Lab3: Iterative task decomposition with OpenMP: the computation of the Mandelbrot set
Q2 2021-22

Integrantes:
Pol Pérez Castillo - par2116
Cristian Sánchez Estapé - par2118

Fecha de entrega: 19-04-2022

Index

Introduction

In this laboratory we work and analyze the computation of the Mandelbrot set. The first step will be to carry out a detailed analysis of the different decomposition options in tasks of this program, dig deeply into row strategy and point strategy and noting which part of the computation gives serial execution problems (options -d and -h). The second part, once we have obtained all the dependency graphs, we will go on to implement these same strategies, explained in the first part, in OpenMP to obtain all the data that we need (helping us with the use of Paraver) and thus be able to investigate and compare which strategy is more efficient than another.

Task decomposition analysis with Tareador

Row strategy

2. Which are the two most important characteristics for the task graph that is generated?

On the one hand, the fact that the innermost row tasks are the ones that take the longest time to execute, which is also given by a procedural increase and decrease in size (from left to right, we can observe that from task 1 to 4, each one increases its size in contrast to its previous one, and from 5 to 8 it decreases).. On the other hand, the fact that all row loops can be executed in parallel (even though the critical path, which is determined by RowTask 4, will crucially determine the execution time of the program).

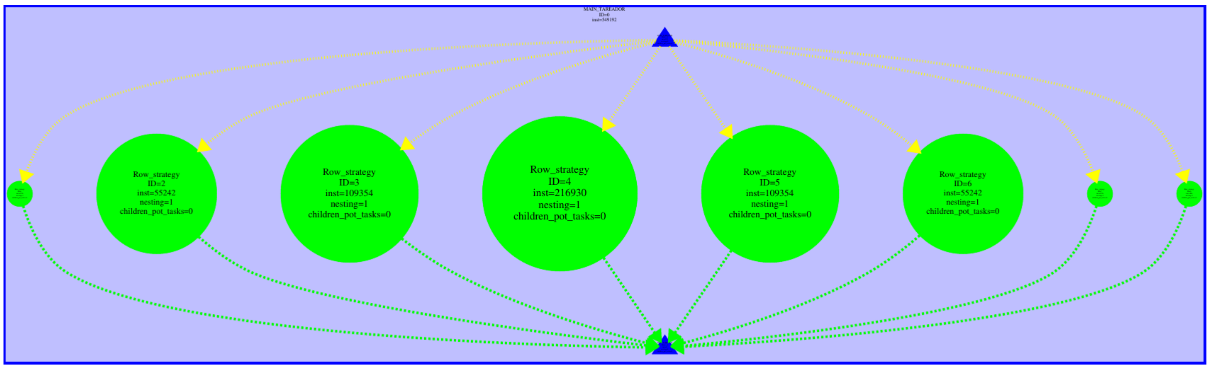


Figure 1: row strategy TDG without any execution option

3. Again, which are the two most important characteristics for the task graph that is generated?
Which part of the code is making the big difference with the previous case? How will you protect this section of code in the parallel OpenMP code that you will program in the next sessions?

On the one hand, the fact (already commented) that the size of each task increases and decreases throughout the execution of the program. On the other hand, the fact that, in contrast to what happened in the previous case, even if we wanted to execute the program in parallel, it would have no effect whatsoever in the end result, given that the observed dependencies forces the program to execute sequentially.
The part of the code that is causing this is the one related to the *output2display*, which depends directly on the immediate, and thus forces us to wait for the end of the visualization process to continue the execution.
The code should be protected the following way: #pragma omp critical

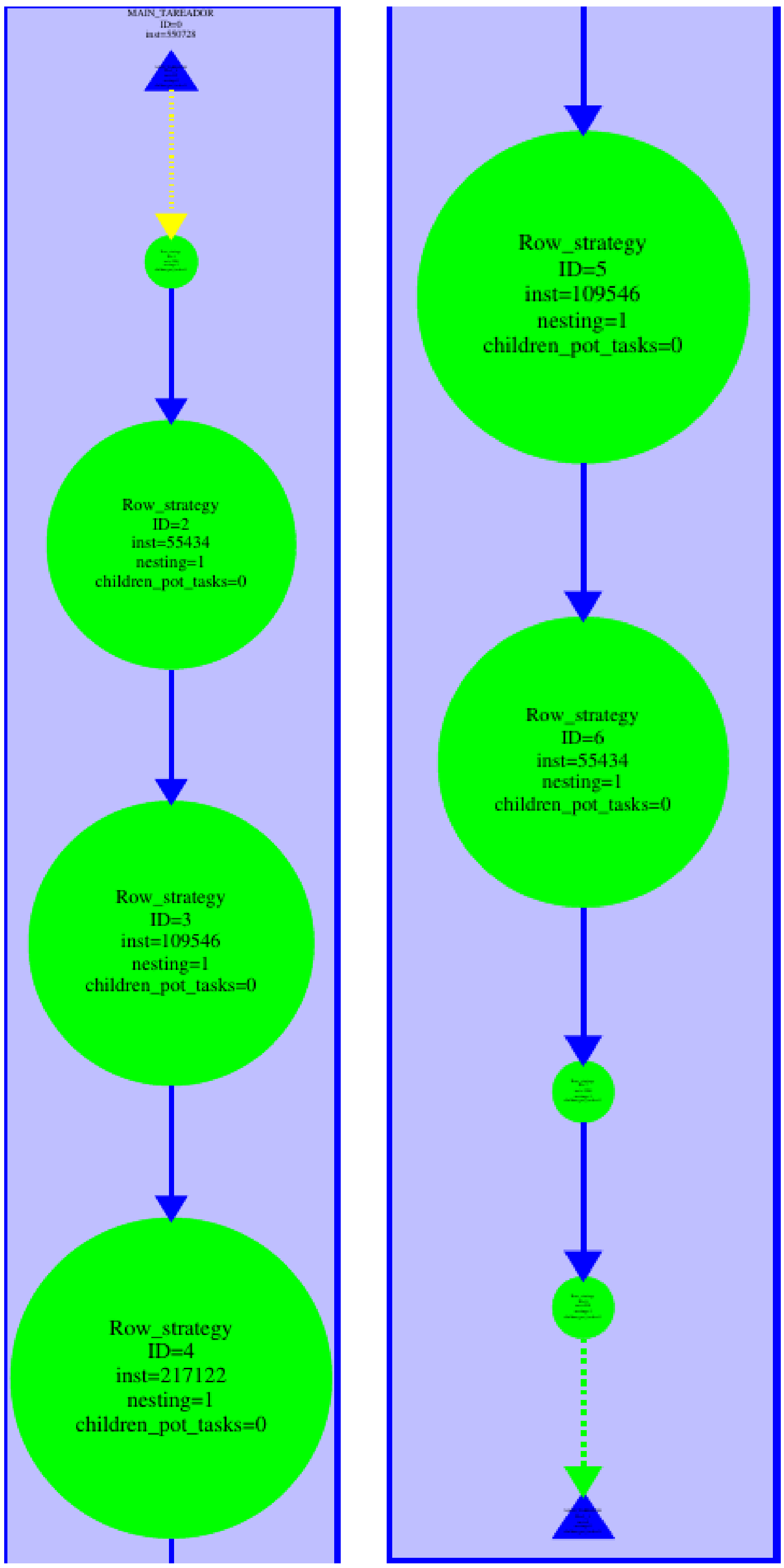


Figure 2: row strategy TDG with -d execution option (display)

4. What does each chain of tasks in the task graph represents? Which part of the code is making the big difference with the two previous cases? How will you protect this section of code in the parallel OpenMP code that you will program in the next sessions?

The part of the code that is crystallizing in this feature is the *output2histogram* conditional, which registers how many times a task has computed its number in the *same number of iterations*. Thus, this condition marks the execution of the program and causes these dependencies to materialize.
The code should be protected the following way: #pragma omp atomic

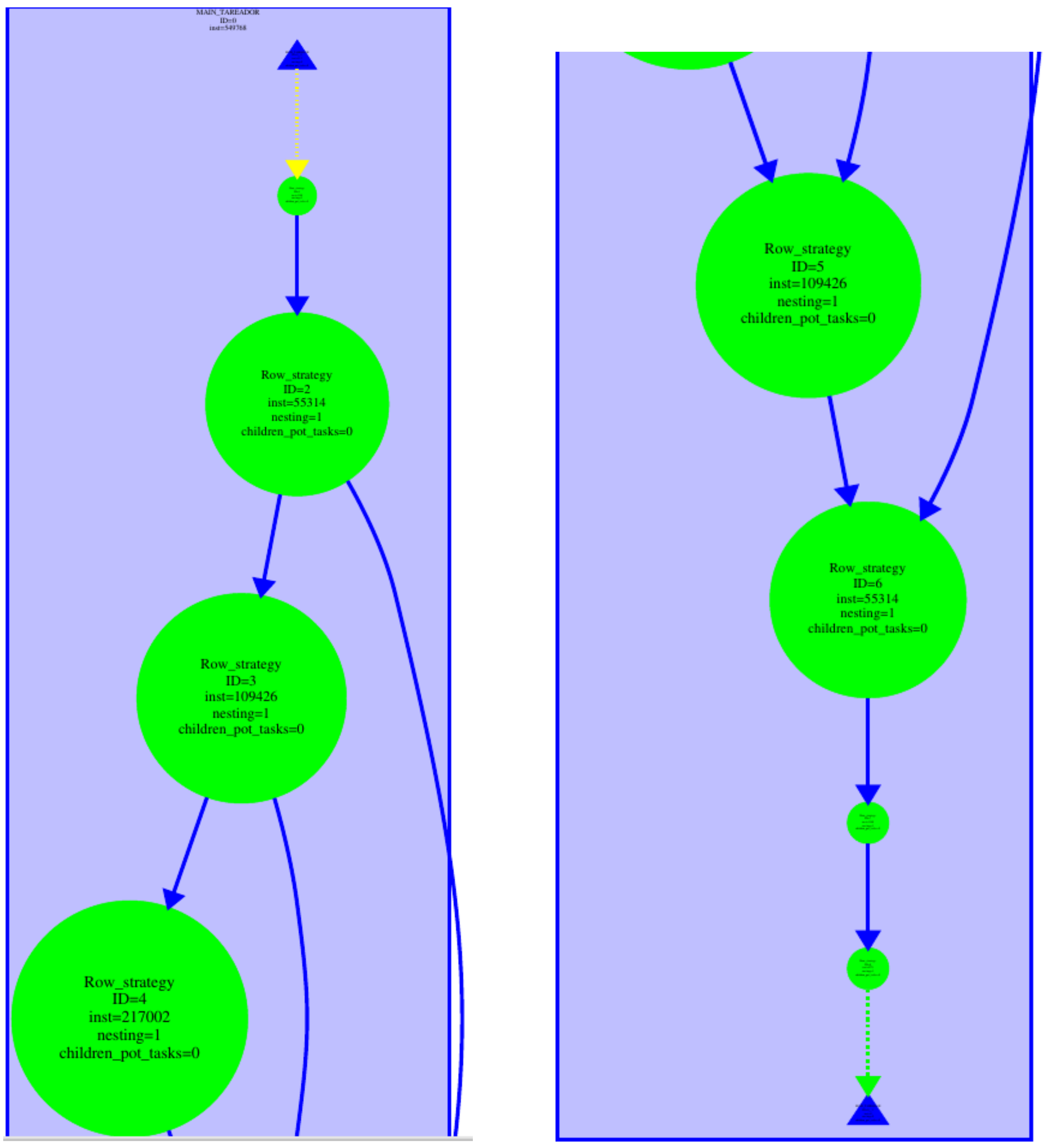


Figure 3: row strategy TDG with -h execution option (histogram)

Point strategy

Are conclusions the same? Which is the main change that you observe with respect to the Row strategy?

With point strategy, the reached conclusions are the same as before: the dependencies are virtually identical to those of the *row* strategy (as it can be seen in figures 1 to 6, each compared according to the type of execution). However, there are two important differences to be considered: on the one hand, the granularity of the tasks, which augment considerably in *point* strategy (which makes sense since the instrumentation is now done for each iteration of *col*, instead of *groups* of iterations of *col*, which was the main feature of the *row* strategy); on the other hand, as it can bee seen in figure 6, the critical path of the generated TDG is characterized by the fact that the biggest tasks (in terms of computational cost) are executed sequentially, which translates into into a crucially important feature of the execution (given that, when parallelizing, there won't be any way to evade such costly dependencies).

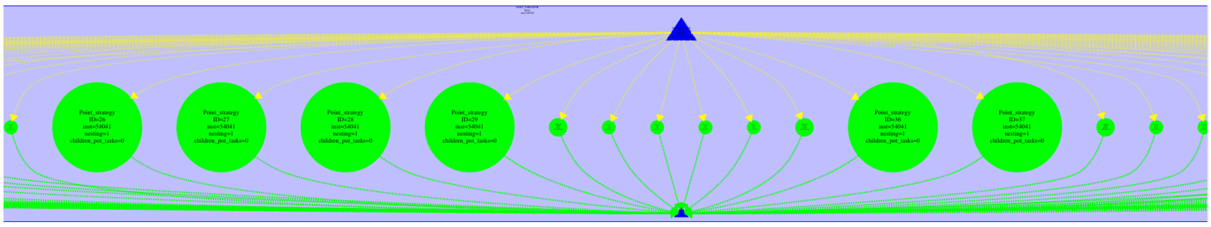


Figure 4: point strategy TDG without any execution option

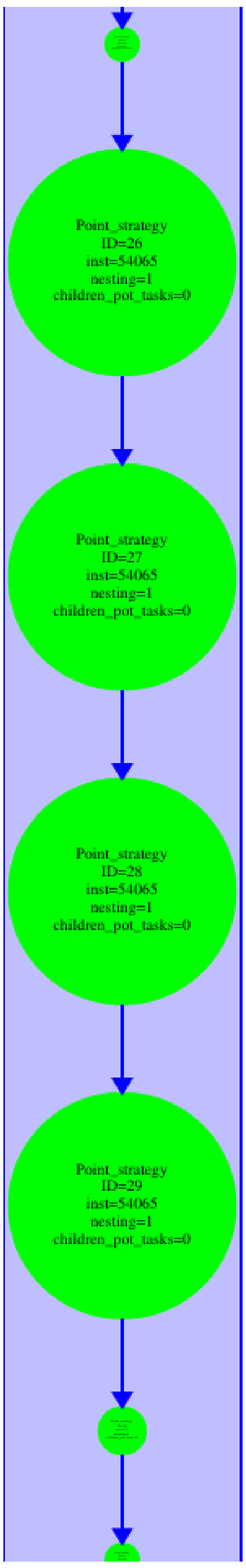


Figure 5: point strategy TDG with -d execution option (display)

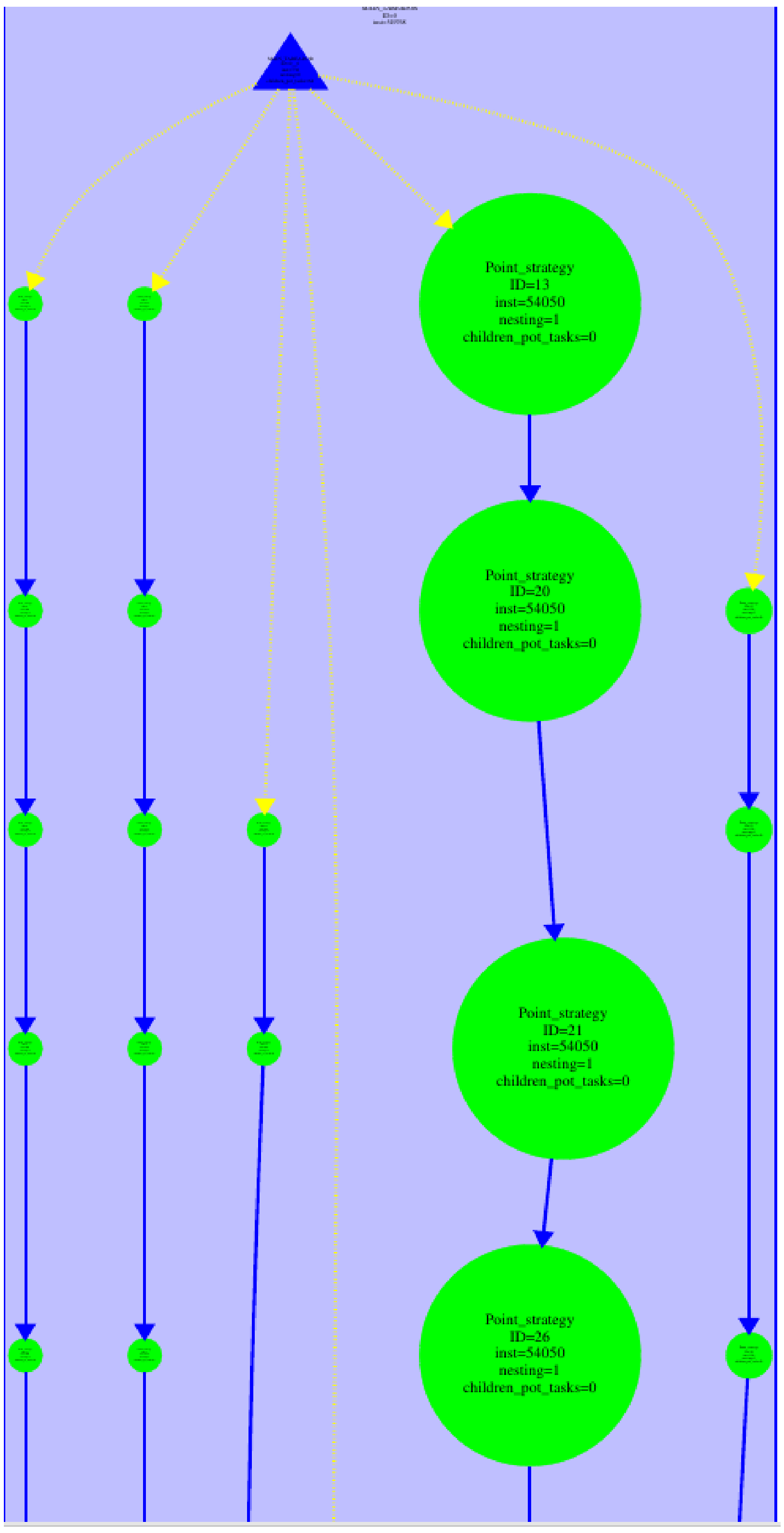


Figure 6: point strategy TDG with -h execution option (histogram)

Implementing Task Decompositions in *OpenMP*

Point strategy implementation using *task*

3. Is the speed–up appropriate? [...] Again, is the scalability appropriate?

Given the fact that the script executes the parallelized binary with the histogram generation option, both the elapsed time as well as the speed-up find themselves conditioned by their respective dependencies (which can be seen in figure 6). With this, it becomes self-evident that, when executing with 5 to 6 threads, the execution stalls and, thus, both the elapsed time and the speed-up stagnate (figure 7 and 8).

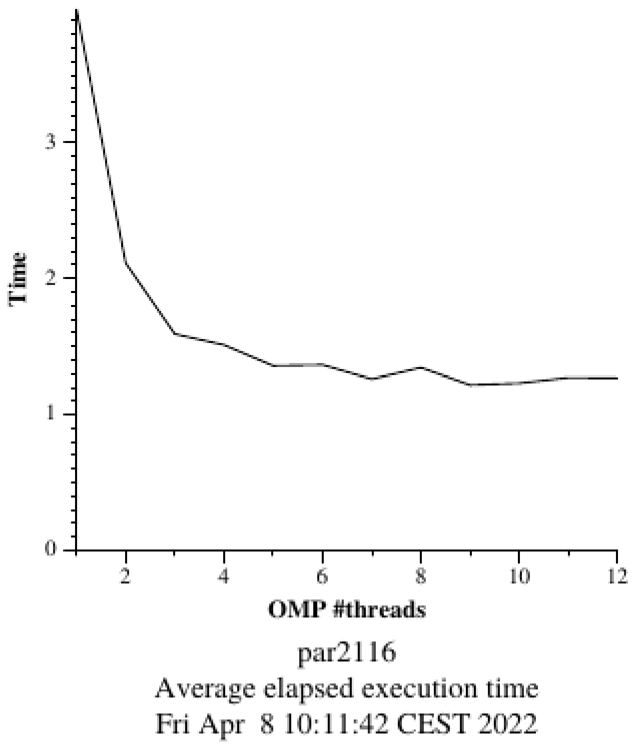


Figure 7: elapsed time of the point strategy execution

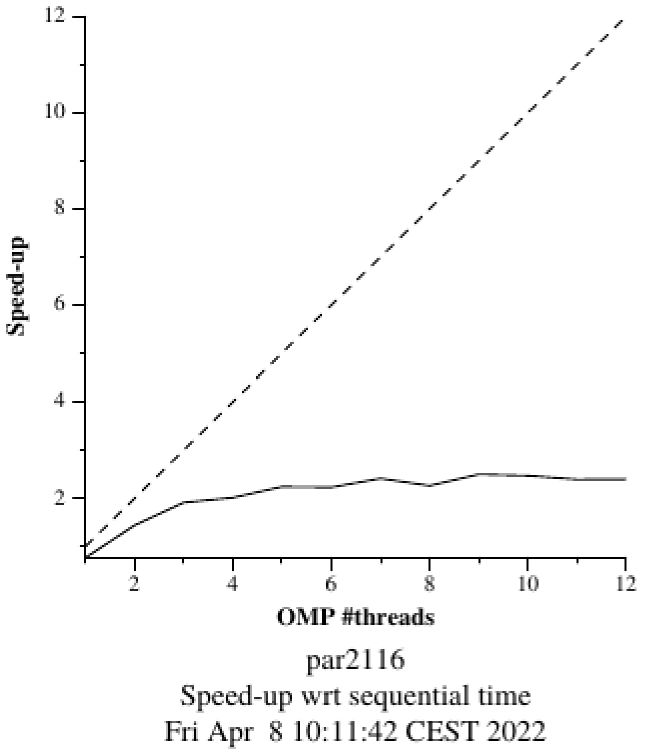


Figure 8: speed-up of the point strategy execution

4.b) Which threads are creating and executing them? How many tasks are created/executed?

As shown in figure 9, the thread that creates the explicit tasks is nº 7, and the remaining ones are those that execute said tasks. In regards to the number of tasks, for each point of the Mandelbrot set, a task is created and executed, as defined by the directive *#pragma omp task firstprivate(row,col)* defined in the innermost loop.

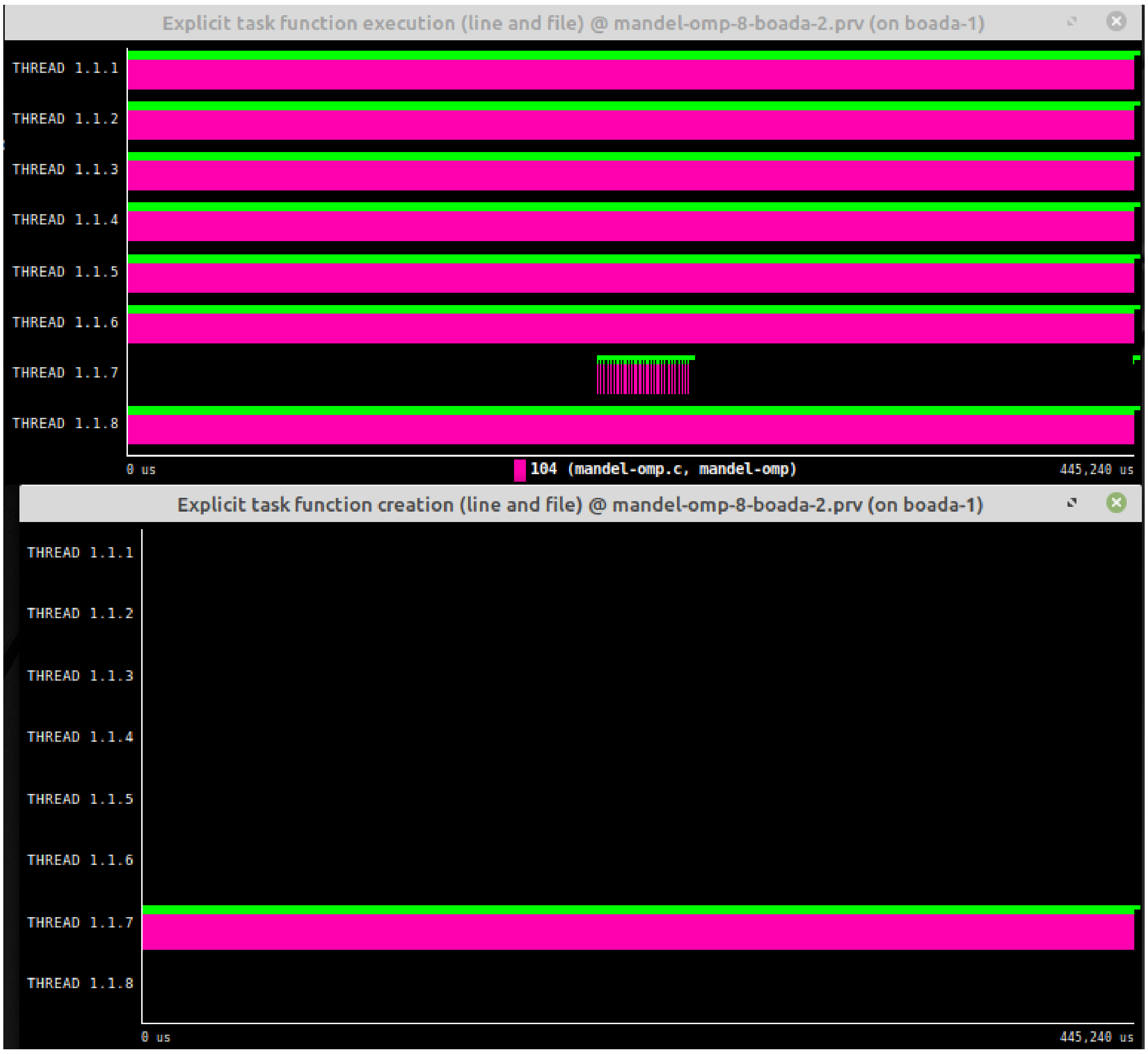
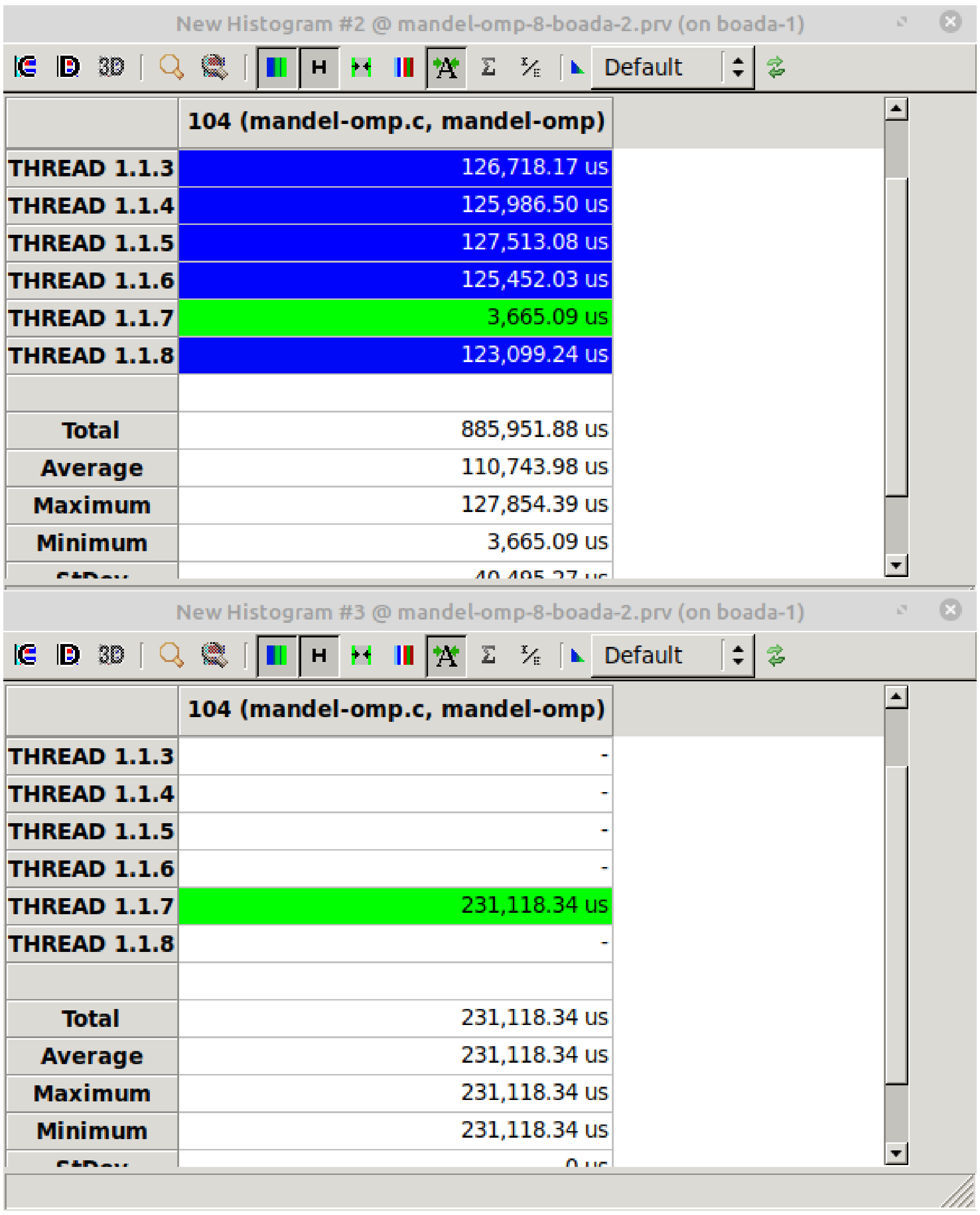


Figure 9: Paraver visualization of the threads creating and executing explicit tasks

Is the load well balanced (both in terms of number of tasks and in terms of toral time executing tasks)? Which of the two is actually relevant?

As it can be noted in the following histograms, the program takes more time in the creation of tasks than in its execution, which means that the overall time is affected by such feature. Also, given the fact that the creation of tasks virtually duplicates their time of execution (which means that, indeed, the creation of tasks is considerably more relevant), the load is not well balanced (there remains space to check if either the presence of more threads or the distribution of the creation of tasks themselves would decrease the overall execution time of the program).


New Histogram #2 @ mandel-omp-8-boada-2.prv (on boada-1)
Default

	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.3	126,718.17 us
THREAD 1.1.4	125,986.50 us
THREAD 1.1.5	127,513.08 us
THREAD 1.1.6	125,452.03 us
THREAD 1.1.7	3,665.09 us
THREAD 1.1.8	123,099.24 us
Total	885,951.88 us
Average	110,743.98 us
Maximum	127,854.39 us
Minimum	3,665.09 us

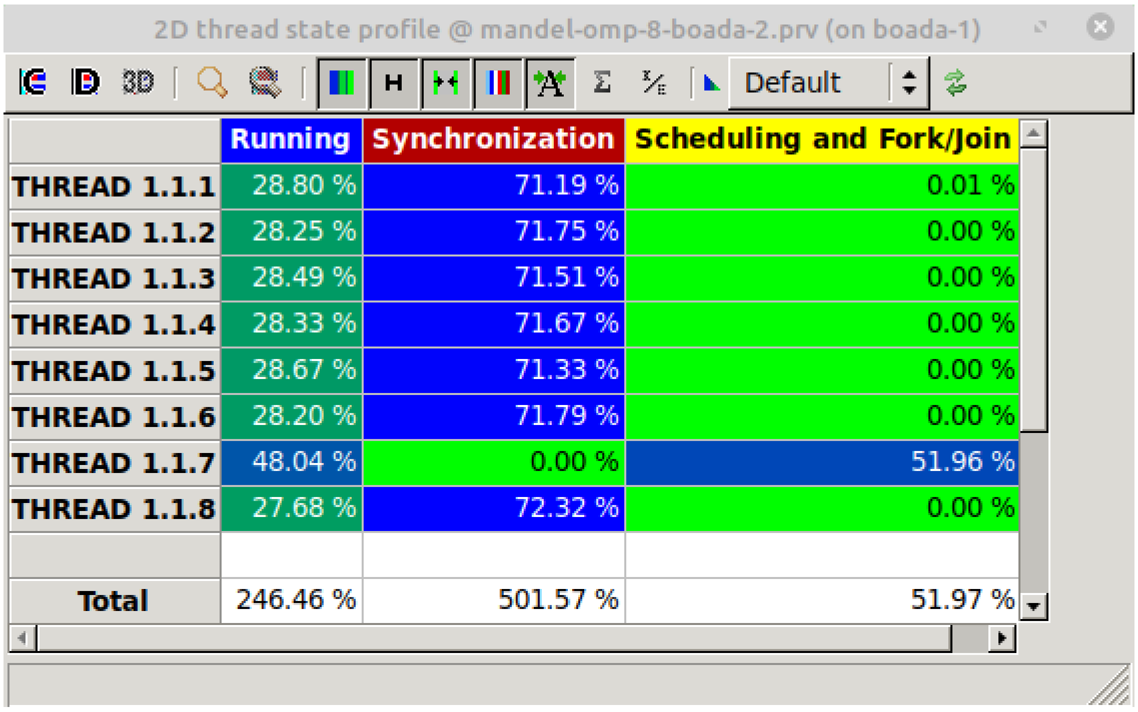
New Histogram #3 @ mandel-omp-8-boada-2.prv (on boada-1)
Default

	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	231,118.34 us
THREAD 1.1.8	-
Total	231,118.34 us
Average	231,118.34 us
Maximum	231,118.34 us
Minimum	231,118.34 us

Figure 10: Histograms of the threads creating and executing explicit tasks

4.d) In summary, is granularity of the tasks appropriate? Would it be easy to change?

As it can be observed, most of the execution time of the program is centered around the synchronization between tasks. Given that *point strategy* is based upon the creation of tasks for every present iteration in the double loop *row-col*, said time execution is derived from the enormous granularity of the strategy itself. Therefore, it can be said that it is not really appropriate, given the distribution of its time of execution. In regards to the difficulty of changing it, it could be easily achieved if we just reimagined what we consider as a task (*row strategy* itself is an example of it).


2D thread state profile @ mandel-omp-8-boada-2.prv (on boada-1)
Default

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	28.80 %	71.19 %	0.01 %
THREAD 1.1.2	28.25 %	71.75 %	0.00 %
THREAD 1.1.3	28.49 %	71.51 %	0.00 %
THREAD 1.1.4	28.33 %	71.67 %	0.00 %
THREAD 1.1.5	28.67 %	71.33 %	0.00 %
THREAD 1.1.6	28.20 %	71.79 %	0.00 %
THREAD 1.1.7	48.04 %	0.00 %	51.96 %
THREAD 1.1.8	27.68 %	72.32 %	0.00 %
Total	246.46 %	501.57 %	51.97 %

Figure 11: Histogram of the threads's state related with synchronization and task scheduling

Point strategy with granularity control using *taskloop*

2. Is the speedup appropriate? [...] Is the version with taskloop performing better than the last version based on the use of task?

Even without specifying exactly which clause should *taskloop* use (*num_tasks* or *grainsize*), the obtained speed-up is considerably greater (1 second less with the execution of 8 threads) than without granularity control. Moreover, given that we have less granularity, the execution time has been reduced (since the number of synchronizations has also decreased).
It can be also noted that, even though both figures 12 and 13 point towards a stall both in terms of execution time and speed-up, there's little bit of room for improvement (note the tendency of the seed-up to keep increasing, even if it is not much in general terms).

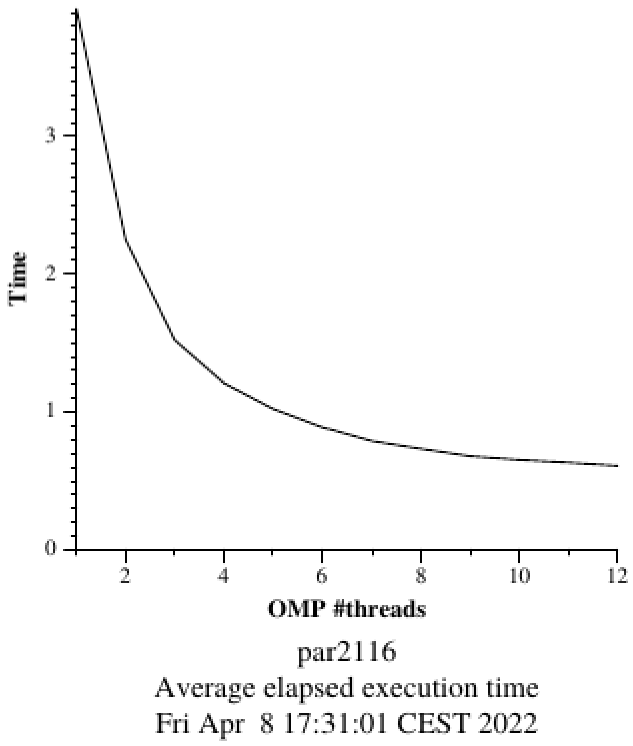


Figure 12: elapsed time of the point strategy execution using taskloop

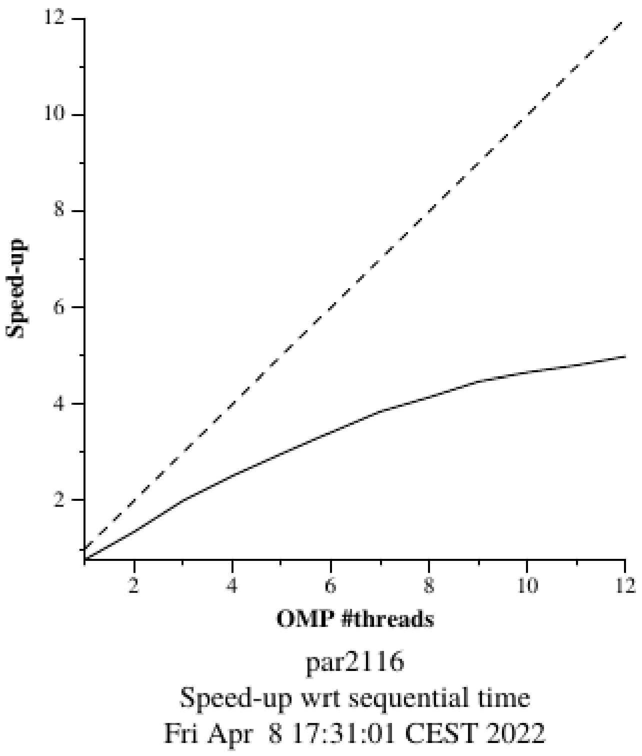


Figure 13: speed-up of the point strategy execution using taskloop

3.b) Which threads are creating and executing them? How many tasks are created/executed?

As shown in figure 14, the thread that creates the explicit tasks is nº 8, and the remaining ones are those that execute said tasks. In regards to the number of tasks, figures 15 and 16 show both the number of tasks created and executed as they've been defined by the directive *#pragma omp taskloop firstprivate(row)* defined in the external loop.

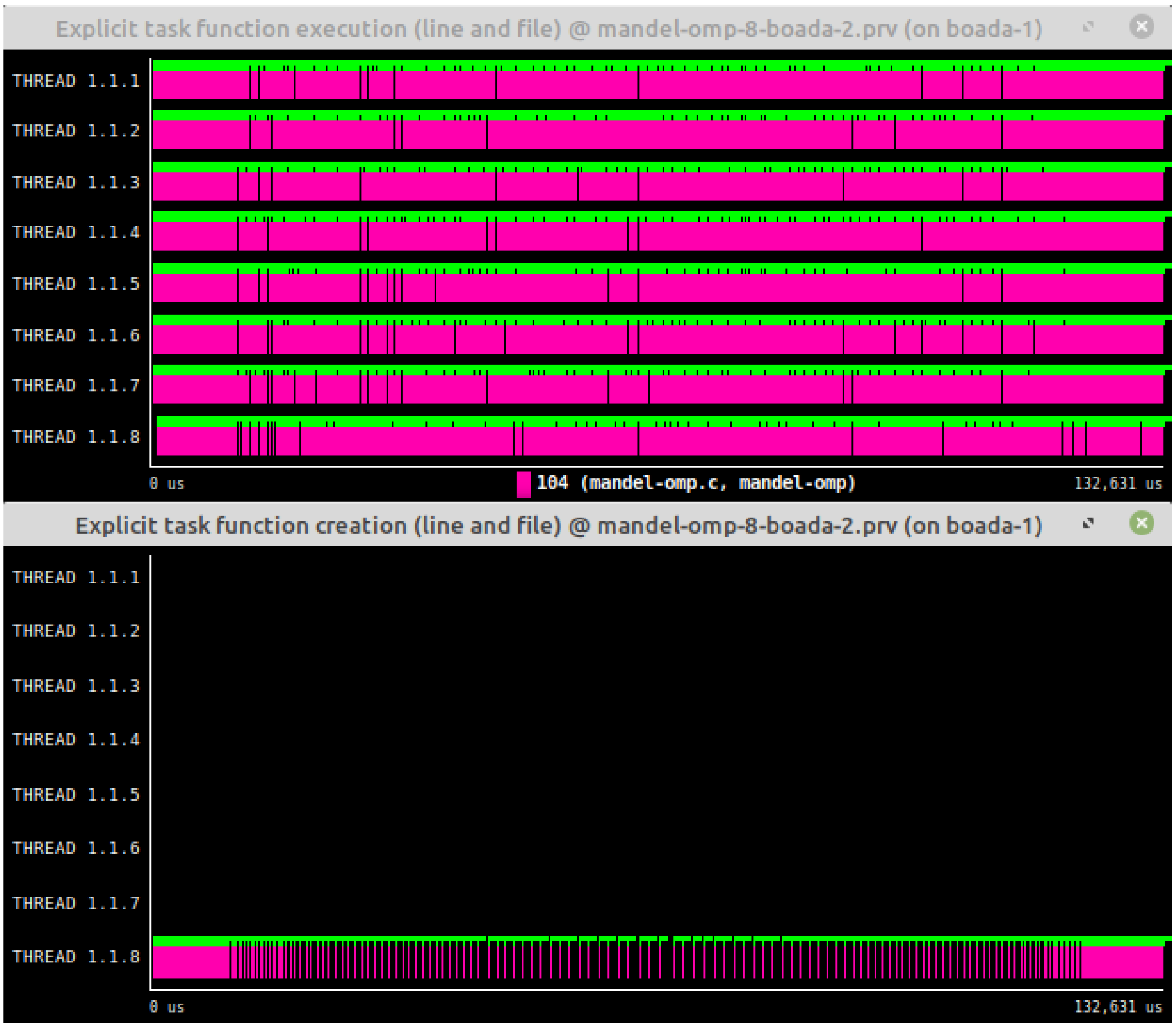
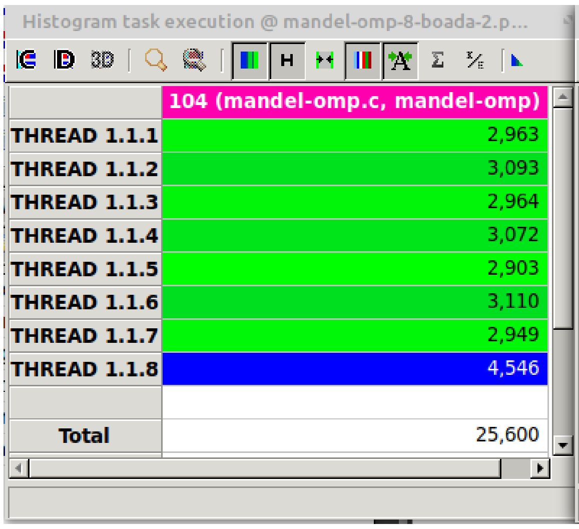
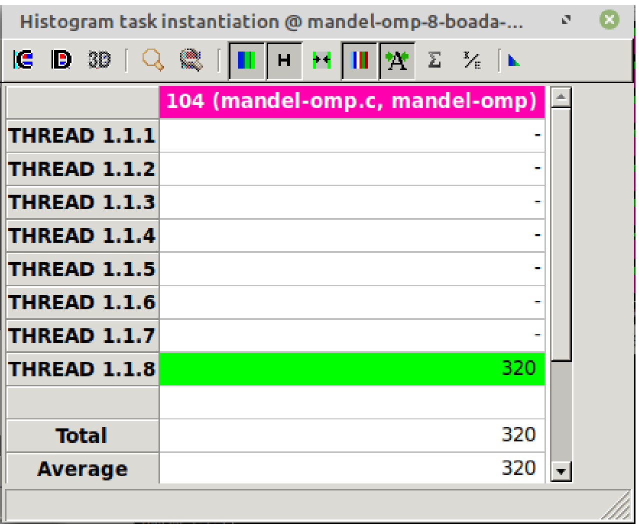


Figure 14: Paraver visualization of the threads creating and executing explicit tasks for the taskloop directive


Histogram task execution @ mandel-omp-8-boada-2.p...

	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	2,963
THREAD 1.1.2	3,093
THREAD 1.1.3	2,964
THREAD 1.1.4	3,072
THREAD 1.1.5	2,903
THREAD 1.1.6	3,110
THREAD 1.1.7	2,949
THREAD 1.1.8	4,546
Total	25,600

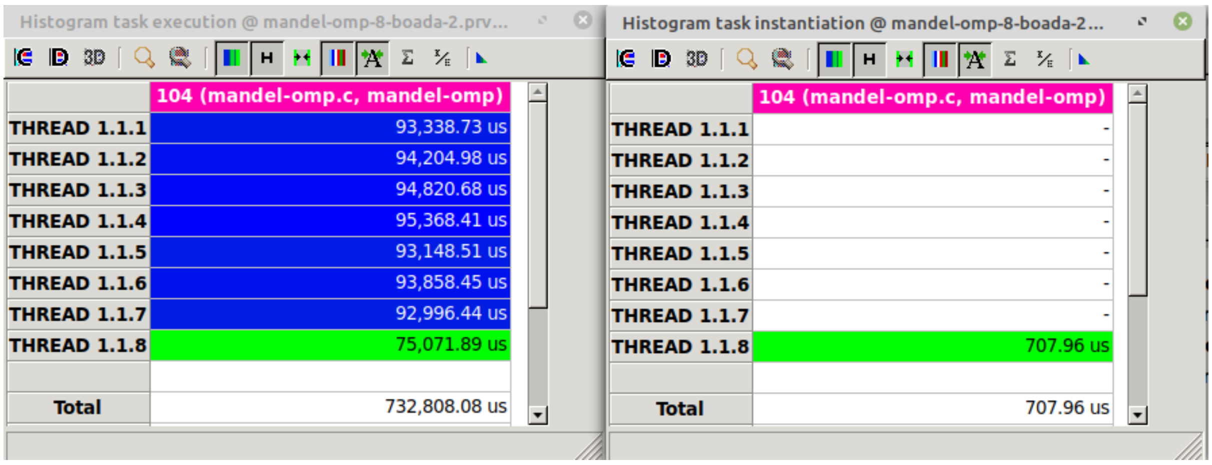

Histogram task instantiation @ mandel-omp-8-boada-...

	104 (mandel-omp.c, mandel-omp)
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	320
Total	320
Average	320

Figure 15: number of tasks executed *Figure 16: number of tasks created*

Is the load well balanced (both in terms of number of tasks and in terms of total time executing tasks)? Which of the two is actually relevant?

In this case, it can be noted that there's a significant improvement, given the fact that, on the one hand, the time of both creation and execution of tasks decreases (compare figures 10 and 17), apart from the increase (in detriment of the synchronization and realization of forks) in the time dedicated to the execution of tasks centered in the computation of the points present in the Mandelbrot set (again, figures 11 and 18 can be compared to check such differences).



Histogram task execution @ mandel-omp-8-boada-2.prv...

	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	93,338.73 us
THREAD 1.1.2	94,204.98 us
THREAD 1.1.3	94,820.68 us
THREAD 1.1.4	95,368.41 us
THREAD 1.1.5	93,148.51 us
THREAD 1.1.6	93,858.45 us
THREAD 1.1.7	92,996.44 us
THREAD 1.1.8	75,071.89 us
Total	732,808.08 us

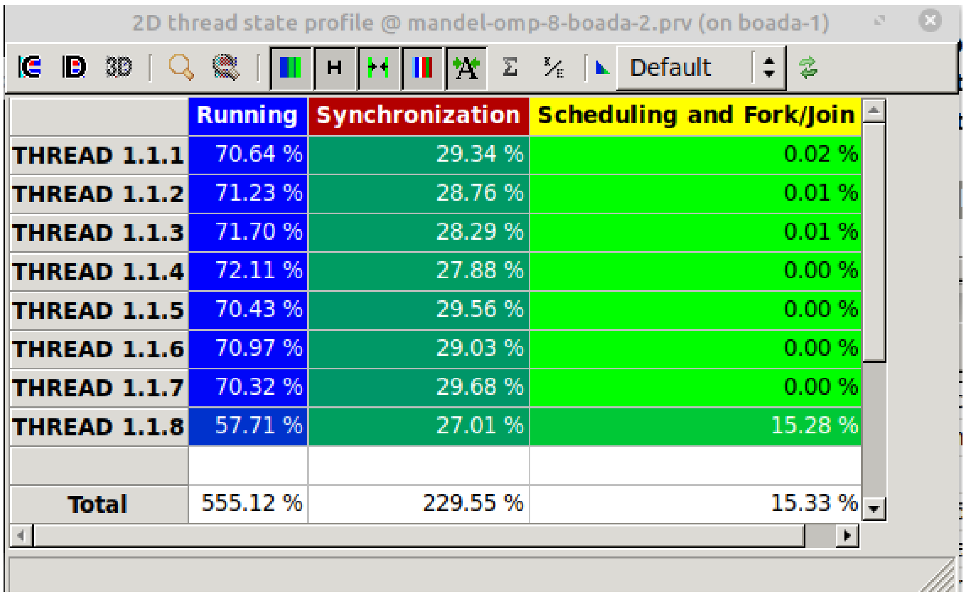
Histogram task instantiation @ mandel-omp-8-boada-2...

	104 (mandel-omp.c, mandel-omp)
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	707.96 us
Total	707.96 us

Figure 17: Histograms of the threads creating and executing explicit tasks

3.d) In summary, is granularity of the tasks appropriate? Would it be easy to change?

Granularity is already appropriate, given that, as it has been said in the previous answer, now there's more time dedicated to the execution of the program itself than in the synchronization between tasks, which implies that the present granularity suits the problem. Regarding the easiness to modify it, we can add, for example, the directive *grainsize* to make such modifications.



2D thread state profile @ mandel-omp-8-boada-2.prv (on boada-1)

Default

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	70.64 %	29.34 %	0.02 %
THREAD 1.1.2	71.23 %	28.76 %	0.01 %
THREAD 1.1.3	71.70 %	28.29 %	0.01 %
THREAD 1.1.4	72.11 %	27.88 %	0.00 %
THREAD 1.1.5	70.43 %	29.56 %	0.00 %
THREAD 1.1.6	70.97 %	29.03 %	0.00 %
THREAD 1.1.7	70.32 %	29.68 %	0.00 %
THREAD 1.1.8	57.71 %	27.01 %	15.28 %
Total	555.12 %	229.55 %	15.33 %

Figure 18: Histogram of the threads's state related with synchronization and task scheduling

3. How many tasks are executed per taskloop? Does this number depend on the number of threads that are used to execute?

In this particular case, there are 320 tasks per *taskloop*. This number indeed depends on the number of threads, given that in the absence of directives such as *num_tasks* or *grainsize* in the *pragma* clause, this division of tasks is done according to the threads used to execute the program.

4. Where are these task synchronisations happening? Do you think these task barriers are necessary? Has the scalability improved?(nogroup clause). Observe how tasks are now executed.

Synchronizations take place when tasks have finished their execution, just before starting the execution of the next group of tasks. Regarding its necessity, it is noticeable that, due to the barriers, threads lose a certain amount of time of possible execution of tasks, which translates into efficiencies that, in cumulative terms, implies an increase in the execution time of the program. That said, with the clause *nogroup* it can be noted the considerable increase, not only in execution time, but also in the speed-up of the program. With this, it is confirmed that said *barriers* are not necessary for the execution of the program.
It is worth noting that, just as we have stated before, there's little room for increasing both speed-up and execution time, given the present tendencies represented in the following figures.

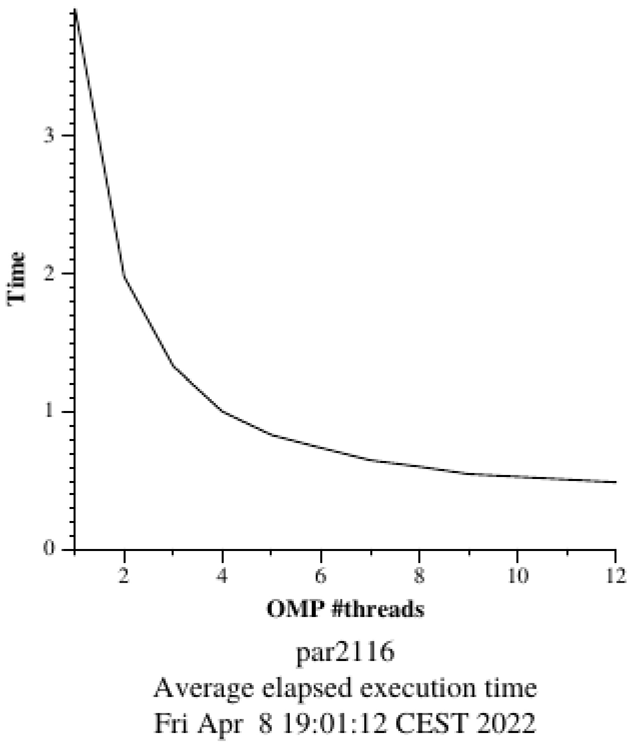


Figure 19: elapsed time of the point strategy execution using taskloop and nogroup clause

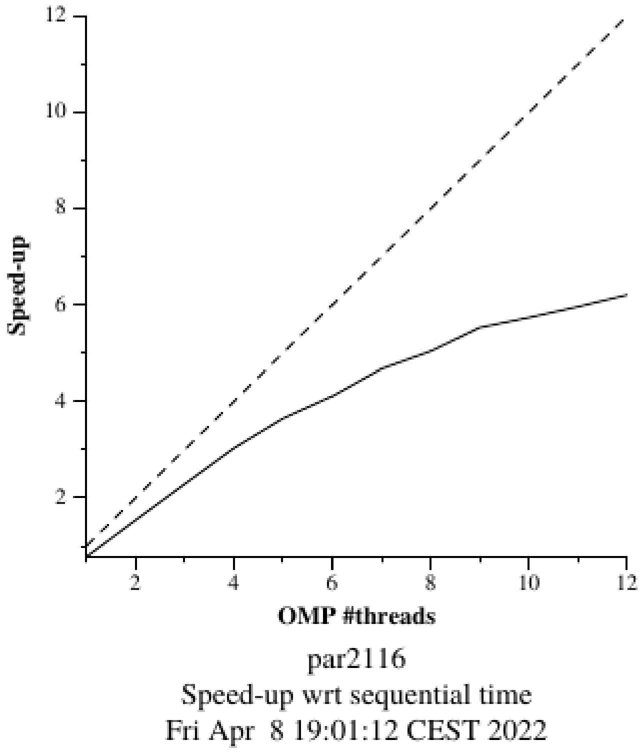


Figure 20: speed-up of the point strategy execution using taskloop and nogroup clause

Now, in contrast to the execution with the implicit *taskloop*, threads don't have to wait for one another, once they have finished executing their task, to execute a new one. WIth this improvement, in the following figure it can be observed that practically all threads use their time mainly to execute tasks.

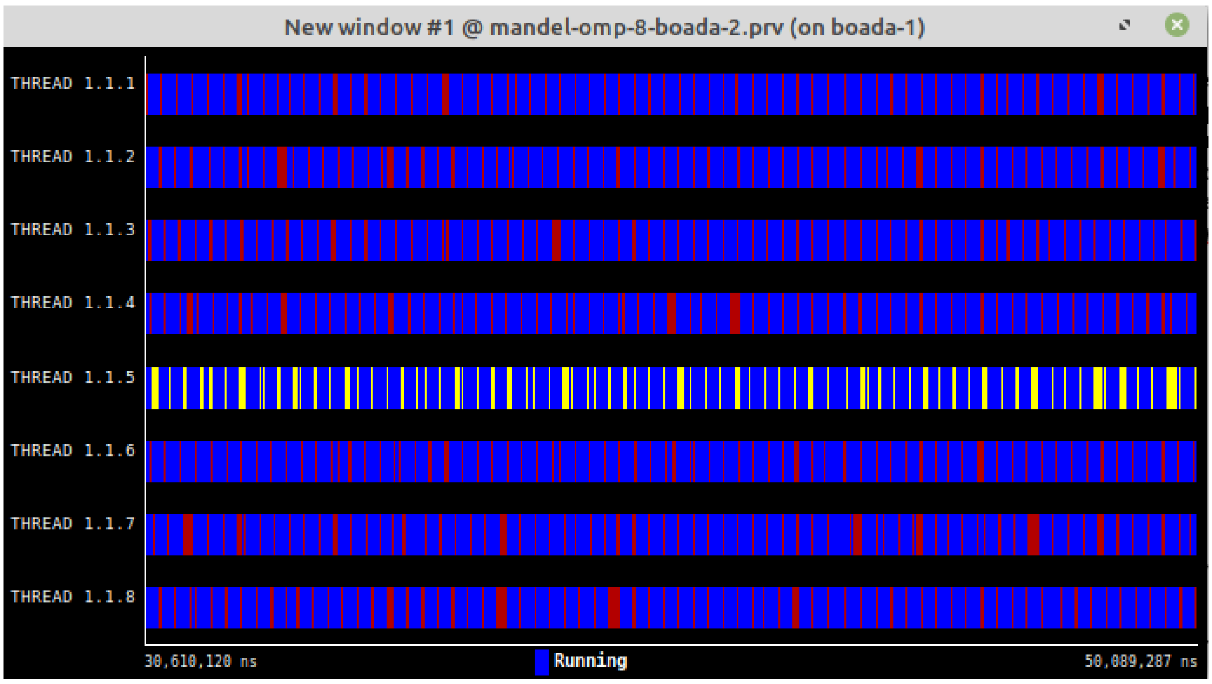


Figure 21: Mandel-omp's execution with taskloop nogroup clause

Row strategy implementation

1. Check that the result is correct and do the analysis of its strong scalability.

With the implementation of *row strategy*, we have been able to reduce the number of tasks to execute (1 task per row). This change has given room to a notable improvement in the execution of the program with a bigger number of threads (granularity decreases and, with a small amount of threads, the time of execution is similar to that of the implementation of *point strategy* with the use of *pragma omp taskloop nogroup*), resulting in a bigger speed-up. Again, it is worth mentioning that the tendency is not only more promising, but that it escalates better with the present number of threads. With it, we can surely state that such a strategy, for now, looks more promising.

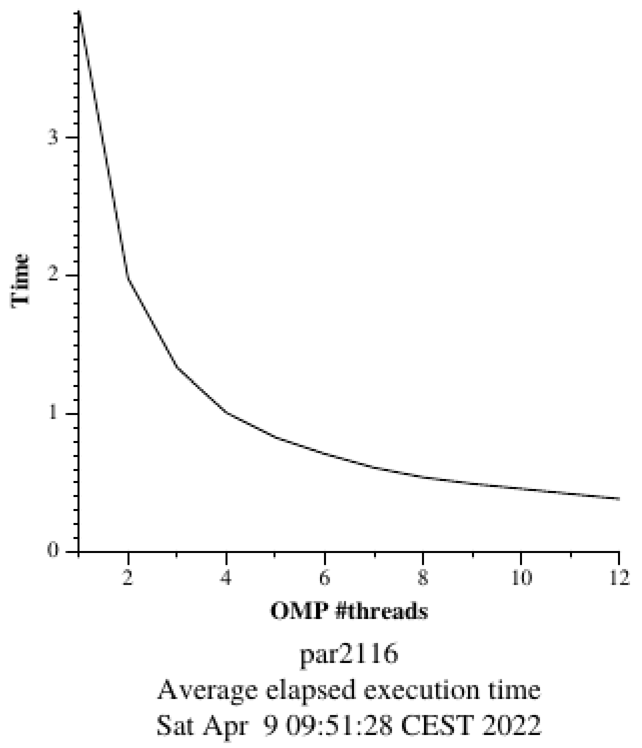


Figure 22: elapsed time of the row strategy execution

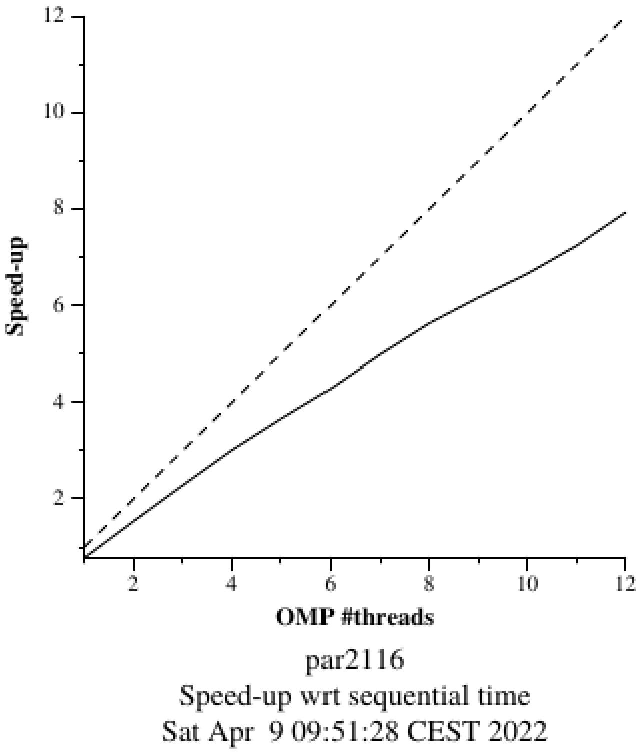


Figure 23: speed-up of the row strategy execution

2. Use again all previous configuration files to support your reasoning and draw the attention to the main differences that you observe when comparing the Row and Point granularities. Analyze the number of tasks created vs. the number of tasks executed and their granularities. Is there any load unbalance? Would the use of grainsize reduce it? Any task synchronization to eliminate?

With *row strategy*, as it can be observed in figure 24, all explicit tasks are created at the start of execution and threads themselves are the ones who reach out to the pool of tasks to pick one and execute them once they finish with the one they were computing previously, quite differently from the dynamic followed in the *point strategy* (where tasks are being created along the execution of the program; notice figures 9 and 14 for said differences).

In figure 25 we could consider that there's a work unbalance, given the fact that the 5-th thread executes more than ⅓-rd of the available tasks, but then we can also notice that the total execution time of each thread is relatively compensated, which means that work is balanced. Moreover, the greatest improvement of this implementation can be observed in figure 26 and it affects the general performance of the strategy. All threads dedicate more than 94% of their time to the execution (*running state*) of the program.
Regarding the use of *grainsize* to reduce the unbalances in work load, it is true that it can be used to get the present results (even though trying, for example, to reduce the load unbalance of executed tasks between processors could provoke a general unbalance in the total execution time of the program). It is worth mentioning the contrast between the total number of tasks executed in both strategies: while *row strategy* executes a total number of 320 tasks, *point strategy* executes 25600 tasks, which leads us to believe that there's a creation of a total of 320 task pools, each one with 80 tasks. It is easy to notice the evident improvement not only in terms of execution time of the tasks, but in the sheer total number (both with regard to their existence and execution) of them.
Finally, there exist no synchronizations to eliminate given the fact that, as it can be noticed in figure 26, their presence has been considerably reduced (which can be confirmed if it is compared with figure 18). This arises from the fact that, once all the tasks have been created, threads just keep picking and executing them, which means that the synchronizations that take place are the ones that cannot be in any way suppressed (be it the ones that take place between the execution of different tasks, or the ones that take place at the end of the program execution).

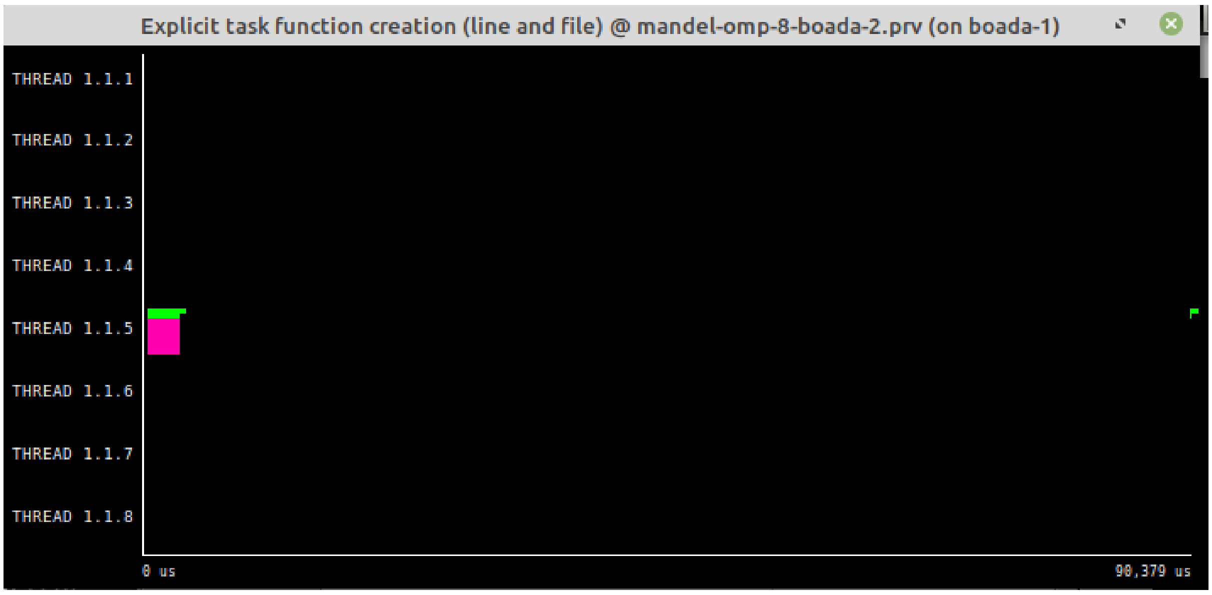
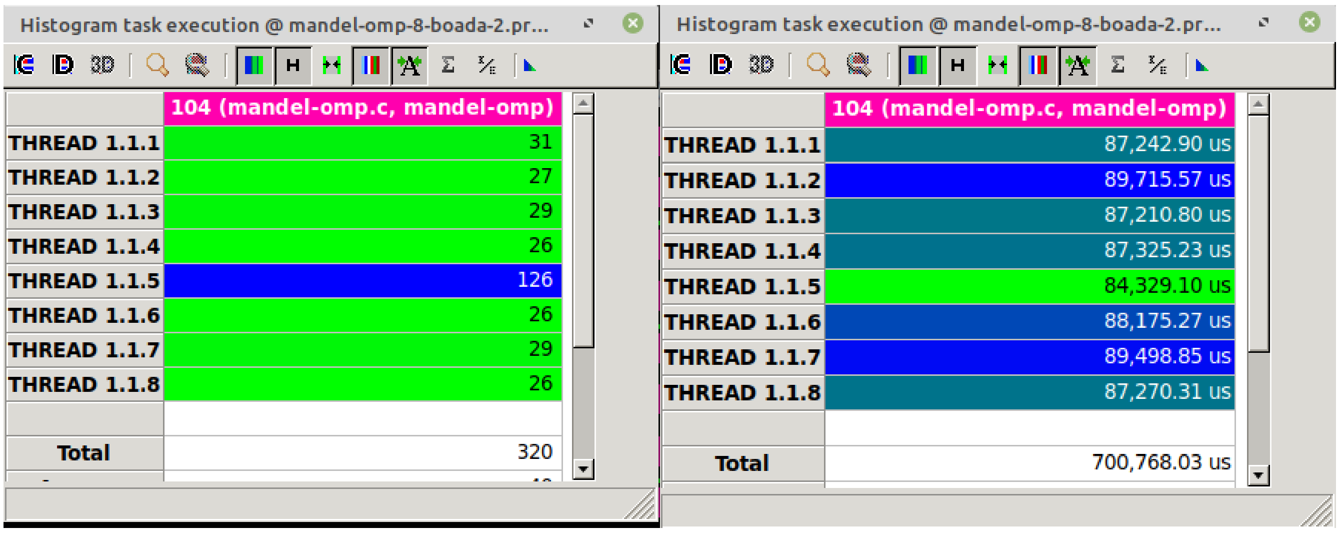


Figure 24: Paraver visualization of the threads creating explicit tasks

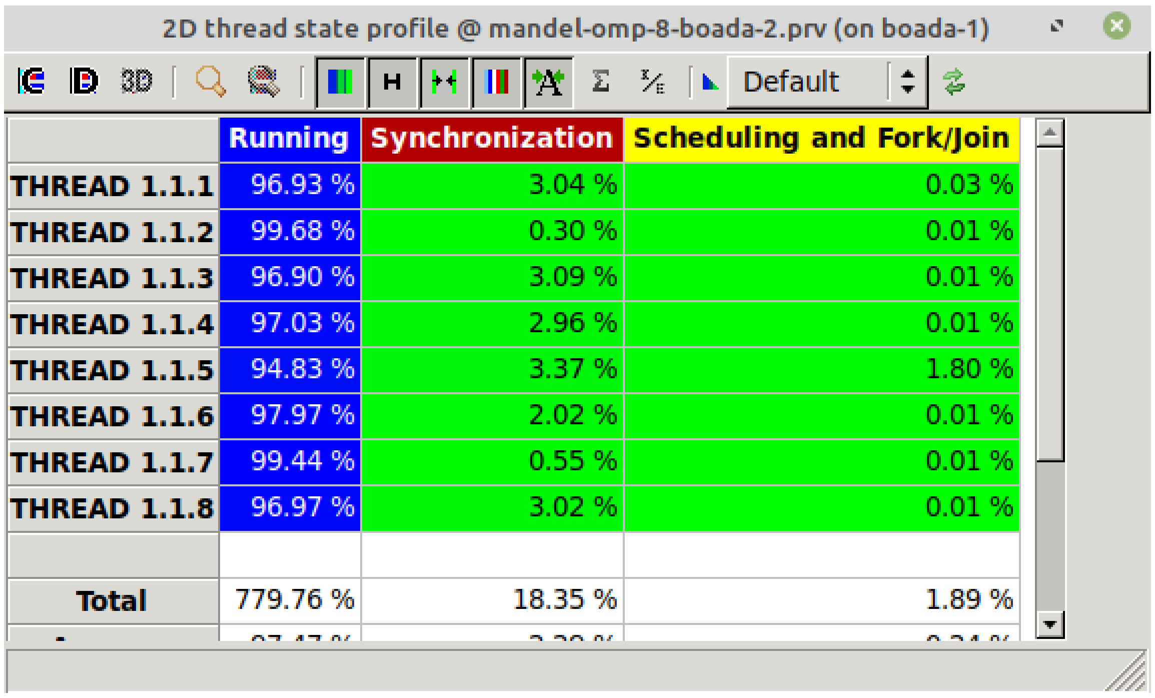

Histogram task execution @ mandel-omp-8-boada-2.pr...

	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	31
THREAD 1.1.2	27
THREAD 1.1.3	29
THREAD 1.1.4	26
THREAD 1.1.5	126
THREAD 1.1.6	26
THREAD 1.1.7	29
THREAD 1.1.8	26
Total	320

Histogram task execution @ mandel-omp-8-boada-2.pr...

	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	87,242.90 us
THREAD 1.1.2	89,715.57 us
THREAD 1.1.3	87,210.80 us
THREAD 1.1.4	87,325.23 us
THREAD 1.1.5	84,329.10 us
THREAD 1.1.6	88,175.27 us
THREAD 1.1.7	89,498.85 us
THREAD 1.1.8	87,270.31 us
Total	700,768.03 us

Figure 25: Histograms of the threads executing explicit tasks


2D thread state profile @ mandel-omp-8-boada-2.prv (on boada-1)

Default

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	96.93 %	3.04 %	0.03 %
THREAD 1.1.2	99.68 %	0.30 %	0.01 %
THREAD 1.1.3	96.90 %	3.09 %	0.01 %
THREAD 1.1.4	97.03 %	2.96 %	0.01 %
THREAD 1.1.5	94.83 %	3.37 %	1.80 %
THREAD 1.1.6	97.97 %	2.02 %	0.01 %
THREAD 1.1.7	99.44 %	0.55 %	0.01 %
THREAD 1.1.8	96.97 %	3.02 %	0.01 %
Total	779.76 %	18.35 %	1.89 %

Figure 26: Histogram of the threads's state related with synchronization and task scheduling

Optional

Explore how the best versions for Point and Row behave in terms of performance for different task granularities, i.e. when setting the number of tasks or the number of iterations per task to a different value than the one chosen by OpenMP.

To adapt our program to the given script for this exercise, we have used the variable *user_param*, which is initialized in the *main* function, where the information given by the script is picked up and transformed into a float to interact with it.
Moreover, for both strategies, we have made use of the following clauses:
#pragma omp taskloop firstprivate(row) num_tasks(user_param) : Point strategy
#pragma omp taskloop num_tasks(user_param) : Row strategy
(Attached files can be checked for more information on the source code)

As it can be seen in figures 27 and 28, the increase in the number of tasks generated in the *taskloop* imply a decrease in the total time of execution of the program, given the fact that they have been successfully parallelized. However, according to the followed strategy, it can be observed that, with more or less celerity, a stallment in optimization is reached, from which no improvement is really achievable anymore: on the one hand, the execution for the *point strategy*, even though it is able to reduce the overall time of execution, the optimizations that are reached are achieved with relative difficulty with regard to their cost in time, and for such affirmation we can compare its execution to that of the *row strategy,* which, for each subsequent increase of number of tasks, the time is considerably better (in contrast to that of the *point strategy*); on the other hand, for the *row strategy* it can be observed that, with a considerable amount of tasks, the execution time doesn't get any better, but it neither gets worst in terms of efficiency, phenomenon that does not take place in *point strategy*, where an increase in the execution time (when reaching the maximum amount of tasks per taskloop) really takes place, which translates into an inefficiency of the execution of the program.

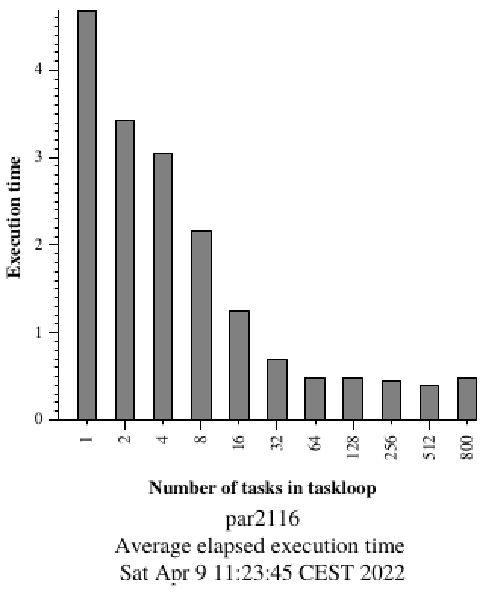


Figure 27: Execution of submit-numtasks-omp.sh for point strategy

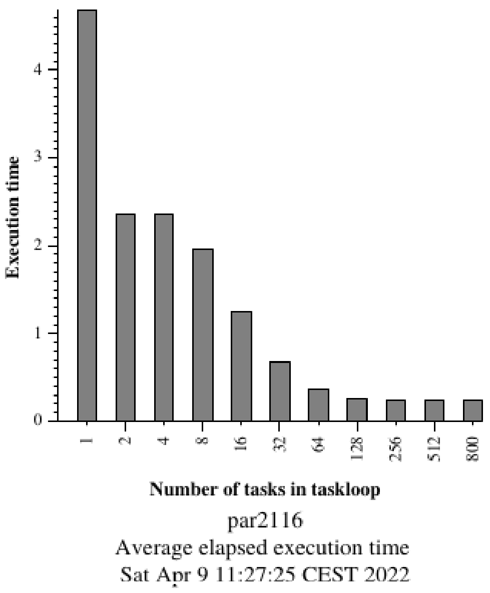


Figure 28: Execution time of submit-numtasks-omp.sh for row strategy

Conclusion

First of all, in this lab we've been able to check the existing relation between task dependencies and granularity itself, for we've seen that, for *point strategy*, we have more parallelism than with *row strategy*, which translates into less dependencies for given executions (this can be seen if figures 1 and 3 are compared with 4 and 6; for example, the dependencies without any execution option translates into the creation of a *lot* more tasks for *point strategy*, which also affects the dependencies of the execution with *-h* execution, which allows more parallelism).
Second of all, we've seen how speed-up and execution time are related to granularity and the direct correlation between them: for *point strategy*, more task creation gives place to more overheads, which in turn reduces both speed-up and execution time of the program. Such a feature also implies that there can be a work unbalance (since there's more time spent in task creation and synchronization between tasks than with execution itself). However, given the existence of different directives, we've also noticed that modifying the granularity of *point strategy* ends up being helpful, for speed-up and execution time find themselves optimized (the former increased, the latter reduced), which at the same time results in a balancing of work (and less time spent in synchronizations/forks). Moreover, in this same line of work, studying *row strategy* confirms these previous observations, for, with less granularity, the program not only takes less time to execute, but scales overall better (given that its speed-up has an observable tendency that leads us to think that it'd keep increasing with each new thread added to the execution of the program). This also gives place to the complete balancing of the work present in the execution of the program, which ensures that the main load of work that there'll ever be is the execution of the program itself. It must be said that one of the main causes of this is that, whereas *point strategy* tasks were created on-the-run, with *row strategy* tasks are created at the beginning of the program and, as execution goes on, each thread returns to the pool to pick up a new task.