PARAL·LELISME

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

Sergio Utrero Preciado (**par1122**)
Jordi Bru Carci (**par1104**)
20/04/2022
2021-2022.Q2

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

TABLE OF CONTENTS

Introduction 2

Task decomposition and granularity analysis 2

- Analysis of the Row strategy 3
- Analysis of the Point strategy 4

Implementing task descompositions in OpenMP 10

- Point strategy implementation using task 10
- Point strategy with granularity control using taskloop 16
- Point strategy with granularity control using taskloop nogroup 21
- Row decomposition in OpenMP 23
- Some conclusions 27

Optional 28

Introduction

In this laboratory session we will work with OpenMP, in order to study different tasking models of parallelization, in particular, the computation of the Mandelbrot set. Before starting, it is necessary to understand what the Mandelbrot set is.

In general, a Mandelbrot set marks the set of points in the complex plane, whose boundaries generate an easy-to-recognize 2-dimensional fractal shape. For a more in-depth explanation of the Mandelbrot set, as we did, we recommend taking a loot to its Wikipedia page: http://en.wikipedia.org/wiki/Mandelbrot_set.

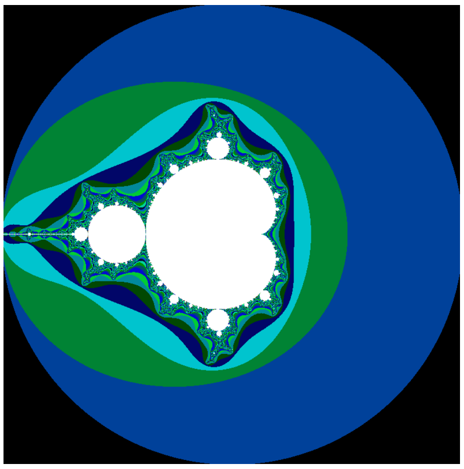


Figure 1. Screenshot of the picture from the execution of mandel-tar.c with the -d option

Before studying different tasking models of the program, we will analyze different task decomposition strategies and granularities explored with the tool introduced in previous assignments, the *Tareador*.

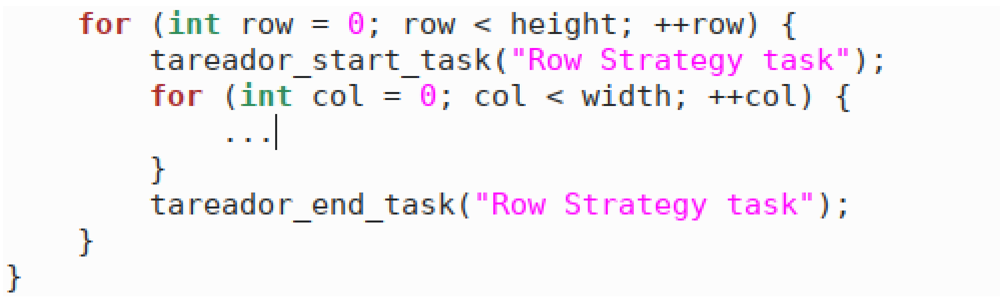
Task decomposition and granularity analysis

This section will be divided into two parts, the granularity analysis by decomposing by task using the row and point strategy.

Analysis of the Row strategy

In this strategy, each created task will correspond to a single row computation of the Mandelbrot set.

Figure 2 shows the modification of the initial code to decompose it by rows.



```
    for (int row = 0; row < height; ++row) {
        tareador_start_task("Row Strategy task");
        for (int col = 0; col < width; ++col) {
            ...
        }
        tareador_end_task("Row Strategy task");
    }
}
```

Figure 2. Screenshot of the key fragment of mandel-tar.c for the row strategy implementation

Now we can execute it with *Tareador* so it will show us the dependency graph of the program. The green nodes represent each row computation of the Mandelbrot set. As it shows in figure 3, the program runs fully parallel and it is important to highlight that there are not any dependencies between all the tasks. Also, there are 8 nodes because we are using -w 8 as the size of the Mandelbrot image in order to generate a reasonable task graph in a reasonable execution time.

It is interesting to see how not all nodes have the same size, since not all tasks execute the same number of instructions. The small nodes in the graph are those ones that most of the iterations cancel out immediately or earlier than the max iterations we have set. If one point does not surpass the threshold in 10.000 iterations we will assume that it will never do. Even though we have an unbalance, we still can parallelize all the tasks without dependencies.

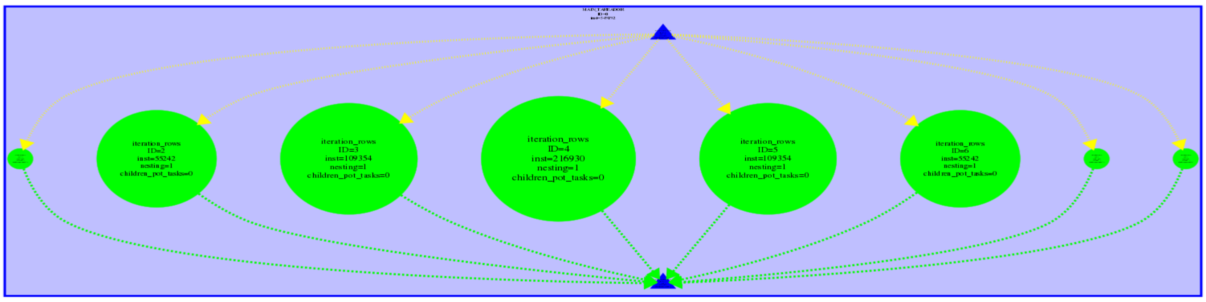

MAIN_TAREADOR
iteration_rows ID=2 inst=55242 nesting=1 children_pot_tasks=0
iteration_rows ID=3 inst=109354 nesting=1 children_pot_tasks=0
iteration_rows ID=4 inst=216930 nesting=1 children_pot_tasks=0
iteration_rows ID=5 inst=109354 nesting=1 children_pot_tasks=0
iteration_rows ID=6 inst=55242 nesting=1 children_pot_tasks=0

Figure 3. Dependency graph of the row strategy based mandel-tar.c

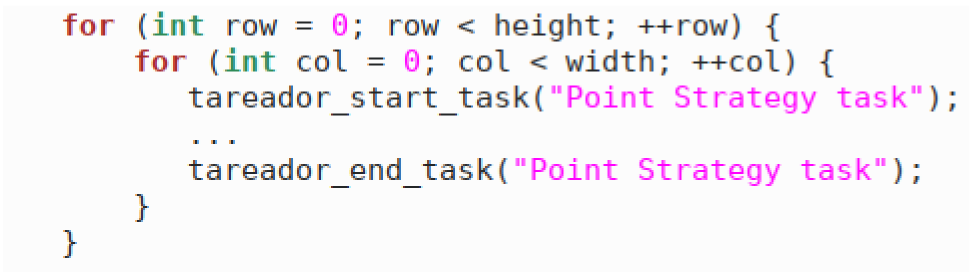
In addition to this execution, we have performed two other runs using the -d and -h options in order to compare results.
Using the -d option, the Mandelbrot set is displayed for visual inspection and the option -h, the histogram for the values is also computed.

In the first case, as we can see in Figure 6, the execution with -d shows us a totally sequential dependency graph with the same nodes as the previous one, maintaining the same sizes as the previous execution and each one depending on the other. That’s because we have to wait for the ending of one row to start the execution on the next.

In the other case, we will compute the set and also the histogram to see how that option affects the parallelization. we can see how in the dependency graph the tasks continue to depend on each other but with different topology. Figure 7 shows that some tasks are executed in parallel because of how we store the histogram. We increment a vector position each time a k-value is found ("histogram[k-1]++"). In the case of two values being equal we have to access the same position inside the vector, creating these dependencies.

Analysis of the Point strategy

In this strategy, each created task will correspond to a single point computation of the Mandelbrot set.
Figure 4 shows the modification of the initial code to decompose it by points.



```
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
       tareador_start_task("Point Strategy task");
       ...
       tareador_end_task("Point Strategy task");
    }
}
```

Figure 4. Screenshot of the key fragment of mandel-tar.c for the point strategy implementation

Now it is the turn to execute it with *Tareador* so it will show us the dependency graph of the program. The green nodes represent each point computation of the Mandelbrot set. As we can see in figure 5, the program runs fully parallel (as the previous strategy).
Also, there are 64 nodes because we are using -w 8 (8 points per row, 8*8 = 64) as the size of the Mandelbrot image in order to generate a reasonable task graph in a reasonable execution time.
If we focus on the size of the nodes, we can say that it shows exactly the same as with the other strategy. Not all nodes have the same size, so we can tell the same conclusions that we have quoted in the previous analysis and then we could say that this strategy has parallel problems.

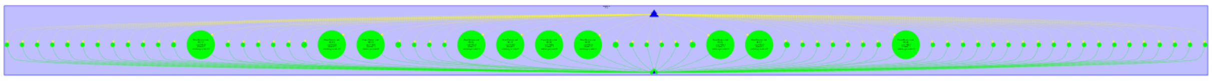


Figure 5. Dependency graph of the point strategy based mandel-tar.c

We will now proceed to do the same as in the previous analysis. We run the program with the options -d and -h so we can compare results.

In the first case, as we can see in Figure 8, the execution with -d option shows us the same topology as the other -d execution, basically a larger sequential dependency graph. To be exact, 64 nodes. As for the histogram graph now we can see which pixels have the same k-value and have to increment the same position on the vector. Interestingly, we have a critical path of eight nodes with max value as their k-value. This is obvious when you take into consideration that the nodes that truly belong to the Mandelbrot set will have to always reach max value to be accounted for.

As we have seen, we have some low cost nodes and others with very high cost, so, depending on the option we want to execute with the code. If we have to choose the best parallelization method we have to take in consideration the amount of processors we are going to use.

If we use less processors than points,the best method should be the row parallelization for a more balanced workshare (as seen in conclusions before).
On the other hand, if we use an infinite amount of processors we should always choose the point parallelization to have a better usage. As we have seen, the problem of this strategy is that the overheads of creation, synchronization and destruction scale at a quadratic rate due to the nature of the matrix problem.

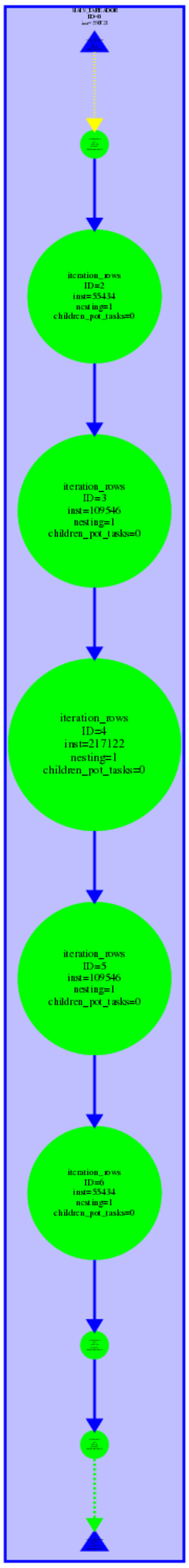


Figure 6. Dependency graph of the **row** strategy based mandel-tar.c with the -d option

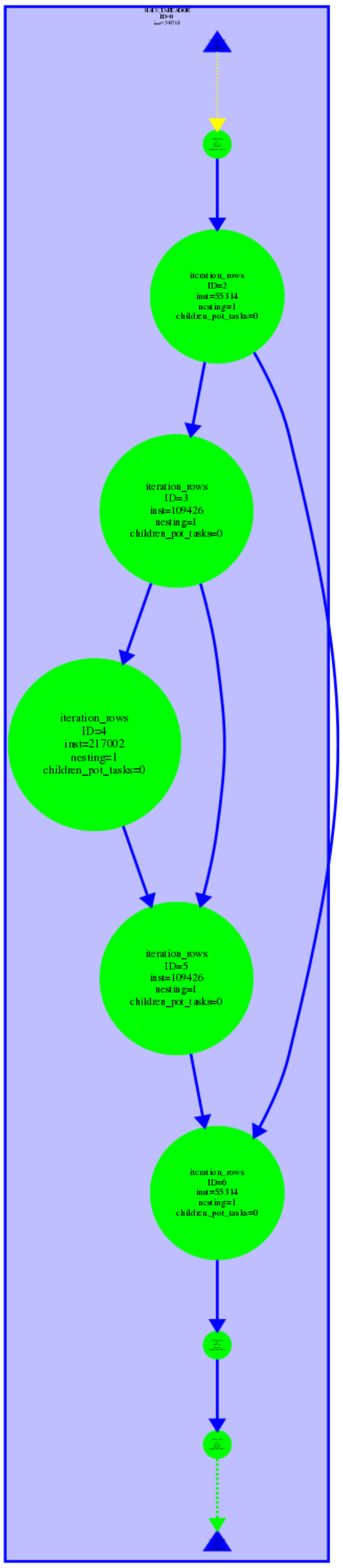


Figure 7. Dependency graph of the **row** strategy based mandel-tar.c with the -h option

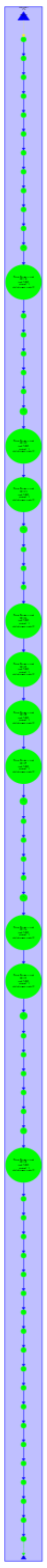


Figure 8. Dependency graph of the **point** strategy based mandel-tar.c with the -d option

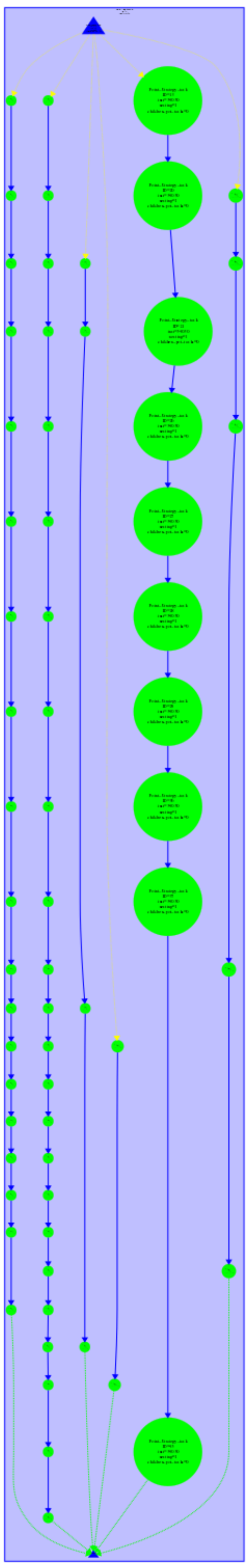
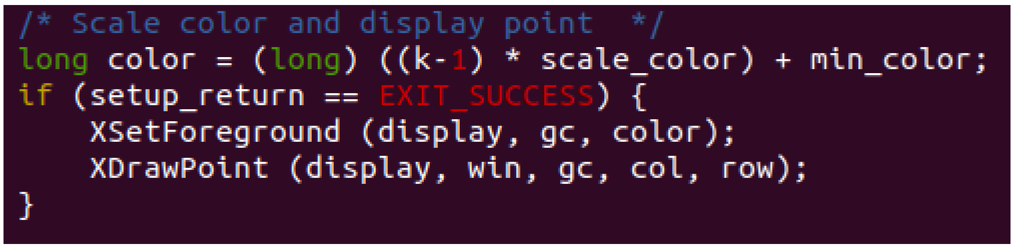


Figure 9. Dependency graph of the **point** strategy based mandel-tar.c with the -h option

To finish this section, we must find out which code fragment is causing the serialization of tasks when using the -d and the -h options.

Figure 10 is the cause of this happening because it is a function that each task has to execute and will execute whenever it can, regardless of the rest. If we want to make sure that this serialization does not occur, we must set the #pragma omp critical clause so that it creates a region where it can only be executed by one thread at a time.



```
/* 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 10. Fragment of madel-tar.c that causes serialization

Implementing task descompositions in OpenMP

For this session we will implement using OpenMP the definitions of tasks for a better implementation of the methods shown before.

Point strategy implementation using task

We start preparing the point strategy protecting the dependencies using both atomic and critical instances.

```
#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)
        {
        //code
        if (output2histogram) {
            #pragma omp atomic
            histogram[k-1]++;
        }

        if (output2display) {
           //code
            if (setup_return == EXIT_SUCCESS) {
                #pragma omp critical
                {
                XSetForeground (display, gc, color);
                XDrawPoint (display, win, gc, col, row);
                }
            }

        }
      }
    }
}
```

Figure 11. Fragment of madel-omp.c with the omp clauses for a point decomposition strategy

First of all, we declare a parallel region followed by a single clause, so we can declare a limit mark on the execution of both loops. We detected that we have to use some restrictions to avoid data races. The atomic clause shown in figure 11 is used to make sure that the histogram executes correctly and the critical one, to guarantee the order of execution of the two dependent functions.

The pragma omp task firstprivate(row,col) clause is used to assign two private variables to every task, row and col.

To check the correctness of this variation of the code, we compiled and executed the binary with “-d -h -i 100000” options and with some limitations on the available threads.

```
>OMP_NUM_THREADS=1 ./mandel-omp -d -h -i 10000
>OMP_NUM_THREADS=2 ./mandel-omp -d -h -i 10000
```

The difference between both executions is almost visibly insignificant. The display is the same and both lack brightness on the colors if we compare them to the initial display of the sequential execution.

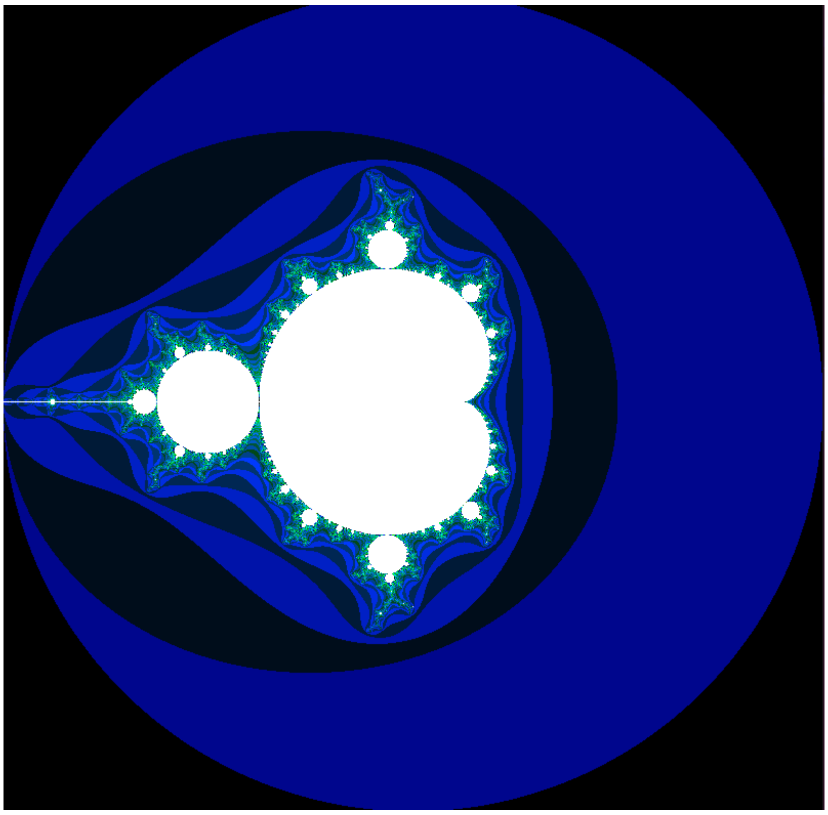


Figure 12. Parallelization of the Mandelbrot set with one processor

Now, we want to see how the speedup affects the parallelization. We are going to execute the code with 1 and 8 threads so we can visually see the different results.

```
>sbatch ./submit-omp.sh mandel-omp 1
>sbatch ./submit-omp.sh mandel-omp 8
```

#Threads	Sequential 1 Thrd	1 Thread	8 Threads
Time	3.039749 seconds	3.971361 seconds	1.306618 seconds

Figure 13. Result table of the time execution with different number of threads

As shown in figure 13, we can see that if we execute it with 8 threads we can actually have a better performance. On the other hand, the execution time using only one thread is worse. That means, for now, it shows that the 8 threads execution has a better performance than the sequential one.

In addition, we can measure the speedup in a more in-depth way with sumbit-strong.sh script provided.
>sbatch ./submit-strong.sh mandel-omp 1 12

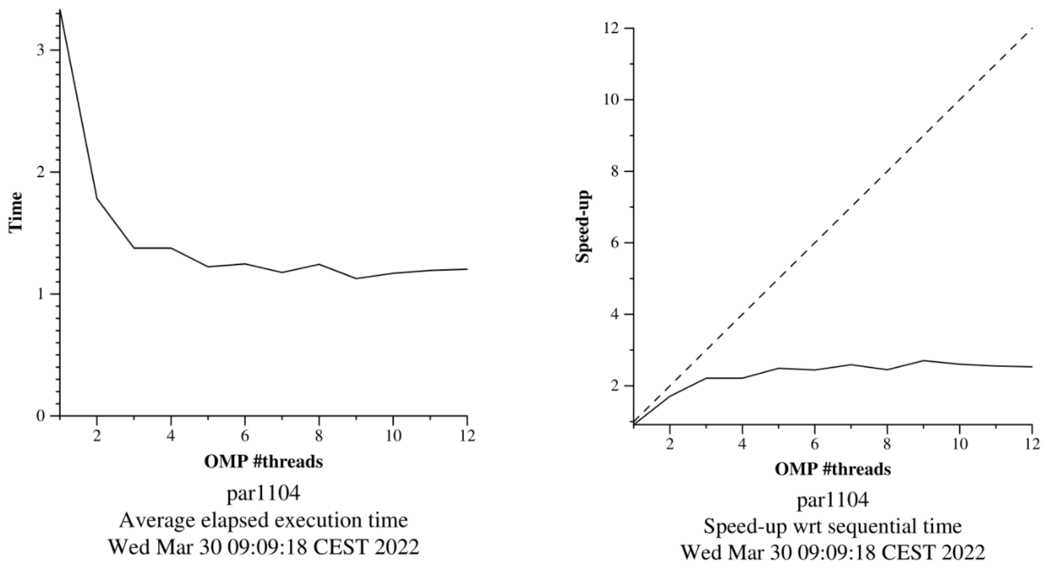
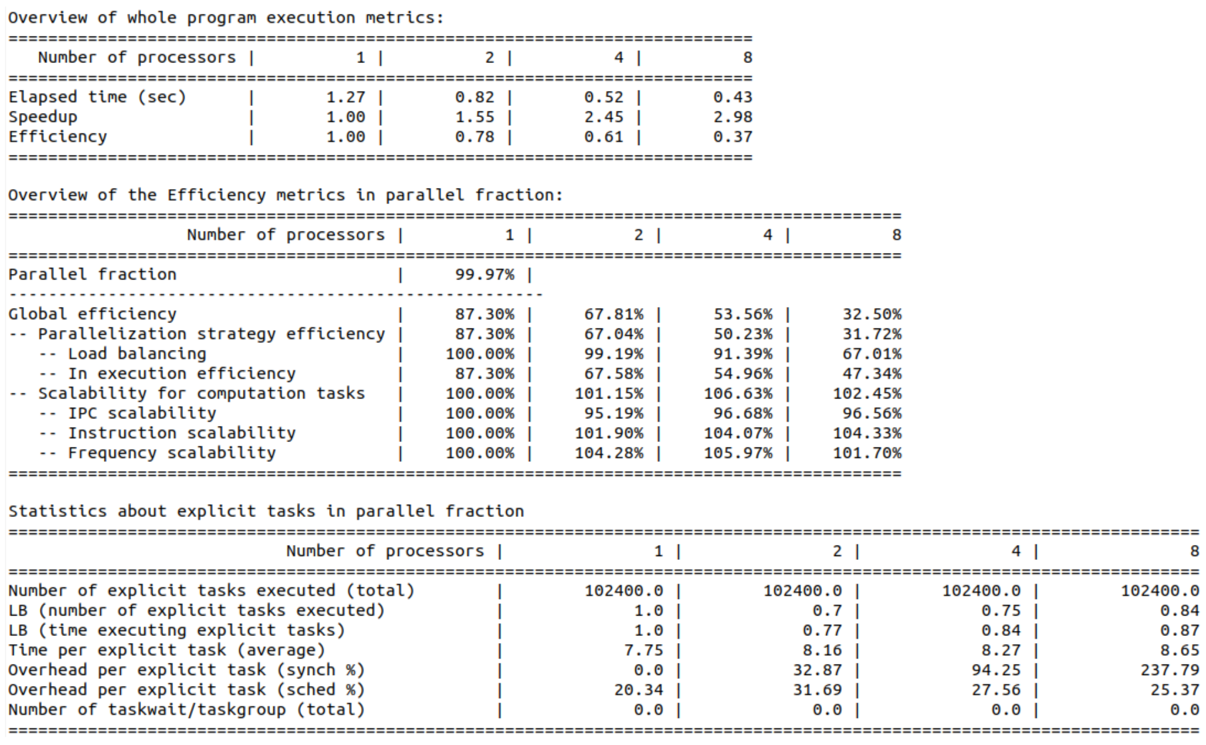


Figure 14. Time and speed-up plots of point task strategy

As we can see in figure 14, the point task strategy does not improve when the number of threads increases.

For a better understanding of how the execution goes, we submitted the submit-strong-extrae.sh script, which performs Extrae instrumented executions for 1, 2, 4 and 8 and invokes modelfactors.py to perform a first analysis of the parallel execution metrics.



```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |          1 |          2 |          4 |          8
===========================================================================
Elapsed time (sec)      |       1.27 |       0.82 |       0.52 |       0.43
Speedup                 |       1.00 |       1.55 |       2.45 |       2.98
Efficiency              |       1.00 |       0.78 |       0.61 |       0.37
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
==========================================================================================
Parallel fraction                      |     99.97% |
------------------------------------------------------
Global efficiency                      |     87.30% |     67.81% |     53.56% |     32.50%
-- Parallelization strategy efficiency |     87.30% |     67.04% |     50.23% |     31.72%
   -- Load balancing                   |    100.00% |     99.19% |     91.39% |     67.01%
   -- In execution efficiency          |     87.30% |     67.58% |     54.96% |     47.34%
-- Scalability for computation tasks   |    100.00% |    101.15% |    106.63% |    102.45%
   -- IPC scalability                  |    100.00% |     95.19% |     96.68% |     96.56%
   -- Instruction scalability          |    100.00% |    101.90% |    104.07% |    104.33%
   -- Frequency scalability            |    100.00% |    104.28% |    105.97% |    101.70%
==========================================================================================

Statistics about explicit tasks in parallel fraction
========================================================================================================================
                              Number of processors |                 1 |                 2 |                 4 |                 8
========================================================================================================================
Number of explicit tasks executed (total)          |          102400.0 |          102400.0 |          102400.0 |          102400.0
LB (number of explicit tasks executed)             |               1.0 |               0.7 |              0.75 |              0.84
LB (time executing explicit tasks)                 |               1.0 |              0.77 |              0.84 |              0.87
Time per explicit task (average)                   |              7.75 |              8.16 |              8.27 |              8.65
Overhead per explicit task (synch %)               |               0.0 |             32.87 |             94.25 |            237.79
Overhead per explicit task (sched %)               |             20.34 |             31.69 |             27.56 |             25.37
Number of taskwait/taskgroup (total)               |               0.0 |               0.0 |               0.0 |               0.0
========================================================================================================================
```

Figure 15. Screenshot of modelfactors.out of point task strategy

Figure 15 is the result of executing the script plus modelfactors.py, which contains three tables with valuable information to understand how this strategy performs. The first table details some parameters from the executed program with a different amount of threads, from 1 to 8. First table clearly shows how the efficiency falls when the amount of threads rises, so we can, at first sight, see how bad it performs.

The second table from figure 15 shows information about the efficiency of the program. A point to be highlighted is that the parameter called “parallelization strategy efficiency” also decreases when the amount of threads increases, when it should be the other way around. So we can give more weight to the argument that this strategy is not as efficient as we feared.

The third table from figure 15, in the same way as the others, details about the explicit tasks in the parallel section. At a first glance, we can highlight that the overhead per explicit task (synch%) is getting bigger as we increase the number of threads. The aim was to keep overhead low, especially in increasing the number of threads, therefore we could say that the cause of this is that the work is not well distributed.

In order to better justify and contrast these results obtained with modelfactors we have used the Paraver tool:

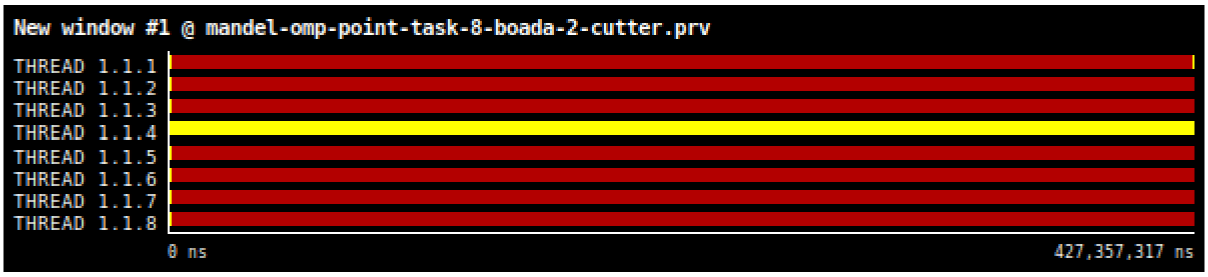


Figure 16. Cronogram of point task strategy with eight threads

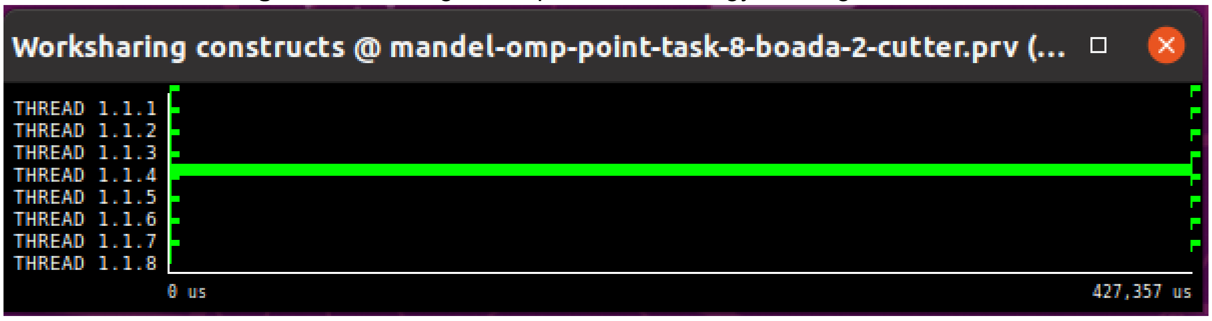
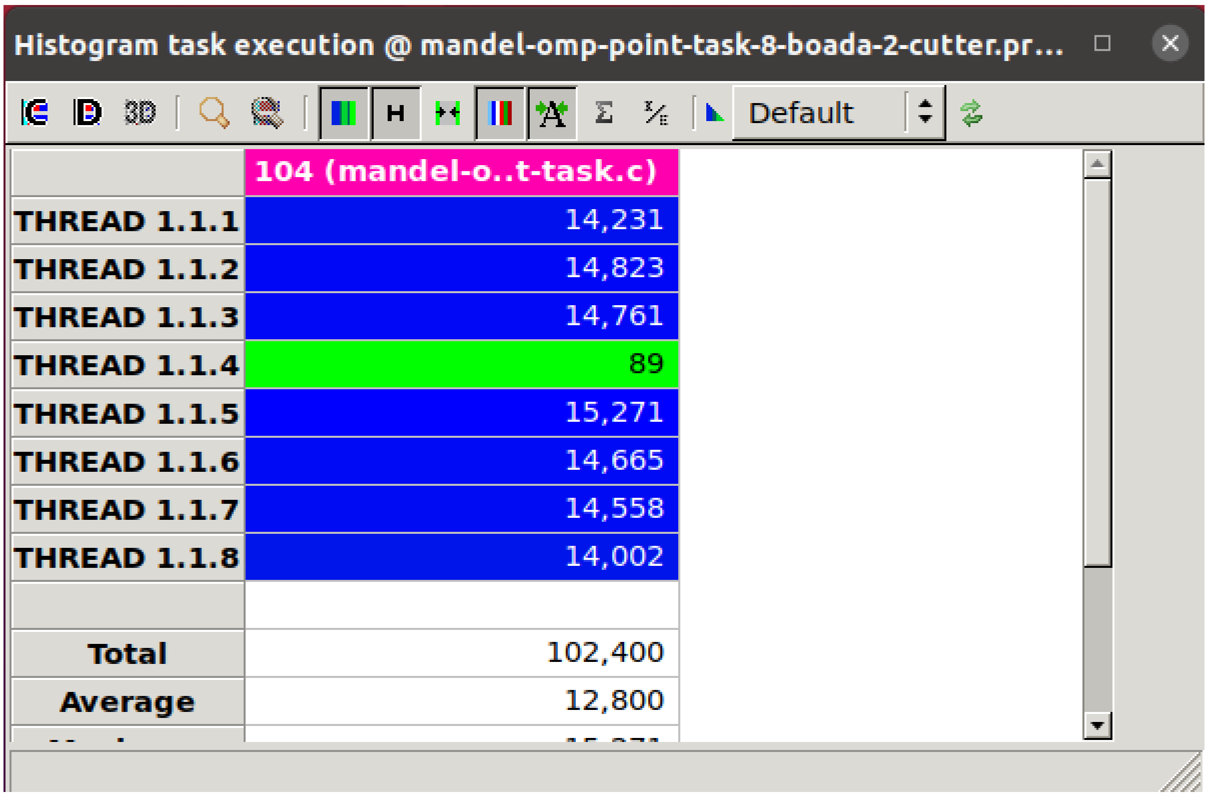


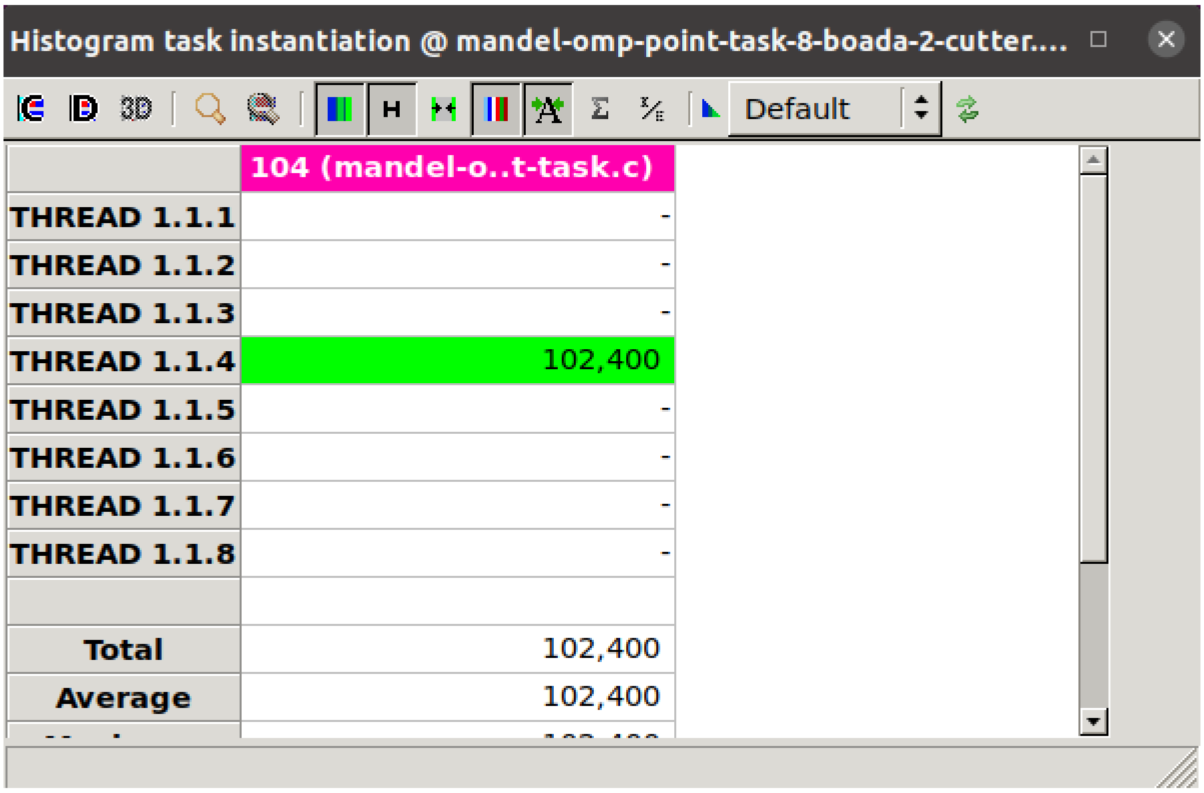
Figure 17. Worksharing constructs of point task strategy with eight threads

Figure 16 and 17 shows that one thread (number 4) enters the parallel region and starts creating tasks for the other threads. To better visualize how the tasks are generated and executed we can open the cronograms and histograms related to that information (figure 18 and 19).


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

	104 (mandel-o..t-task.c)
THREAD 1.1.1	14,231
THREAD 1.1.2	14,823
THREAD 1.1.3	14,761
THREAD 1.1.4	89
THREAD 1.1.5	15,271
THREAD 1.1.6	14,665
THREAD 1.1.7	14,558
THREAD 1.1.8	14,002
Total	102,400
Average	12,800

Figure 18. Histogram task execution of point task strategy with eight threads


Histogram task instantiation @ mandel-omp-point-task-8-boada-2-cutter....
Default

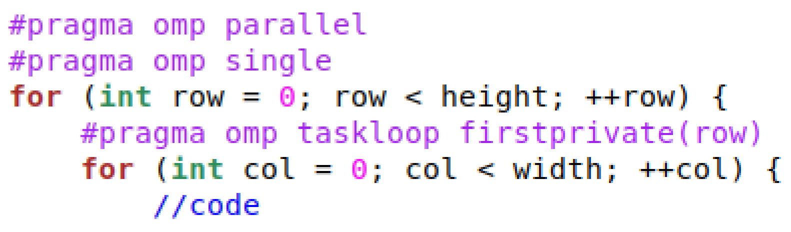
	104 (mandel-o..t-task.c)
THREAD 1.1.1	-
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	102,400
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	-
THREAD 1.1.8	-
Total	102,400
Average	102,400

Figure 19. Histogram task instantiation of point task strategy with eight threads

Both figures, 18 and 19, clearly show how bad the tasks are distributed. We can see that, as we mentioned earlier, the thread in charge of creating tasks is the number seven (figure 19). This processor creates 102.400 unique tasks, that is exactly the square of the resolution in pixels/points submit-extrae.sh sets to compute. The balance between the computation threads is fairly balanced, exceptuating the thread seven and one for aforementioned reasons (figure 18).

Point strategy with granularity control using taskloop

After developing some insights of the point strategy with tasks declaration, now we are going to set a task with a coarser granularity. We are going to repeat the same procedures and executions we did in the previous section.



```
#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) {
        //code
```

Figure 20. Fragment of the relevant code of mandel-omp.c using a point taskloop declaration

As we can see in figure 20, we removed the task declaration and we put the taskloop clause before the second for instead, so we can divide the subsequent loop in bits of some determined size that the program decides (we are not using num_tasks(N)). Now, we proceed to execute the new version of the program with 1 and 8 processors so we can see the execution time, as we did in previous sections.

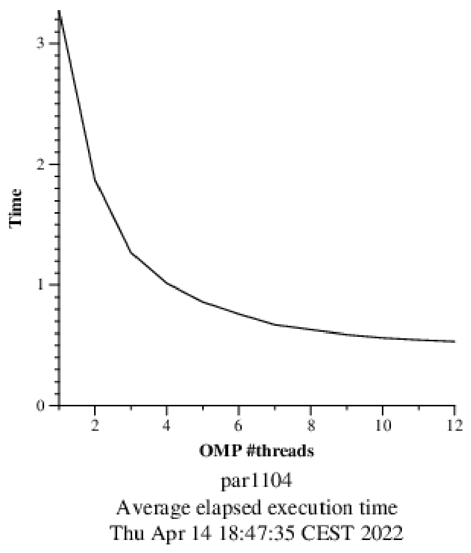
```
>sbatch ./submit-omp.sh mandel-omp 1
>sbatch ./submit-omp.sh mandel-omp 8
```

#Threads	Sequential 1 Thrd	1 Thread	8 Threads
Time	3.039749 seconds	3.259590 seconds	0.630575 seconds

Figure 21. Result table of the time execution with different number of threads in a point taskloop strategy

Figure 21 clearly shows that the performance of this new version is much better because we reduced the execution time significantly. The execution time with one thread now is almost at the same level as the serial one and the one executed with 8 threads it performed twice as fast than the previous section.

>sbatch ./submit-strong-omp.sh mandel-omp 1 12



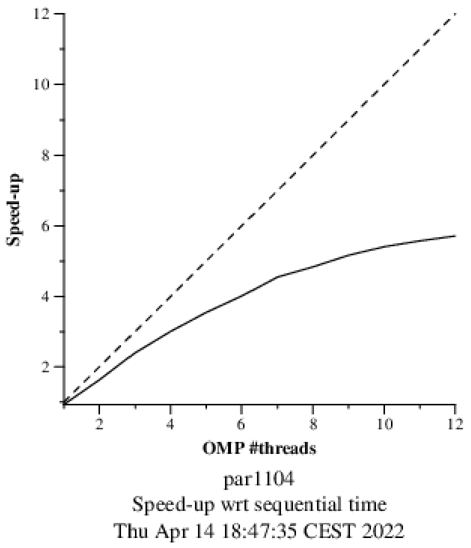
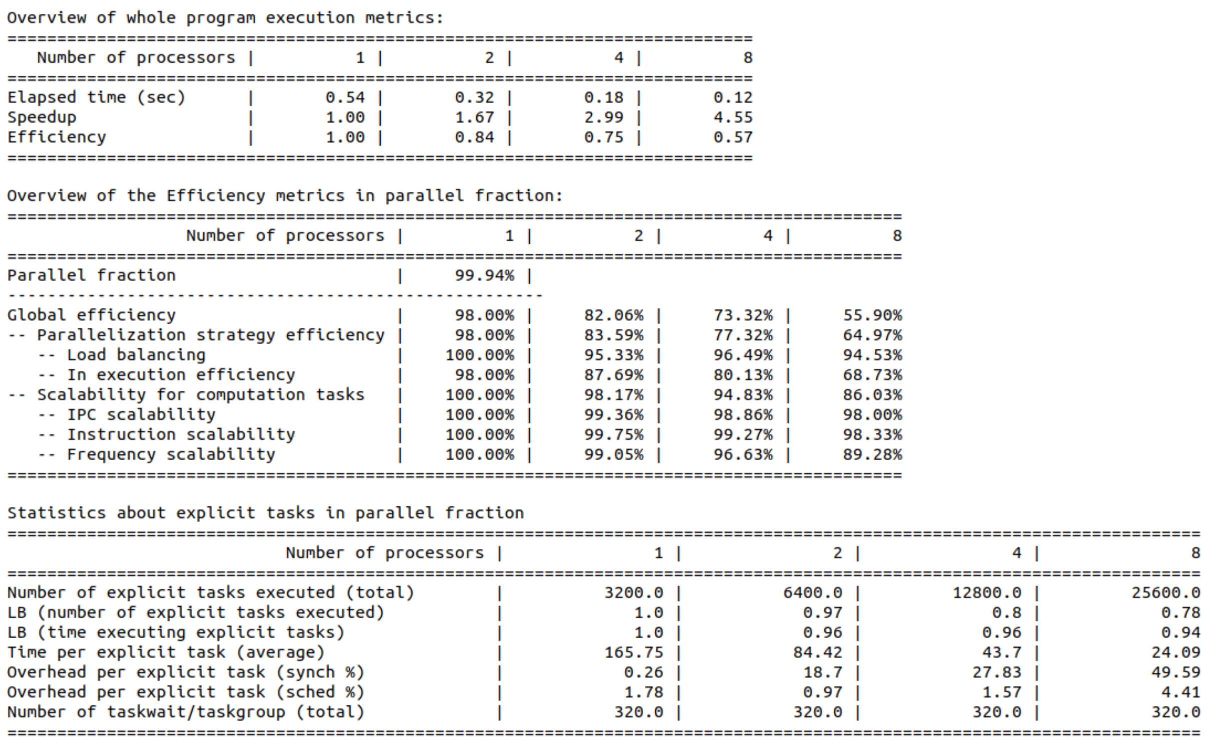


Figure 22. Time and speed-up plots of point taskloop strategy

Thanks to figure 22, we can visualize the difference in performance between the two variants of the code. We can see a huge improvement in both time and speedup.

Submitting the script provided with modelfactors, it gives us the same kind of three tables as seen before. Same procedure. First table of figure 25 shows that the efficiency parameter decreases like the previous strategy so we can’t see any improvement yet.



```
Overview of whole program execution metrics:
=========================================================================
   Number of processors |          1 |          2 |          4 |          8
=========================================================================
Elapsed time (sec)      |       0.54 |       0.32 |       0.18 |       0.12
Speedup                 |       1.00 |       1.67 |       2.99 |       4.55
Efficiency              |       1.00 |       0.84 |       0.75 |       0.57
=========================================================================

Overview of the Efficiency metrics in parallel fraction:
=========================================================================================
                 Number of processors |          1 |          2 |          4 |          8
=========================================================================================
Parallel fraction                     |     99.94% |
------------------------------------------------------
Global efficiency                     |     98.00% |     82.06% |     73.32% |     55.90%
-- Parallelization strategy efficiency |     98.00% |     83.59% |     77.32% |     64.97%
   -- Load balancing                  |    100.00% |     95.33% |     96.49% |     94.53%
   -- In execution efficiency         |     98.00% |     87.69% |     80.13% |     68.73%
-- Scalability for computation tasks  |    100.00% |     98.17% |     94.83% |     86.03%
   -- IPC scalability                 |    100.00% |     99.36% |     98.86% |     98.00%
   -- Instruction scalability         |    100.00% |     99.75% |     99.27% |     98.33%
   -- Frequency scalability           |    100.00% |     99.05% |     96.63% |     89.28%
=========================================================================================

Statistics about explicit tasks in parallel fraction
======================================================================================================================
                           Number of processors |                 1 |                 2 |                 4 |                 8
======================================================================================================================
Number of explicit tasks executed (total)       |            3200.0 |            6400.0 |           12800.0 |           25600.0
LB (number of explicit tasks executed)          |               1.0 |              0.97 |               0.8 |              0.78
LB (time executing explicit tasks)              |               1.0 |              0.96 |              0.96 |              0.94
Time per explicit task (average)                |            165.75 |             84.42 |              43.7 |             24.09
Overhead per explicit task (synch %)            |              0.26 |              18.7 |             27.83 |             49.59
Overhead per explicit task (sched %)            |              1.78 |              0.97 |              1.57 |              4.41
Number of taskwait/taskgroup (total)            |             320.0 |             320.0 |             320.0 |             320.0
======================================================================================================================
```

Figure 23. Screenshot of modelfactors.out of point taskloop strategy

The second table from figure 23 also shows that the “in execution efficiency” parameter decreases as we increase the number of threads, and we can see an improvement from the previous strategy in the “load balancing” parameter where it now stays on the same level. Figure 26 gives more sense to these results because we can clearly see that it is now more balanced than figure 18 (previous strategy) and also that now we have 25.600 tasks in total from the previous 102.400 tasks, meaning that OpenMP has opted for a grainsize of four.

Finally, in the third table from figure 23, the number of explicit tasks executed increases when the amount of processors increases as well. It is important to highlight the fact that the number of tasks stays the same in the point task strategy and this one is more focused on generating new ones when the threads increase. In addition, comparing these results with figure 15, the overheads per explicit task (synch%) have decreased by a lot, so we can say there is a huge improvement on this side, but still a must-solve problem. And the overhead per explicit task (sched%) stays low. If we want to talk about the number of taskwait/taskgroup generated, we can see that the results are supported by the figure 27. It is important to emphasize that the amount of taskwait/taskgroup generated takes a big part in the efficiency of the program, meaning that the lack of taskwait/taskgroups in the previous strategy causes a less efficient execution.

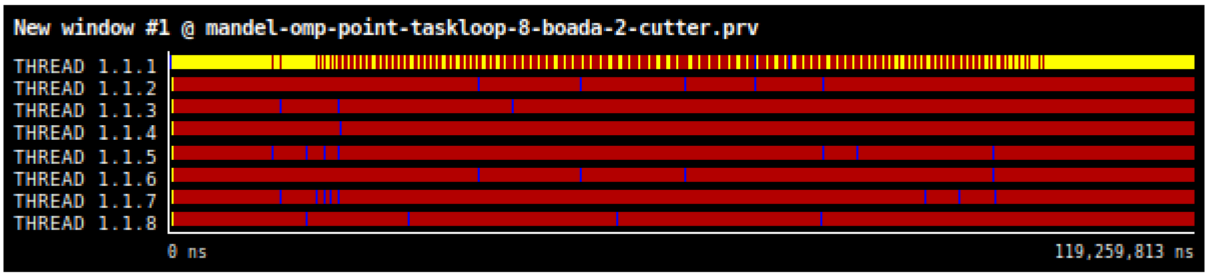


Figure 24. Cronogram of point taskloop strategy with eight threads

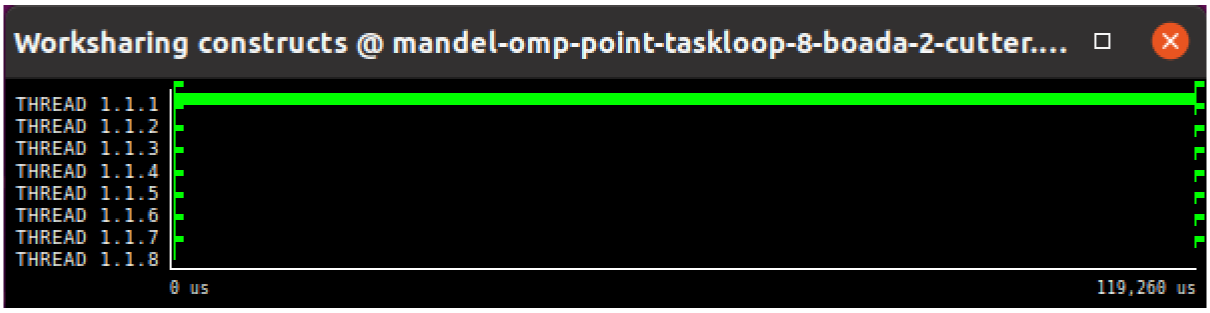
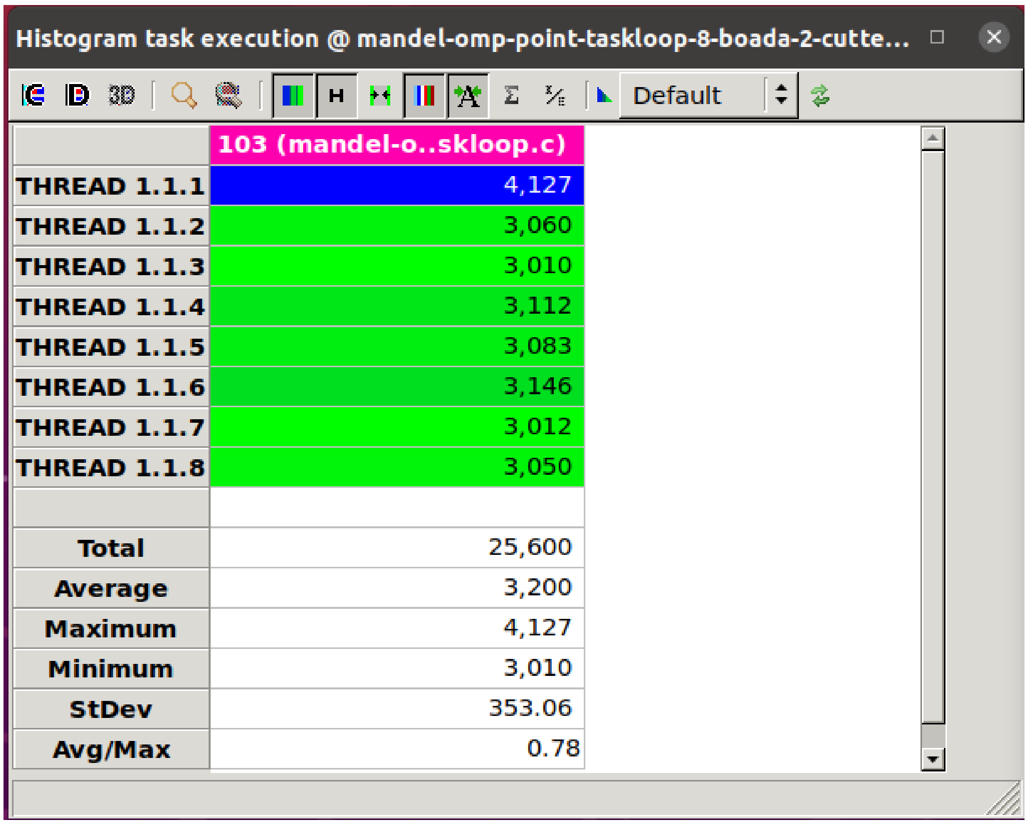


Figure 25. Worksharing constructs of point taskloop strategy with eight threads

To support our results, in figures 24 and 25, we now can say that, at first glance, it's a more computing-centered code with less synchronization and that thread 1 is in charge of creating tasks and the rest compute them.

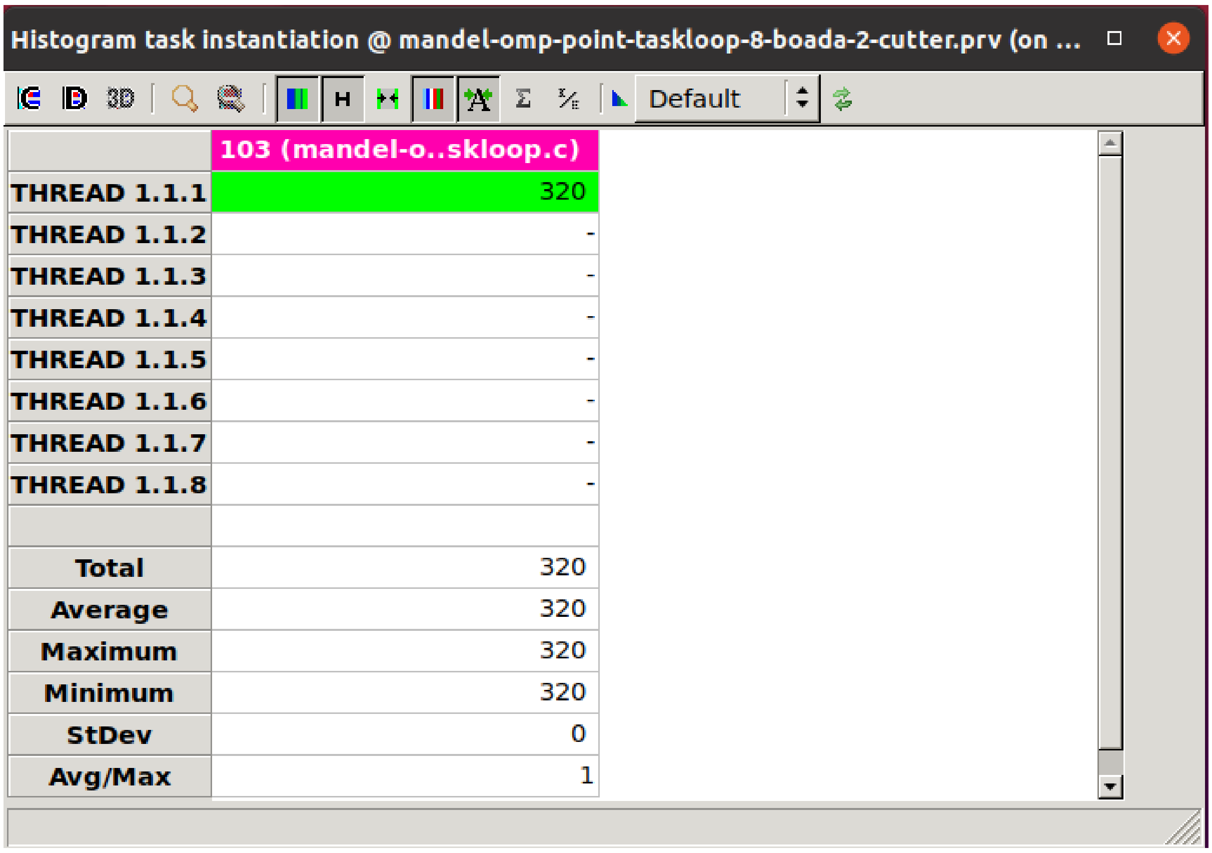


Histogram task execution @ mandel-omp-point-taskloop-8-boada-2-cutte...

Default

	103 (mandel-o..skloop.c)
THREAD 1.1.1	4,127
THREAD 1.1.2	3,060
THREAD 1.1.3	3,010
THREAD 1.1.4	3,112
THREAD 1.1.5	3,083
THREAD 1.1.6	3,146
THREAD 1.1.7	3,012
THREAD 1.1.8	3,050
Total	25,600
Average	3,200
Maximum	4,127
Minimum	3,010
StDev	353.06
Avg/Max	0.78

Figure 26. Histogram task execution of point taskloop strategy with eight threads



Histogram task instantiation @ mandel-omp-point-taskloop-8-boada-2-cutter.prv (on ...

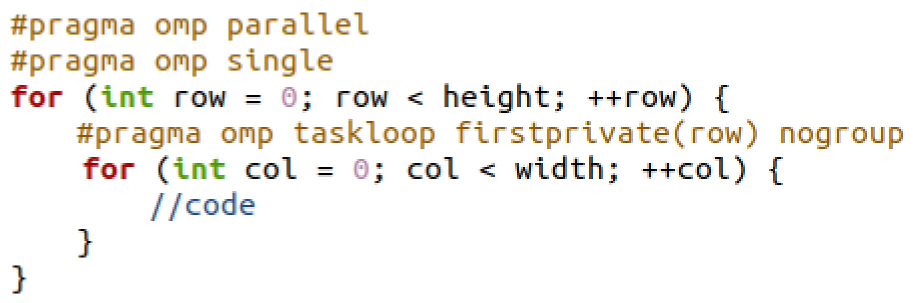
Default

	103 (mandel-o..skloop.c)
THREAD 1.1.1	320
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	-
Total	320
Average	320
Maximum	320
Minimum	320
StDev	0
Avg/Max	1

Figure 27. Histogram task instantiation of point taskloop strategy with eight threads

Point strategy with granularity control using taskloop nogroup

Now, to check another option of parallelization, we are going to modify our strategy including a nogroup clause so we can see if we can solve some deficiencies. Because all these barriers that we saw on the point taskloop cause a waste of time because it has to wait most of the times.



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

Figure 28. Fragment of the relevant code of mandel-omp.c using a point taskloop nogroup declaration

Now, we proceed to execute the new version of the program with 1 and 8 processors so we can see the execution time, as we did in previous sections. So we can compare it with the point taskloop strategy.

As we can see in the figure 29 and 30, the addition of the nogroup clause has caused an improvement of the execution time and speedup derived from the elimination of the restriction of waiting until all the tasks of each batch have finished. With the nogroup clause we have neutralized all this wasted time.

```
>sbatch ./submit-omp.sh mandel-omp 1
>sbatch ./submit-omp.sh mandel-omp 8
```

#Threads	Sequential 1 Thrd	1 Thread	8 Threads
Time	3.039749 seconds	3.258733 seconds	0.508274 seconds

Figure 29. Result table of the time execution with different number of threads in a row taskloop nogroup strategy

>sbatch ./submit-strong-omp.sh mandel-omp 1 12

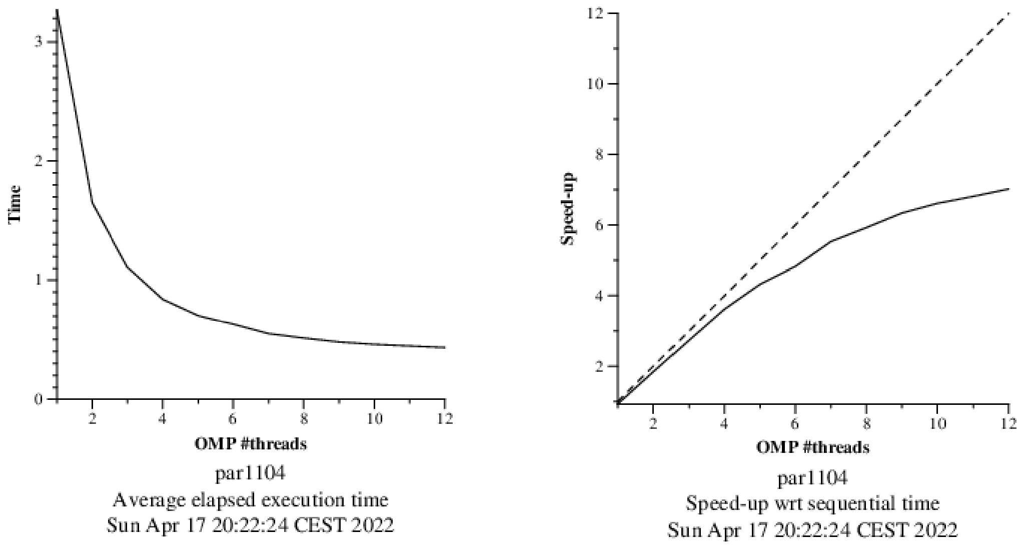
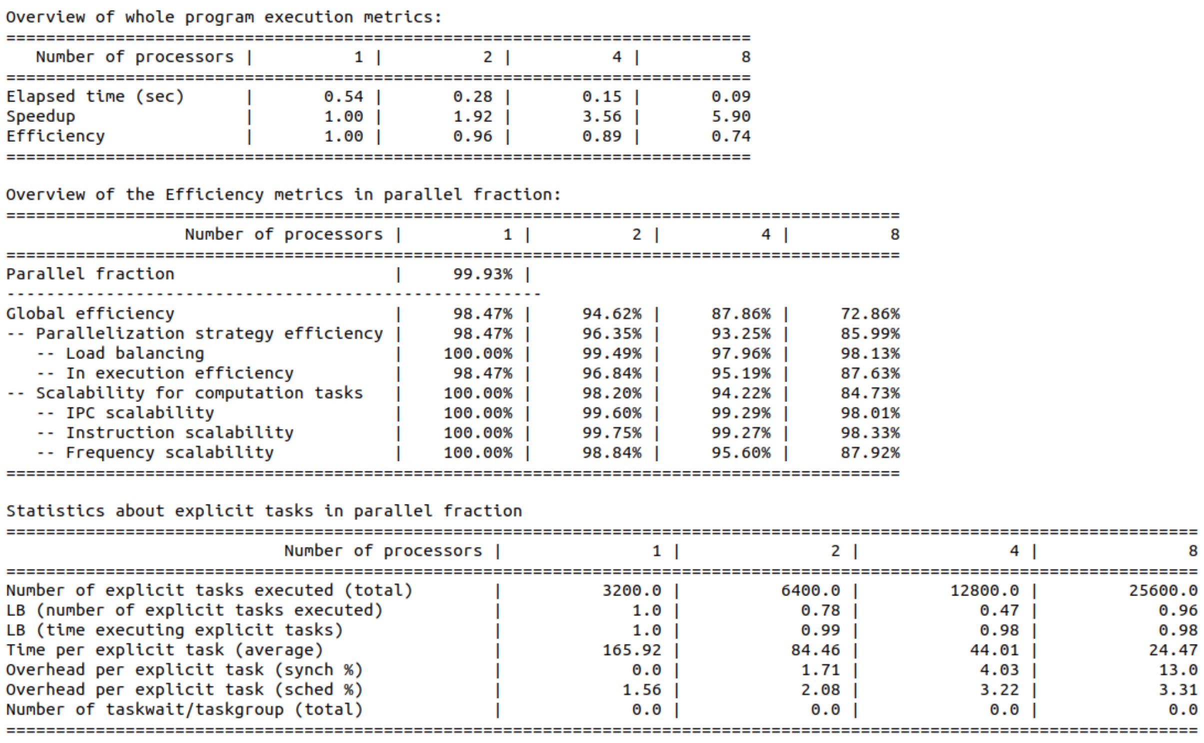


Figure 30. Time and speed-up plots of row taskloop nogroup strategy



```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |            1 |            2 |            4 |            8
===========================================================================
Elapsed time (sec)      |         0.54 |         0.28 |         0.15 |         0.09
Speedup                 |         1.00 |         1.92 |         3.56 |         5.90
Efficiency              |         1.00 |         0.96 |         0.89 |         0.74
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |            1 |            2 |            4 |            8
==========================================================================================
Parallel fraction                      |       99.93% |
-------------------------------------------------------
Global efficiency                      |       98.47% |       94.62% |       87.86% |       72.86%
-- Parallelization strategy efficiency |       98.47% |       96.35% |       93.25% |       85.99%
   -- Load balancing                   |      100.00% |       99.49% |       97.96% |       98.13%
   -- In execution efficiency          |       98.47% |       96.84% |       95.19% |       87.63%
-- Scalability for computation tasks   |      100.00% |       98.20% |       94.22% |       84.73%
   -- IPC scalability                  |      100.00% |       99.60% |       99.29% |       98.01%
   -- Instruction scalability          |      100.00% |       99.75% |       99.27% |       98.33%
   -- Frequency scalability            |      100.00% |       98.84% |       95.60% |       87.92%
==========================================================================================

Statistics about explicit tasks in parallel fraction
========================================================================================================================
                            Number of processors |                 1 |                 2 |                 4 |                 8
========================================================================================================================
Number of explicit tasks executed (total)        |            3200.0 |            6400.0 |           12800.0 |           25600.0
LB (number of explicit tasks executed)           |               1.0 |              0.78 |              0.47 |              0.96
LB (time executing explicit tasks)               |               1.0 |              0.99 |              0.98 |              0.98
Time per explicit task (average)                 |            165.92 |             84.46 |             44.01 |             24.47
Overhead per explicit task (synch %)             |               0.0 |              1.71 |              4.03 |              13.0
Overhead per explicit task (sched %)             |              1.56 |              2.08 |              3.22 |              3.31
Number of taskwait/taskgroup (total)             |               0.0 |               0.0 |               0.0 |               0.0
========================================================================================================================
```

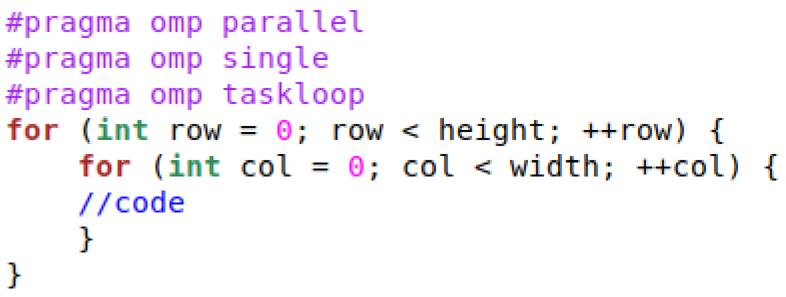
Figure 31. Screenshot of modelfactors.out of point taskloop no group strategy

Comparing figure 23 with figure 31, I would highlight that we can support the previous results that the execution of the code becomes faster and more efficient, we can also see that the overhead in the explicit tasks (synch %) has been considerably reduced and also that the number of taskwait/taskgroup becomes 0.

Row decomposition in OpenMP

For the final section of this report, we have to implement a row strategy in OpenMP, the best one if possible.

We are going to set the definition of each task using the clause taskloop but in a row strategy (row loop). In this case, we don’t need firstprivate or private clauses because the variables row and col are not yet declared.



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

Figure 32. Fragment of the relevant code of mandel-omp.c using a row taskloop declaration

For evaluation we are going to use the same procedure and executions that we did in the previous sections.

First, we will execute this version of the program using 1 and 8 processors so we can see its evolution. And after comparing these new values we will create an execution time and speedup plot from 1 to 12 processors to finally see how it develops.

```
>sbatch ./submit-omp.sh mandel-omp 1
>sbatch ./submit-omp.sh mandel-omp 8
```

#Threads	Sequential 1 Thrd	1 Thread	8 Threads
Time	3.039749 seconds	3.257216 seconds	0.465162 seconds

Figure 33. Result table of the time execution with different number of threads in a row taskloop strategy

As figure 33 shows, the execution time with 1 thread is almost the same as the one from the point taskloop declaration. They both are not the same level as the serial execution but we can clearly see some improvements from the first section of the report. The execution time with 8 threads has improved from the point taskloop declaration. So we can say that this definition and the point taskloop are equivalently good at parallelizing this problem. In fact, we can say that if we are aiming to execute it with a greater amount of threads, the optimal definition is the row taskloop one.
The following figure will show how this definition performs with different amounts of threads to have a better understanding of its performance.

```
>sbatch ./submit-strong-omp.sh mandel-omp 1 12
```

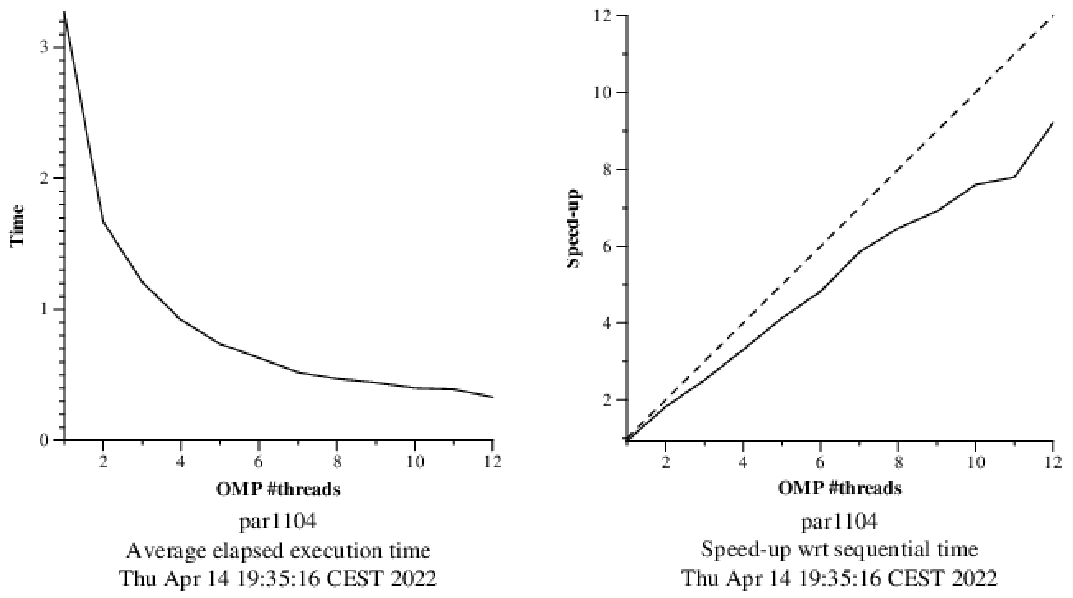
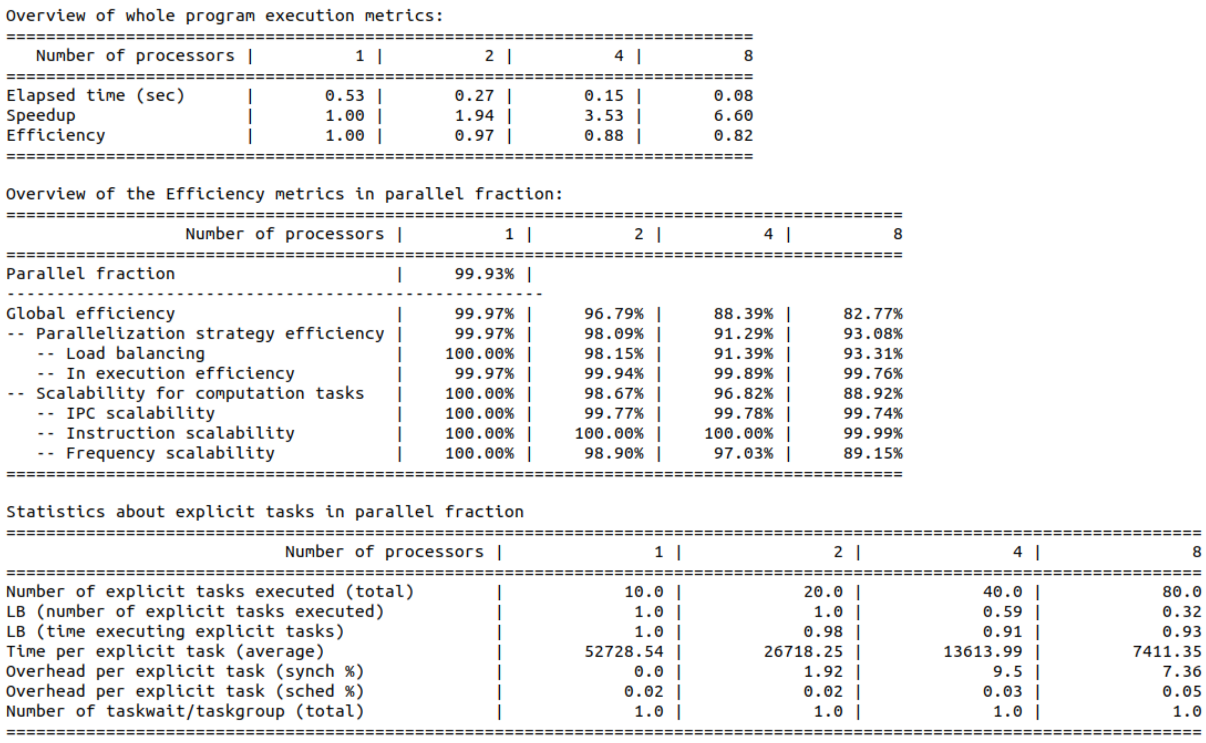


Figure 34. Time and speed-up plots of row taskloop strategy

We can clearly see how the execution time falls when more threads are working and consequently the speedup grows.

Now we are going to introduce some data generated by the script provided with modelfactors.py, so we can support our results and in a more visible way. And afterwards, some brief conclusions.



```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |          1 |          2 |          4 |          8
===========================================================================
Elapsed time (sec)      |       0.53 |       0.27 |       0.15 |       0.08
Speedup                 |       1.00 |       1.94 |       3.53 |       6.60
Efficiency              |       1.00 |       0.97 |       0.88 |       0.82
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
==========================================================================================
Parallel fraction                      |     99.93% |
------------------------------------------------------
Global efficiency                      |     99.97% |     96.79% |     88.39% |     82.77%
-- Parallelization strategy efficiency |     99.97% |     98.09% |     91.29% |     93.08%
   -- Load balancing                   |    100.00% |     98.15% |     91.39% |     93.31%
   -- In execution efficiency          |     99.97% |     99.94% |     99.89% |     99.76%
-- Scalability for computation tasks   |    100.00% |     98.67% |     96.82% |     88.92%
   -- IPC scalability                  |    100.00% |     99.77% |     99.78% |     99.74%
   -- Instruction scalability          |    100.00% |    100.00% |    100.00% |     99.99%
   -- Frequency scalability            |    100.00% |     98.90% |     97.03% |     89.15%
==========================================================================================

Statistics about explicit tasks in parallel fraction
========================================================================================================================
                            Number of processors |                 1 |                 2 |                 4 |                 8
========================================================================================================================
Number of explicit tasks executed (total)        |              10.0 |              20.0 |              40.0 |              80.0
LB (number of explicit tasks executed)           |               1.0 |               1.0 |              0.59 |              0.32
LB (time executing explicit tasks)               |               1.0 |              0.98 |              0.91 |              0.93
Time per explicit task (average)                 |          52728.54 |          26718.25 |          13613.99 |           7411.35
Overhead per explicit task (synch %)             |               0.0 |              1.92 |               9.5 |              7.36
Overhead per explicit task (sched %)             |              0.02 |              0.02 |              0.03 |              0.05
Number of taskwait/taskgroup (total)             |               1.0 |               1.0 |               1.0 |               1.0
========================================================================================================================
```

Figure 35. Screenshot of modelfactors.out of row taskloop strategy

At first glance, the differences between this strategy and the previous ones can be appreciated. It is obvious that it takes 10 times less time to execute than point task strategy and 3 times less than point taskloop strategy and its efficiency does not fall as it did with previous strategies.

Speaking explicitly about the second table in Figure 35, it can be observed that its overall efficiency remains stable, so we could conclude that there is no load imbalance.

As for the explicit tasks, this strategy executes 10 per thread, so the number is very small, making the average time per task very large compared to the previous strategies. It is also relevant to note that the overhead per explicit task (synch %) becomes a more coherent percentage.

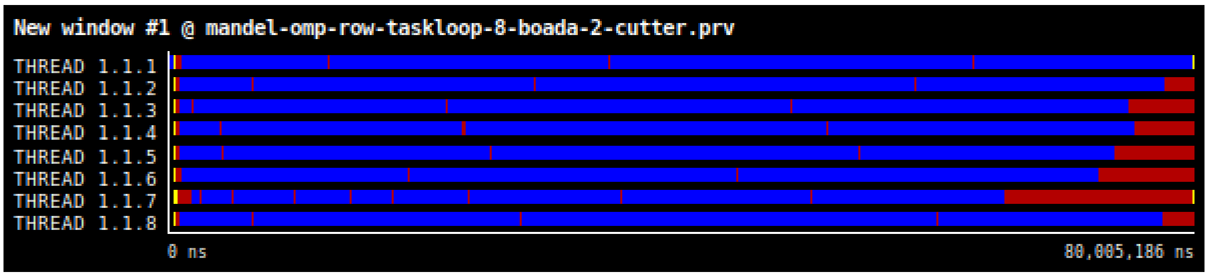


Figure 36. Cronogram of row taskloop strategy with eight threads

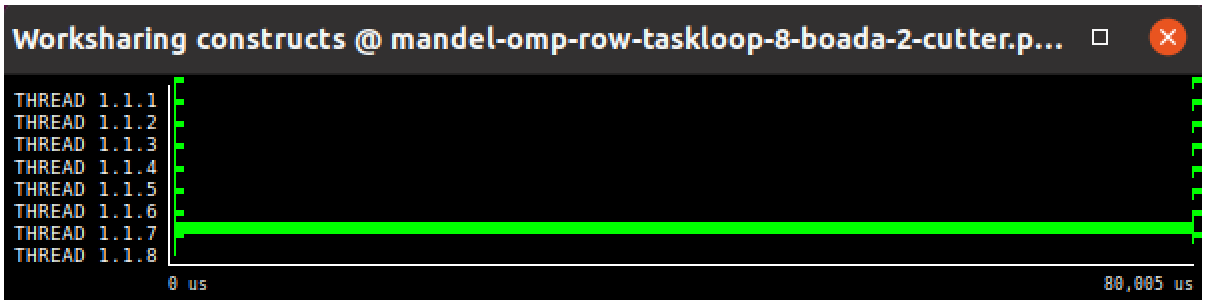
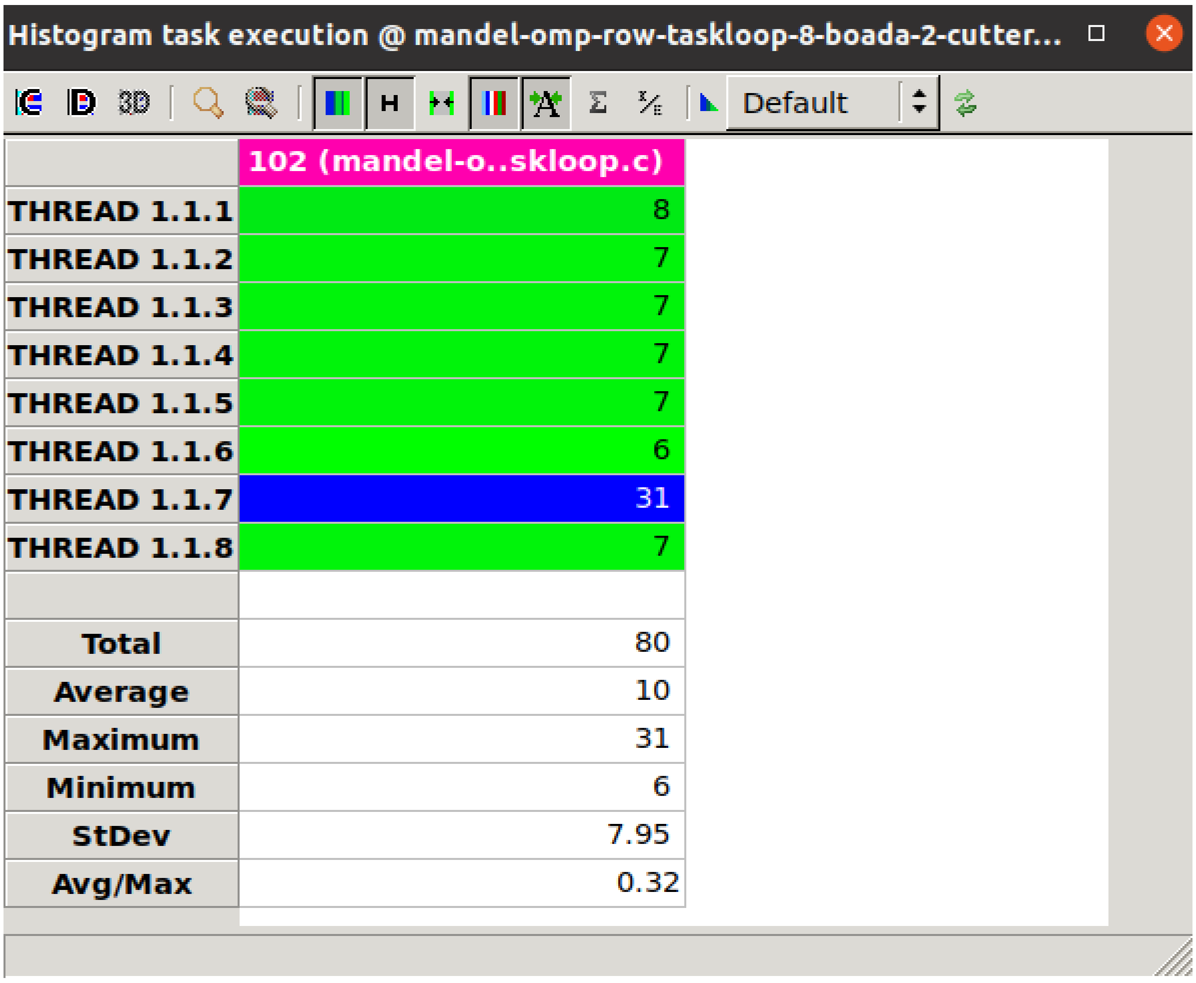


Figure 37. Worksharing constructs of row taskloop strategy with eight threads


Histogram task execution @ mandel-omp-row-taskloop-8-boada-2-cutter...
Default

	102 (mandel-o..skloop.c)
THREAD 1.1.1	8
THREAD 1.1.2	7
THREAD 1.1.3	7
THREAD 1.1.4	7
THREAD 1.1.5	7
THREAD 1.1.6	6
THREAD 1.1.7	31
THREAD 1.1.8	7
Total	80
Average	10
Maximum	31
Minimum	6
StDev	7.95
Avg/Max	0.32

Figure 38. Histogram task execution of row taskloop strategy with eight threads

If we want to support our modelfactors results, we can say that in figure 36, we can observe that the predominant regions are now the Running regions instead of the Synchronization ones of the firsts types of tasks. This is due to the fact that now we only have 80 tasks to synchronize (Figure 38) instead of the 102.400 or the 25.600 tasks we had on earlier versions. Less tasks imply less overheads of creation and synchronization.

Some conclusions

As we have seen, there are plenty of options to parallelise a program but it is up to us, the programmer, to finally think for the best one to aim for the best performance. In the case of the Mandelbrot set, after developing some definitions and strategies we could say that the row and taskloop declaration or the point taskloop one are the best methods because of its performance shown previously.

Optional

If we ever wanted to analyze the influence of task granularity in both Point and Row strategies we could rely on what we developed in the previous sections. That's why we are going to work with the taskloop versions of each strategy.

In order to do this section correctly, we have been given the script submit-numtasks-omp.sh. This script will execute the same 10.000 iterations (by default) but with the user parameter set as 800. This parameter will be used to set the number of tasks we want to execute on each execution, so we can analyze the granularity on boths strategies. This new parameter will help us to show a large range of tasks, so the greater the parameter is, a better sample shows.

To be able to do this, we will have to modify our current versions of each strategy adding the clause num_tasks(user_param) in the pragma omp instance.

```
>sbatch ./submit-numtasks-omp.sh mandel-omp 800
```

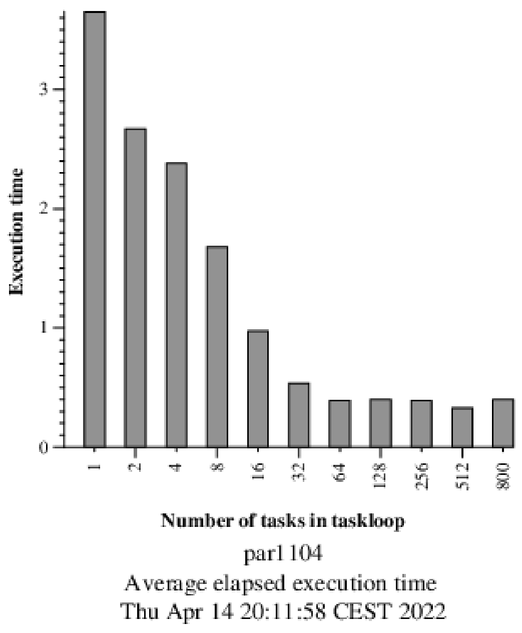


Figure 40. Execution time plot for the point-taskloop strategy with different number of tasks

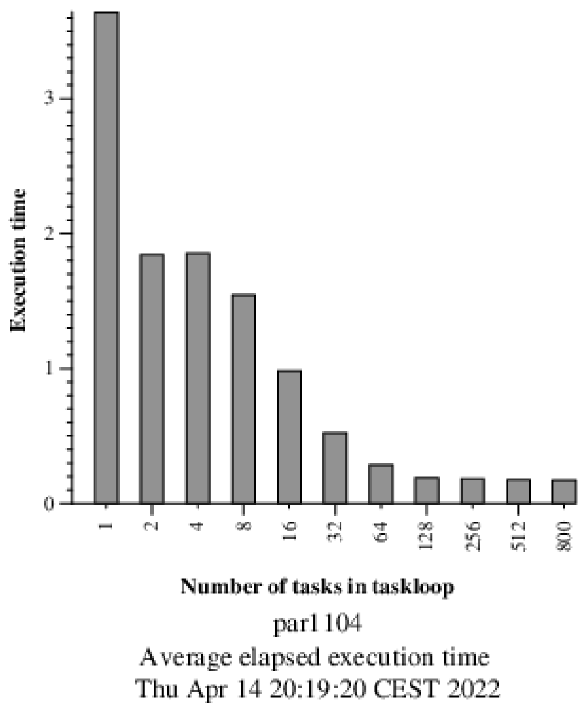


Figure 41. Execution time plot for the row-taskloop strategy with different number of tasks

Based on the results we can see in figures 40 and 41 we can say that both strategies perform similarly because both look almost the same. It should be noted that the row taskloop strategy performs better with a increasing number of tasks since we work from a very coarse granularity (1 taskloop) and the point strategy behaves oppositely, it performs worse with an increasing number of tasks since we depart from a very fine granular (800 taskloops).

In conclusion, If we have to choose one of the two strategies, then we should look at the number of tasks we want them to work on. From there, based on what is written in the previous paragraph, we can choose the strategy that fits the best our program.