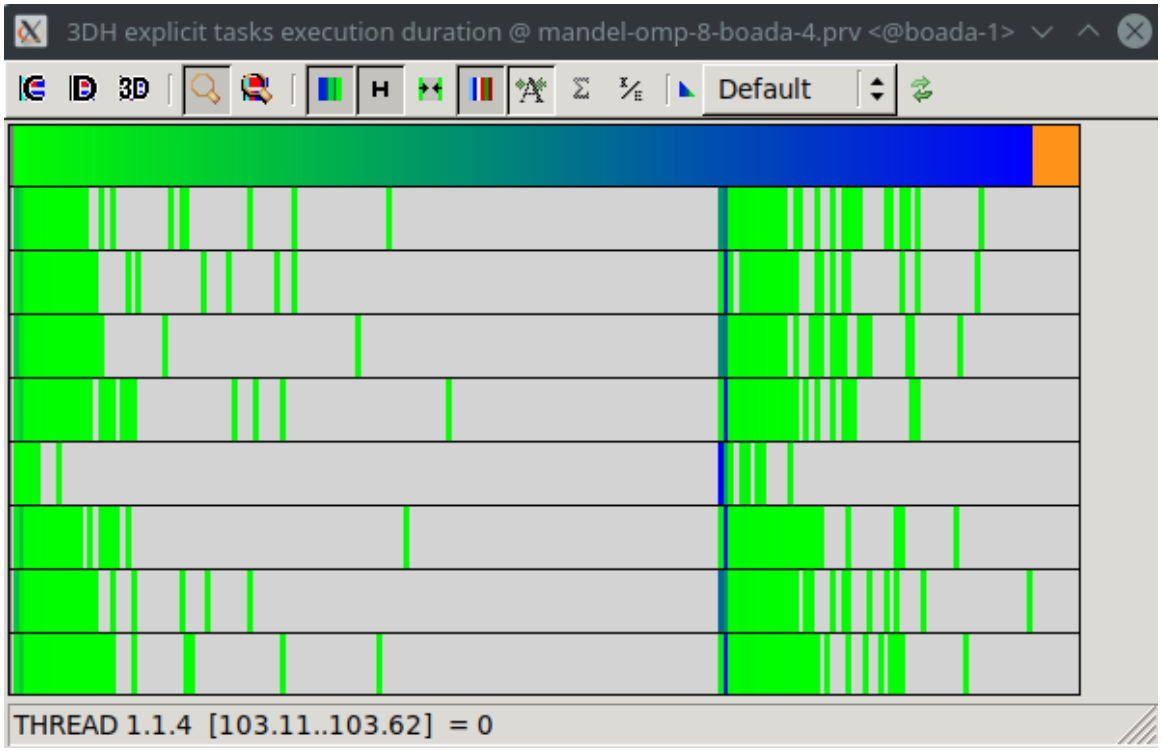
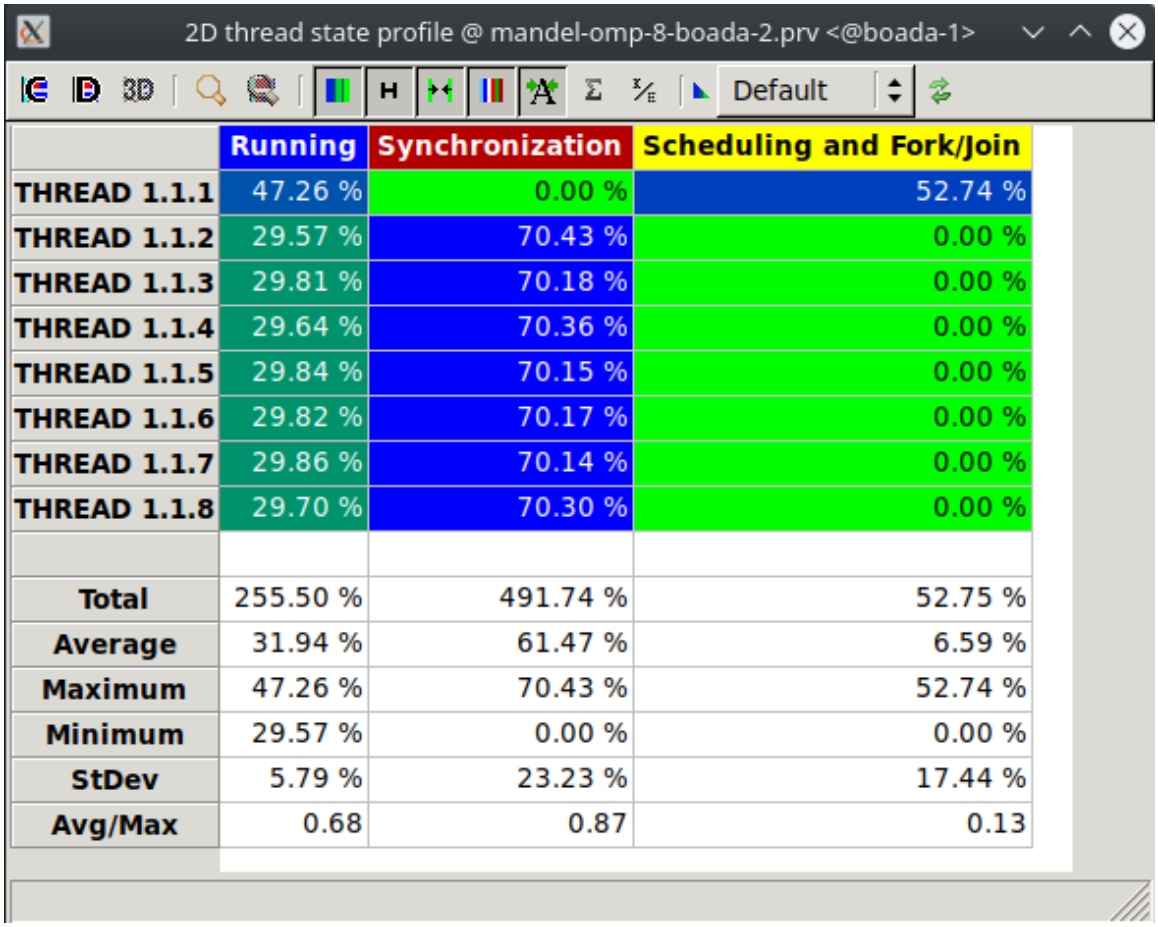


Figure 10: Histogram of explicit tasks execution duration (% time)

Which the “Thread state profile” hint we can see the percentage of synchronization time for each thread (Figure 11). During parallel execution, the threads that execute the tasks (2-8) consume about 70% of the time synchronizing, in other words, without doing any useful work.



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	47.26 %	0.00 %	52.74 %
THREAD 1.1.2	29.57 %	70.43 %	0.00 %
THREAD 1.1.3	29.81 %	70.18 %	0.00 %
THREAD 1.1.4	29.64 %	70.36 %	0.00 %
THREAD 1.1.5	29.84 %	70.15 %	0.00 %
THREAD 1.1.6	29.82 %	70.17 %	0.00 %
THREAD 1.1.7	29.86 %	70.14 %	0.00 %
THREAD 1.1.8	29.70 %	70.30 %	0.00 %
Total	255.50 %	491.74 %	52.75 %
Average	31.94 %	61.47 %	6.59 %
Maximum	47.26 %	70.43 %	52.74 %
Minimum	29.57 %	0.00 %	0.00 %
StDev	5.79 %	23.23 %	17.44 %
Avg/Max	0.68	0.87	0.13

Figure 11: Thread state profile

2.2 Point strategy with granularity control using taskloop

For this second version of the program, we are going to use taskloop construct for the column iterations. We will modify the code as follows (Figure 12).

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

Figure 12: Second parallel version with taskloop construct

As we can see, the execution time for 8 threads is lower than the previous version, so the speed-up increases slightly (Figure 13).

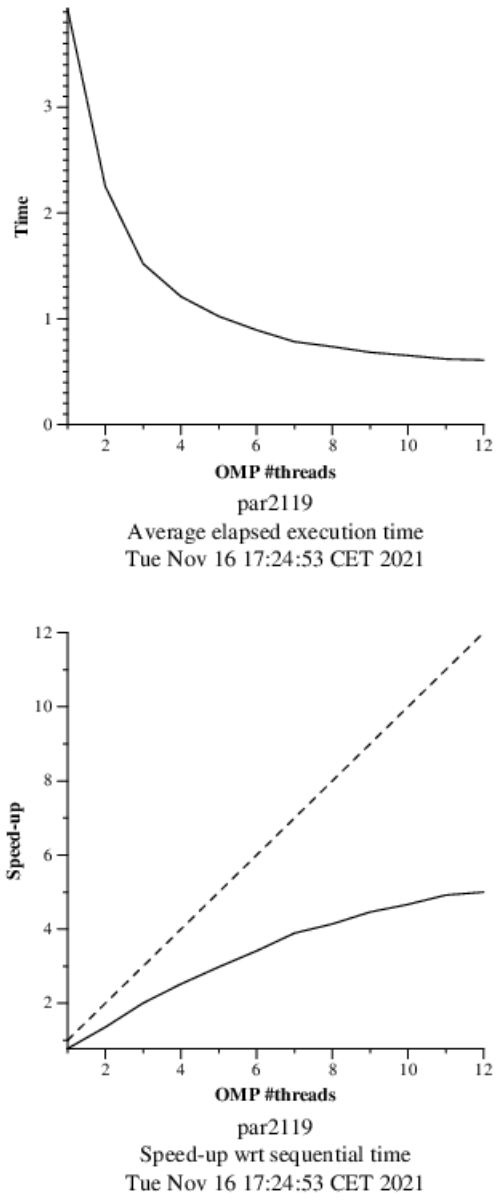


Figure 13: Speed-up for taskloop version

Unlike the previous case, where all the execution time of the threads was shown in red (synchronization), here we can observe some task executions without the need to expand (Figure 14). In this execution the thread that creates the tasks is thread (Figure 14 & 15).

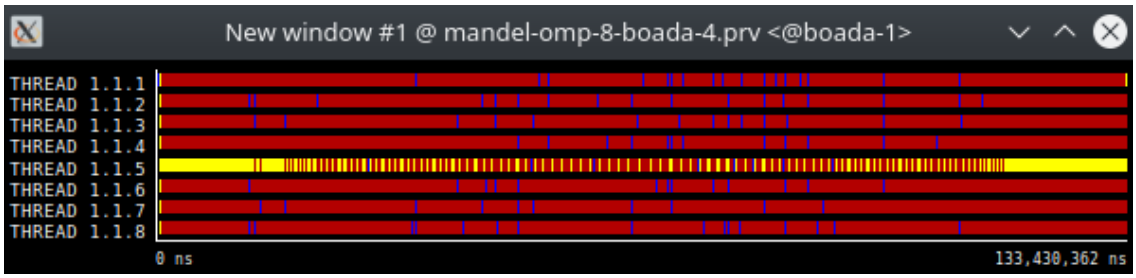


Figure 14: Timeline window v2

Figure 15: Explicit tasks function created and executed

The dispersion of the tasks, the amount of tasks executed in a specific task duration (Figure 16) and the percentage of executing tasks with a certain task duration (Figure 17) remains similar to the previous version.

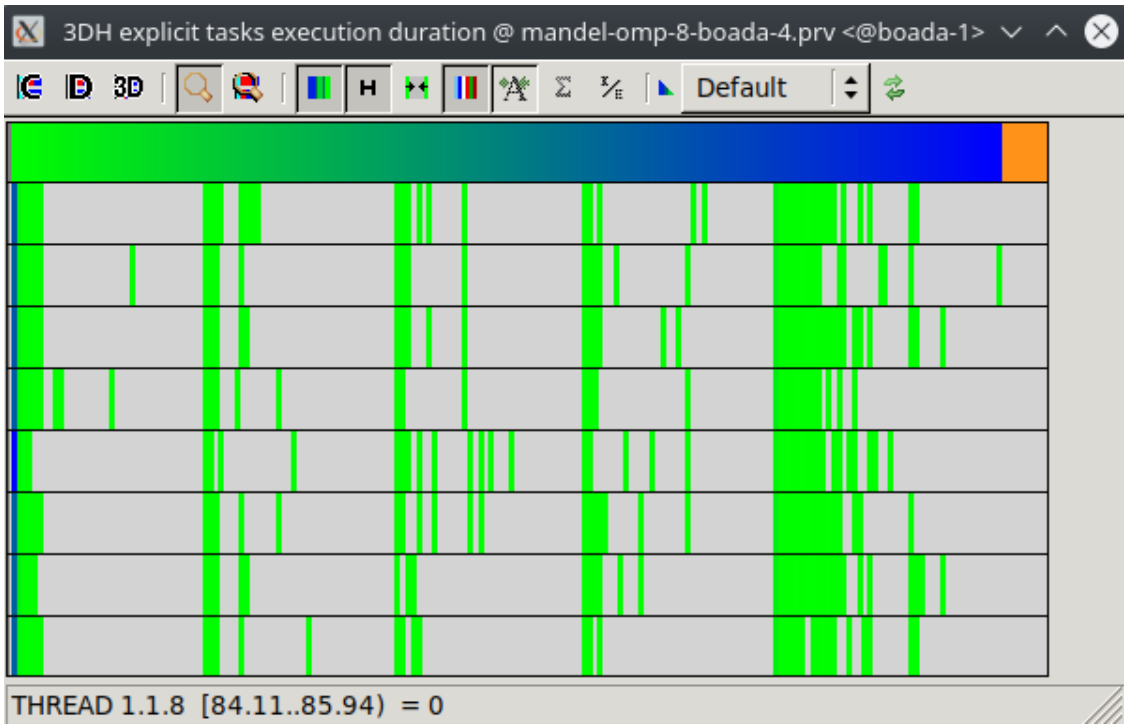


Figure 16: Histogram of explicit tasks execution duration (num bursts)

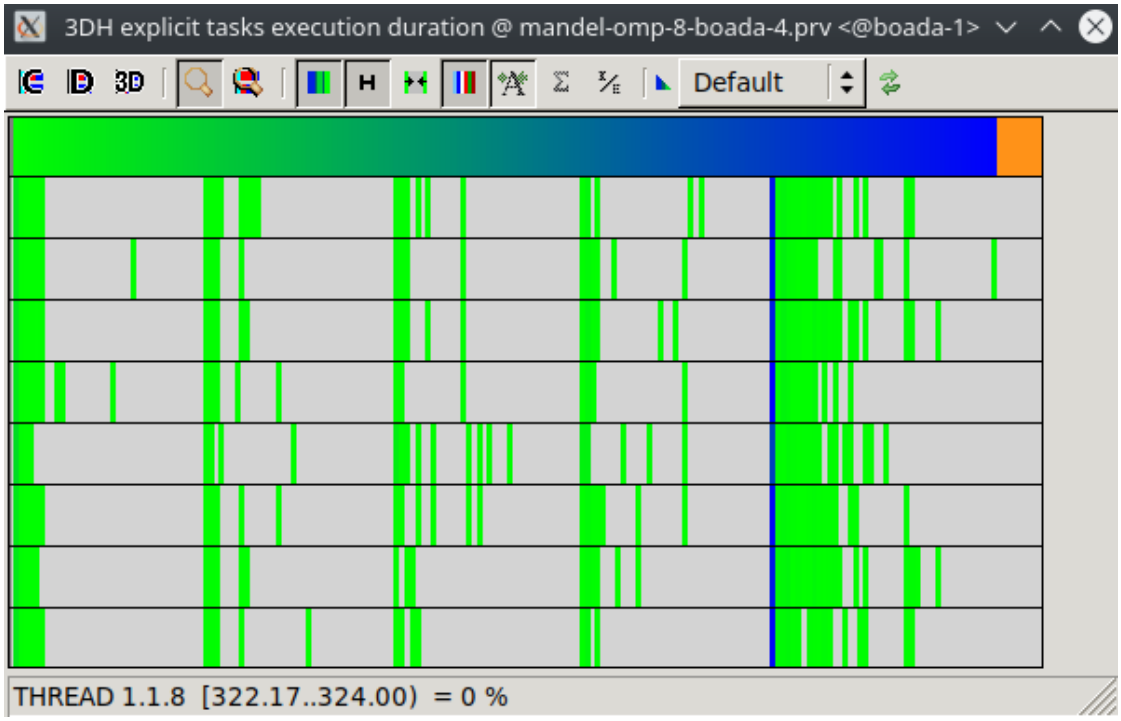
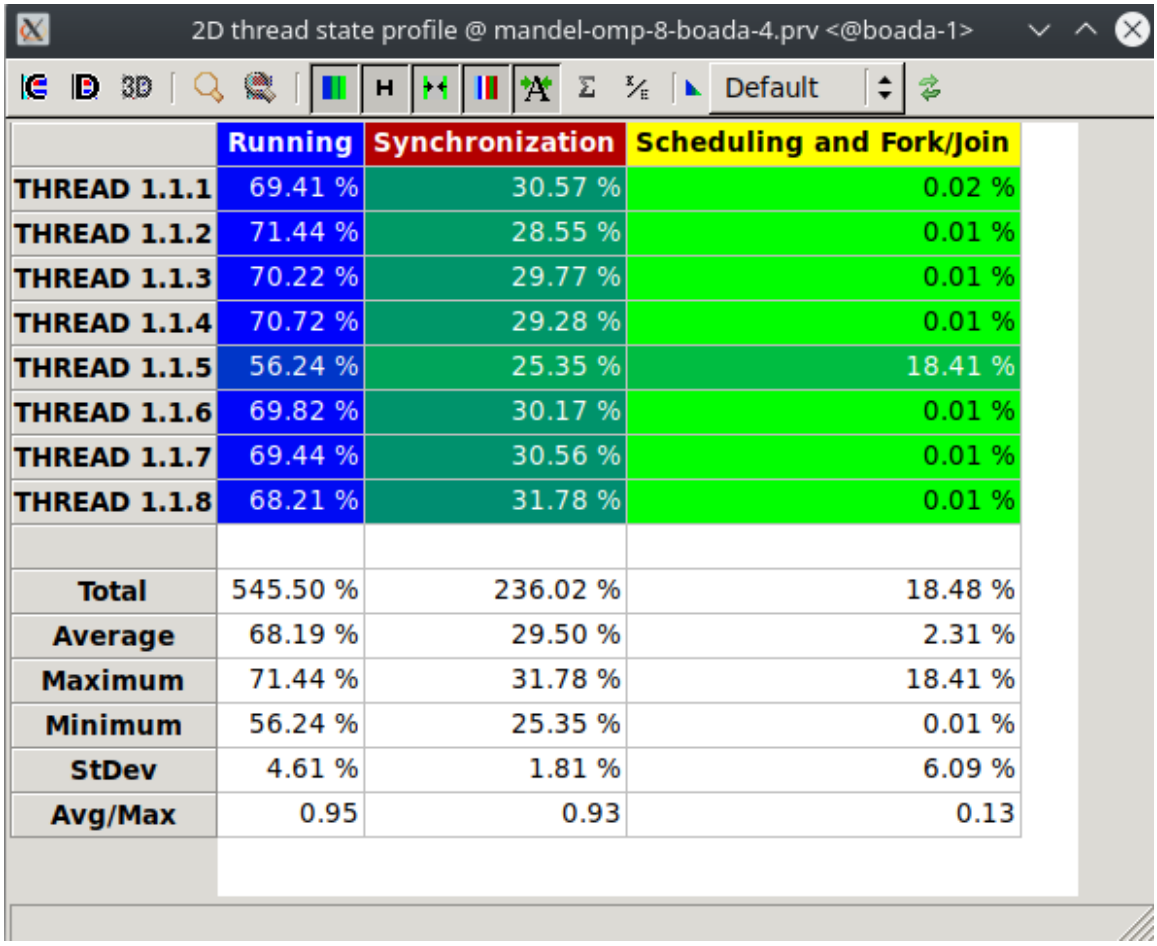


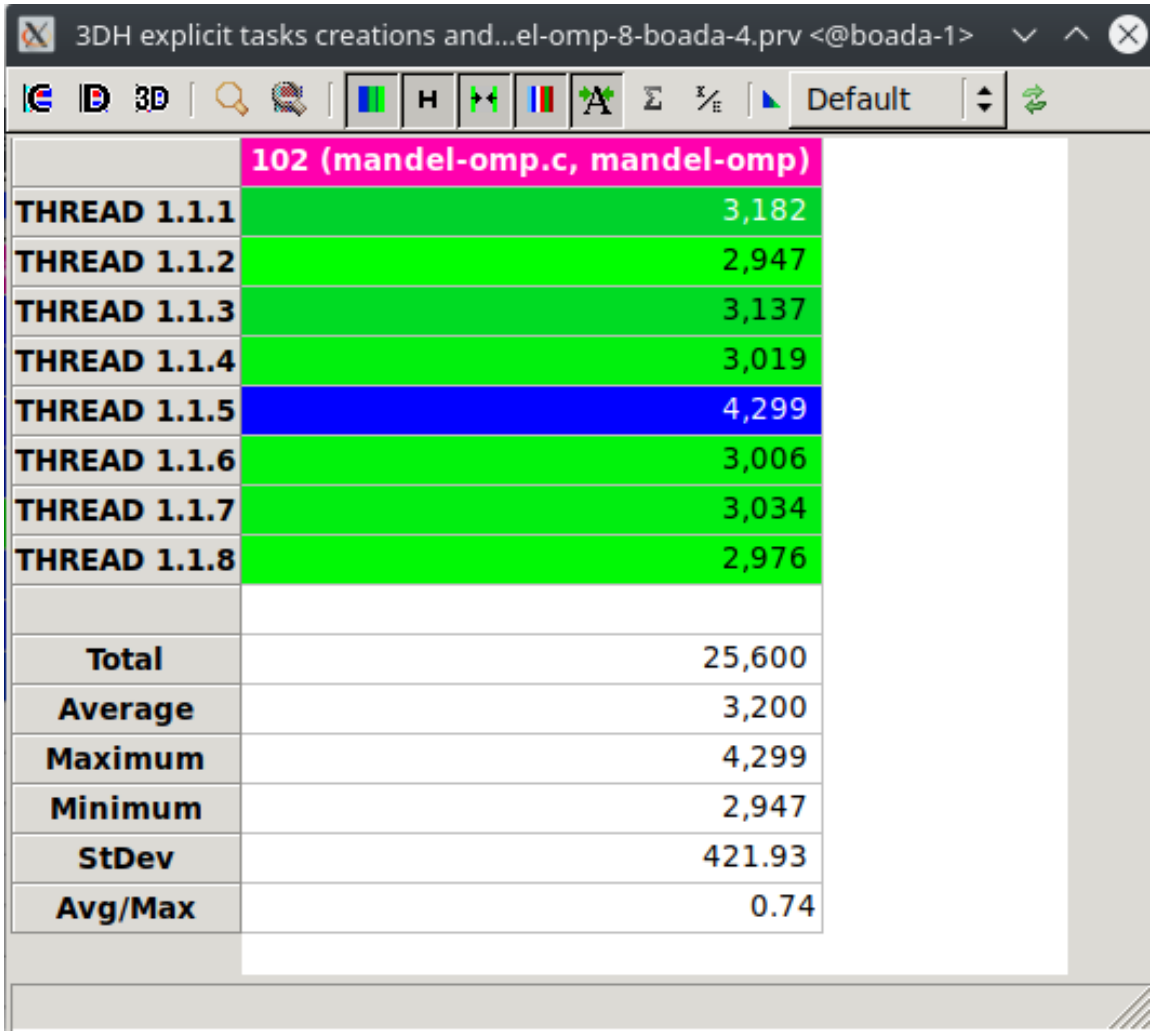
Figure 17: Histogram of explicit tasks execution duration (% time)

Unlike the previous version, in this one we can observe a significant decrease in synchronization time (Figure 18) and in the number of tasks (Figure 19), and that has an impact on the speed-up of the program.



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	69.41 %	30.57 %	0.02 %
THREAD 1.1.2	71.44 %	28.55 %	0.01 %
THREAD 1.1.3	70.22 %	29.77 %	0.01 %
THREAD 1.1.4	70.72 %	29.28 %	0.01 %
THREAD 1.1.5	56.24 %	25.35 %	18.41 %
THREAD 1.1.6	69.82 %	30.17 %	0.01 %
THREAD 1.1.7	69.44 %	30.56 %	0.01 %
THREAD 1.1.8	68.21 %	31.78 %	0.01 %
Total	545.50 %	236.02 %	18.48 %
Average	68.19 %	29.50 %	2.31 %
Maximum	71.44 %	31.78 %	18.41 %
Minimum	56.24 %	25.35 %	0.01 %
StDev	4.61 %	1.81 %	6.09 %
Avg/Max	0.95	0.93	0.13

Figure 18: Thread state profile



	102 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	3,182
THREAD 1.1.2	2,947
THREAD 1.1.3	3,137
THREAD 1.1.4	3,019
THREAD 1.1.5	4,299
THREAD 1.1.6	3,006
THREAD 1.1.7	3,034
THREAD 1.1.8	2,976
Total	25,600
Average	3,200
Maximum	4,299
Minimum	2,947
StDev	421.93
Avg/Max	0.74

Figure 19: Profile of explicit tasks creation and execution

If we use the group clause we can avoid unnecessary barriers, so we will decrease the time that threads lose in synchronization, and we will improve speedup (Figure 20).

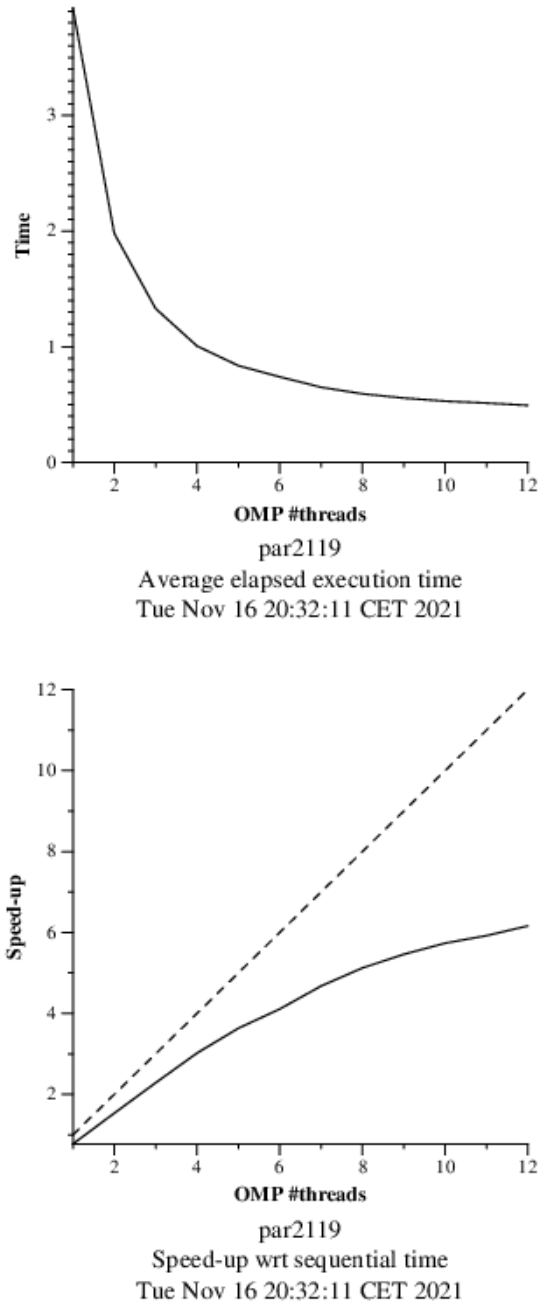


Figure 20: Speed-up for taskloop nogroup version

In this version, the efficiency improvement of this parallel execution is already noticeable (Figure 21): the synchronization time of the threads decreases (red) while the execution time of the tasks increases (blue). In this execution, thread 7 is the one who enters the single and creates the tasks.

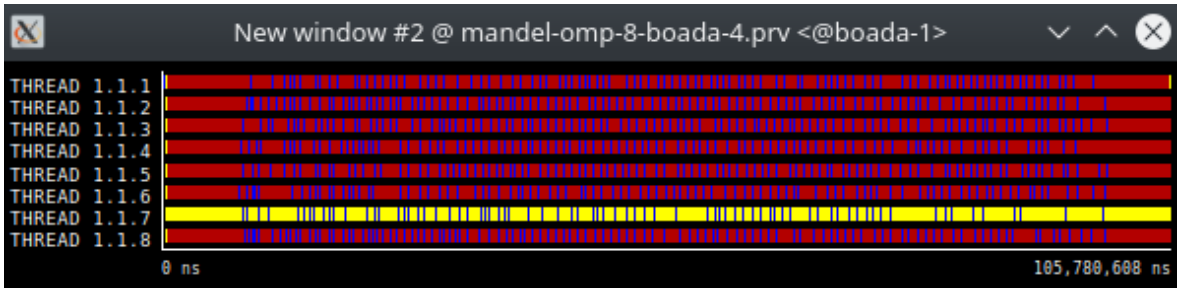
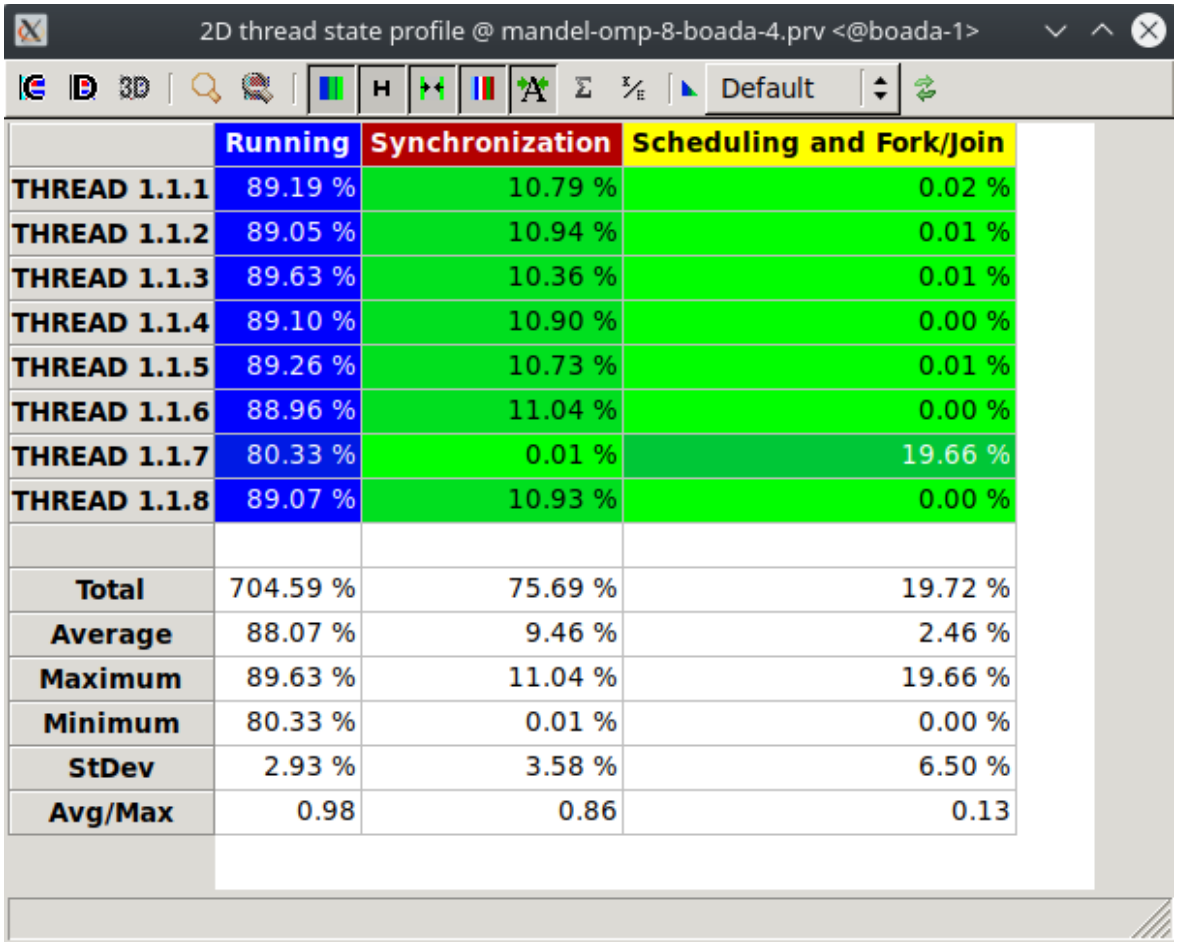


Figure 21: Timeline window v3



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	89.19 %	10.79 %	0.02 %
THREAD 1.1.2	89.05 %	10.94 %	0.01 %
THREAD 1.1.3	89.63 %	10.36 %	0.01 %
THREAD 1.1.4	89.10 %	10.90 %	0.00 %
THREAD 1.1.5	89.26 %	10.73 %	0.01 %
THREAD 1.1.6	88.96 %	11.04 %	0.00 %
THREAD 1.1.7	80.33 %	0.01 %	19.66 %
THREAD 1.1.8	89.07 %	10.93 %	0.00 %
Total	704.59 %	75.69 %	19.72 %
Average	88.07 %	9.46 %	2.46 %
Maximum	89.63 %	11.04 %	19.66 %
Minimum	80.33 %	0.01 %	0.00 %
StDev	2.93 %	3.58 %	6.50 %
Avg/Max	0.98	0.86	0.13

Figure 22: Thread state profile

With this version, we can further reduce the synchronization time of the threads and, therefore, the execution time.

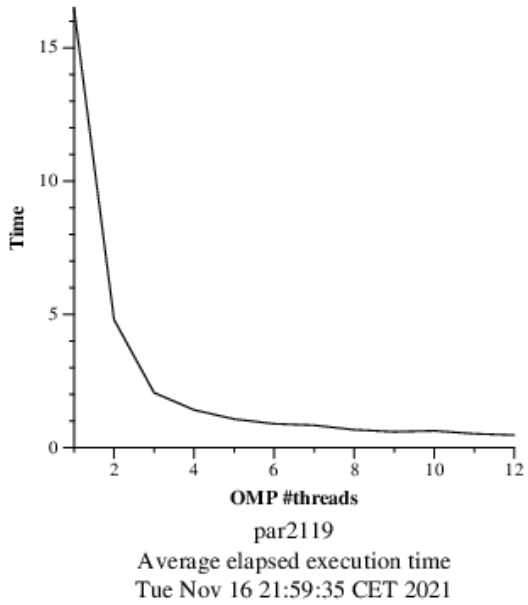
3. *Row* decomposition in OpenMP

For this latest version we will modify the code to parallelize each row:

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

Figure 23: last parallel version

With the row strategy we get the following speed-up (Figure 24):



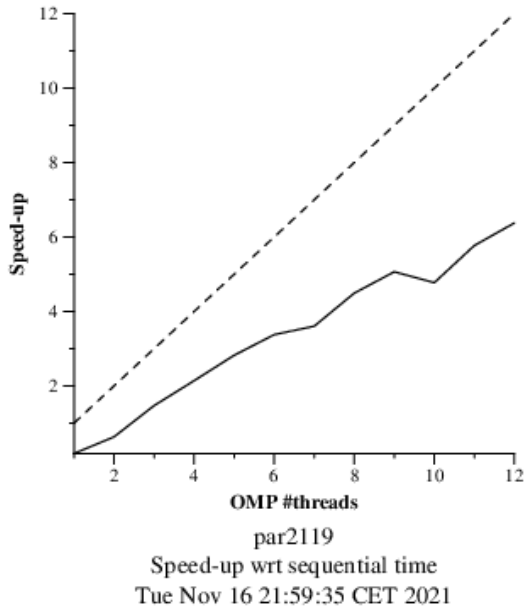


Figure 24: Speed-up for row strategy

Unlike the previous strategy, with this one the threads spend most of their time executing tasks.

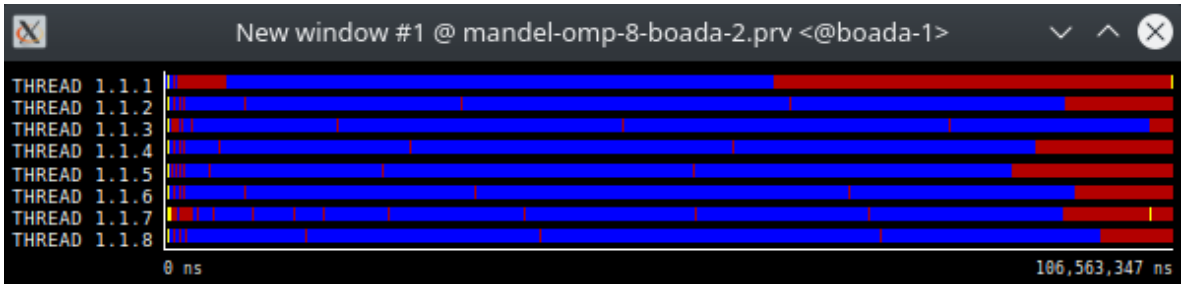
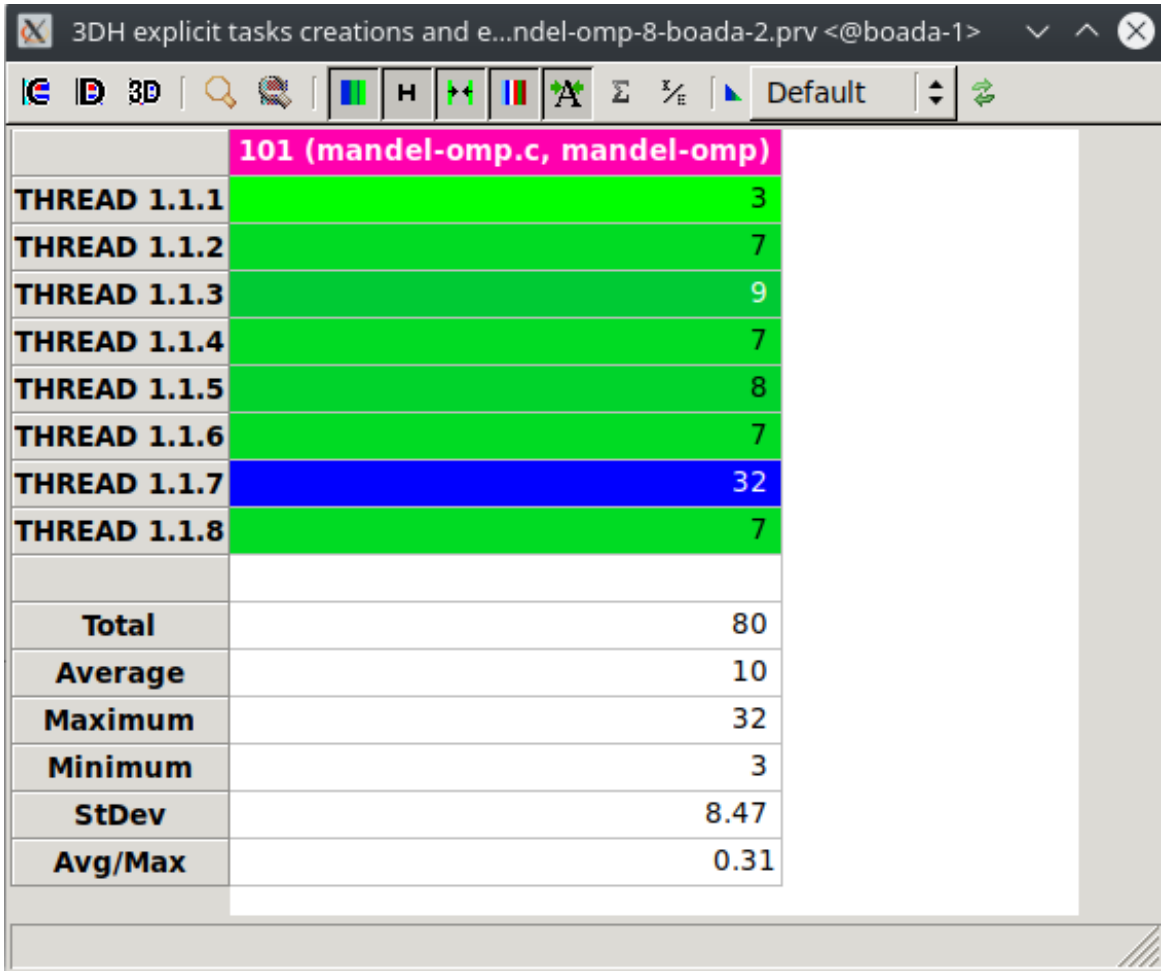


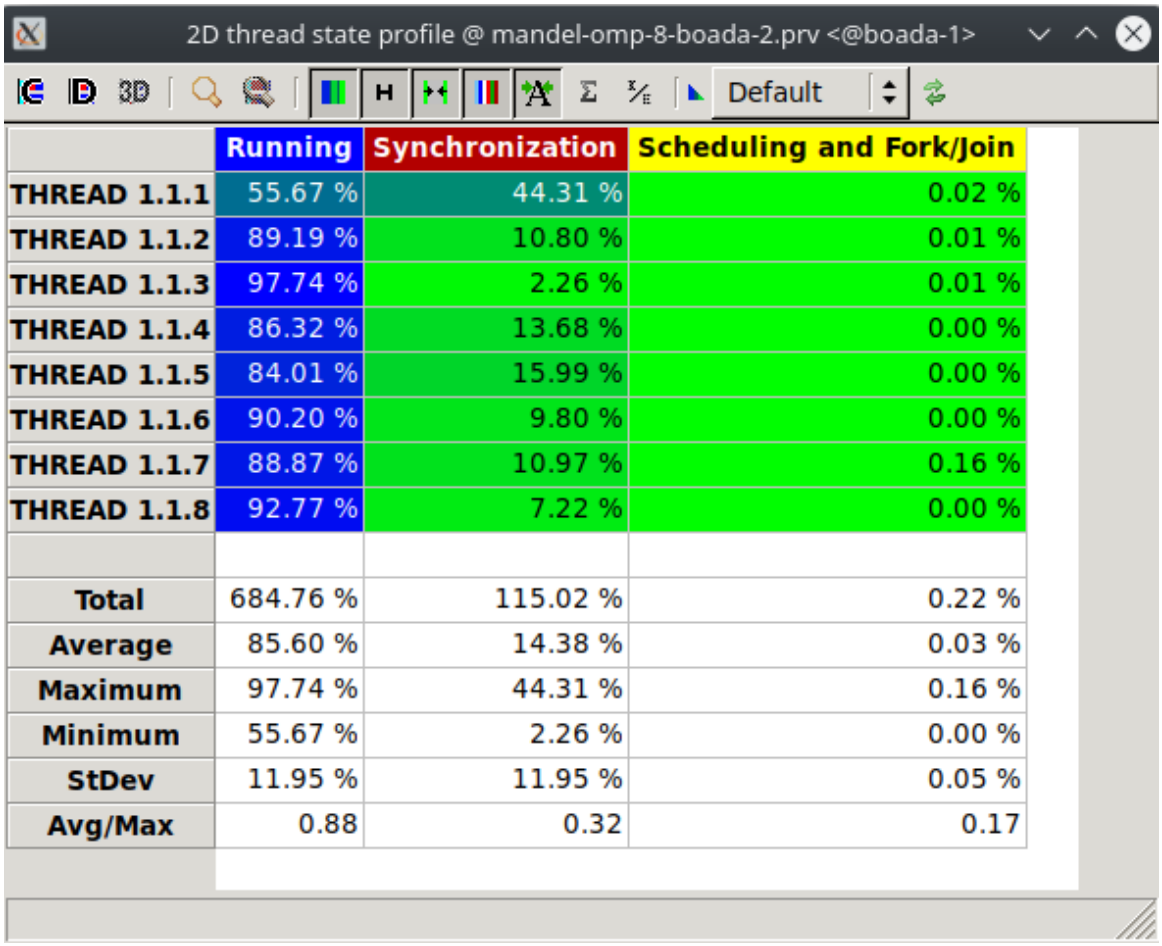
Figure 25: Timeline window v4

Figure 26: Explicit tasks function created and executed



	101 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	3
THREAD 1.1.2	7
THREAD 1.1.3	9
THREAD 1.1.4	7
THREAD 1.1.5	8
THREAD 1.1.6	7
THREAD 1.1.7	32
THREAD 1.1.8	7
Total	80
Average	10
Maximum	32
Minimum	3
StDev	8.47
Avg/Max	0.31

Figure 27: Profile of explicit tasks creation and execution



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	55.67 %	44.31 %	0.02 %
THREAD 1.1.2	89.19 %	10.80 %	0.01 %
THREAD 1.1.3	97.74 %	2.26 %	0.01 %
THREAD 1.1.4	86.32 %	13.68 %	0.00 %
THREAD 1.1.5	84.01 %	15.99 %	0.00 %
THREAD 1.1.6	90.20 %	9.80 %	0.00 %
THREAD 1.1.7	88.87 %	10.97 %	0.16 %
THREAD 1.1.8	92.77 %	7.22 %	0.00 %
Total	684.76 %	115.02 %	0.22 %
Average	85.60 %	14.38 %	0.03 %
Maximum	97.74 %	44.31 %	0.16 %
Minimum	55.67 %	2.26 %	0.00 %
StDev	11.95 %	11.95 %	0.05 %
Avg/Max	0.88	0.32	0.17

Figure 28: Thread state profile

One of the most notable differences between the two strategies is the amount of tasks created and executed by the threads, while the execution time and efficiency are similar.

4. Optional

For this section, in both versions, we add to the algorithm:

```
#pragma omp taskloop num_tasks(user_param)
```

Where the number of tasks is passed with the option -u and the algorithm stores the number in the variable *user_param*.

4.1 Point version

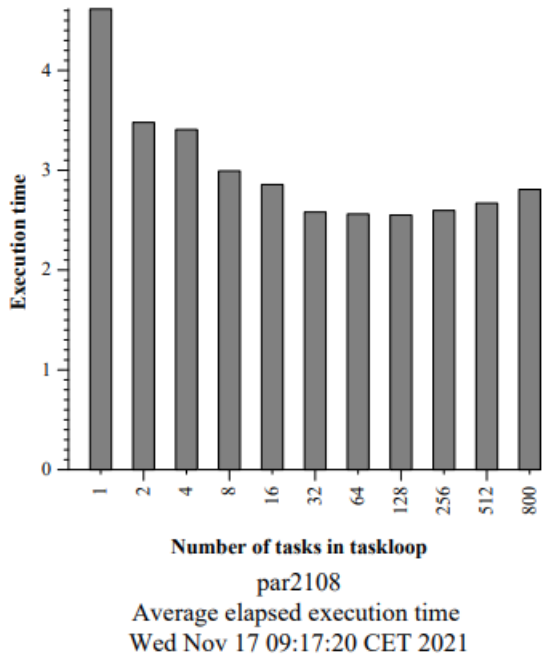


Figure 29: Execution time depending on the number of tasks for the point version with 2 threads

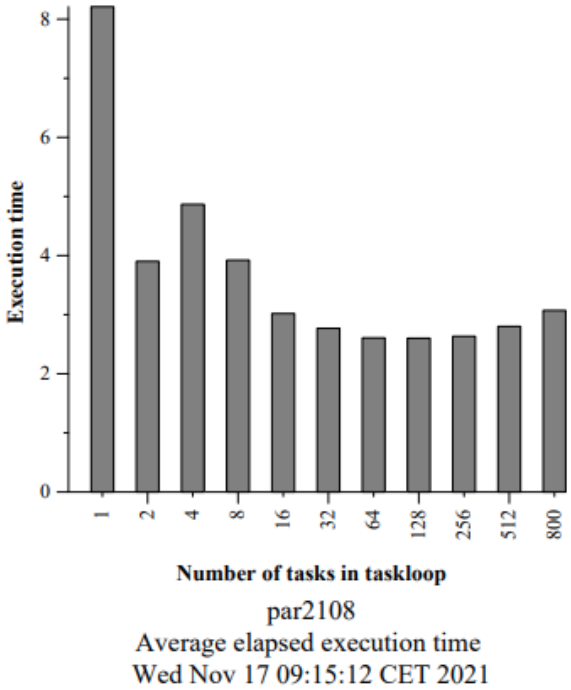


Figure 30: Execution time depending on the number of tasks for the point version with 4 threads

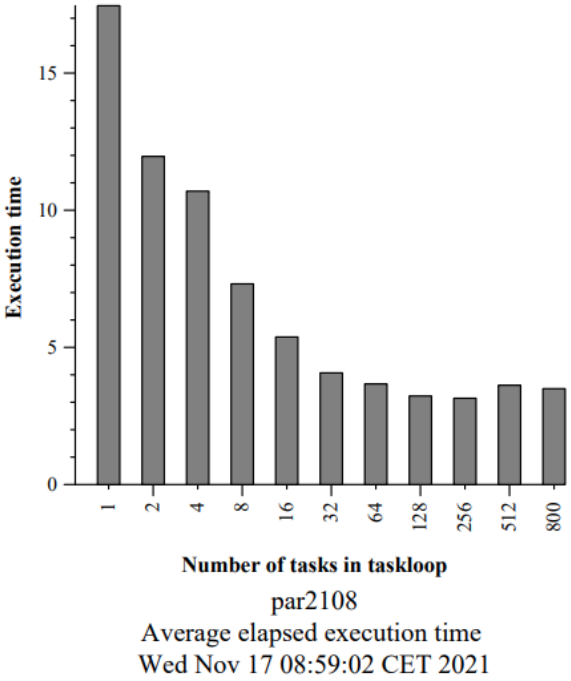


Figure 31: Execution time depending on the number of tasks for the point version with 8 threads

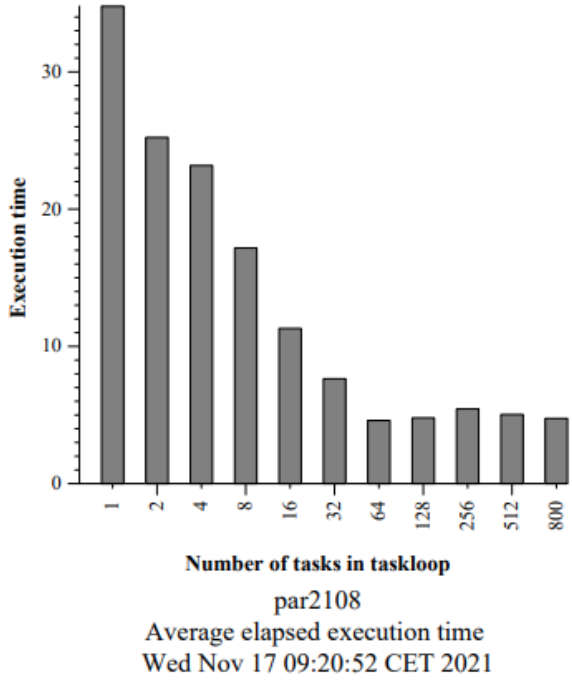


Figure 32: Execution time depending on the number of tasks for the point version with 16 threads

As we can see above the optimal number of tasks depends on the number of threads we have. We tried with 2, 4, 8 and 16 threads.

In the following table we conclude the optimal number of tasks according to the number of threads:

Threads	*num_tasks*
2	128
4	128
8	256
16	64

4.2 Row version

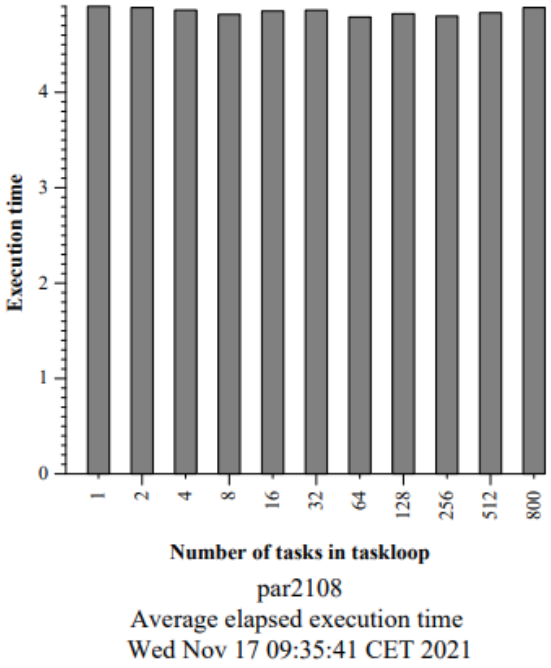


Figure 33: Execution time depending on the number of tasks for the row version with 2 threads

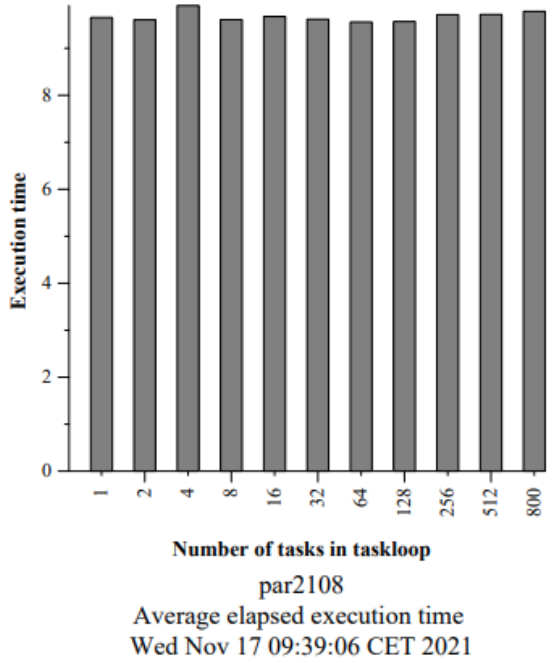


Figure 34: Execution time depending on the number of tasks for the row version with 4 threads

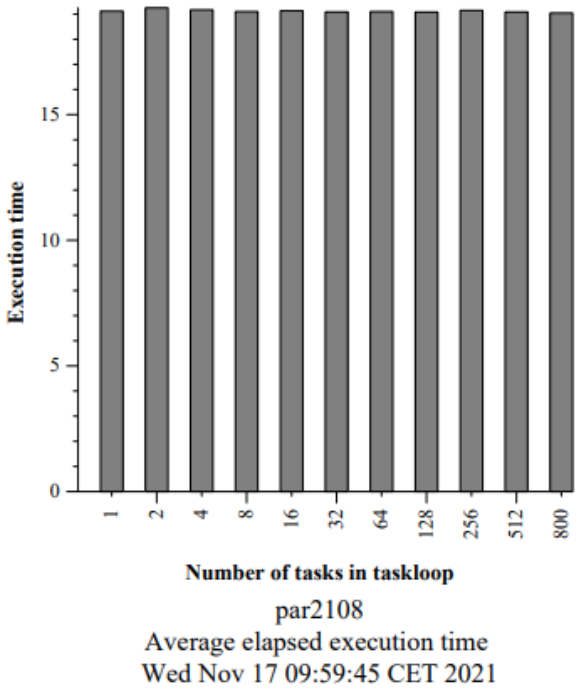


Figure 35: Execution time depending on the number of tasks for the row version with 8 threads

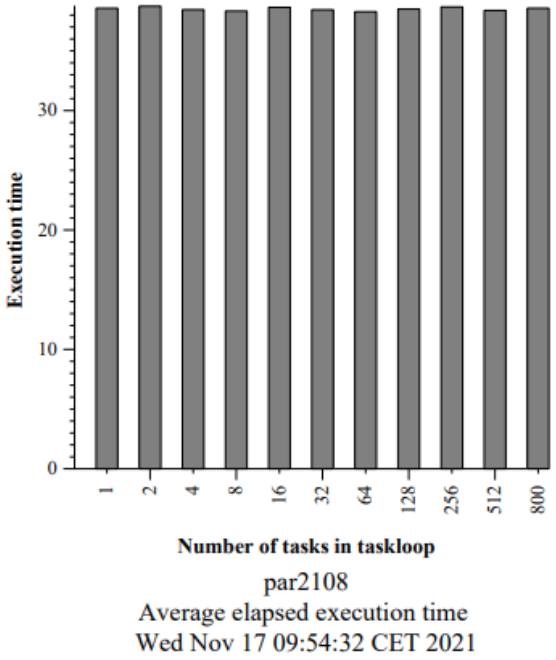


Figure 36: Execution time depending on the number of tasks for the row version with 16 threads

Notice that for the row version we have that the number of threads that we use not affect as the point version in the execution time depending on the number of tasks.

To conclude, we can say that the task granularity does not affect in the same way the point and the row versions, and we want to remark the importance of this tools when we want to extract conclusions to improve the performance of our algorithms, because with this tools we can obtain the best performance for our algorithm in any environment (any machine).