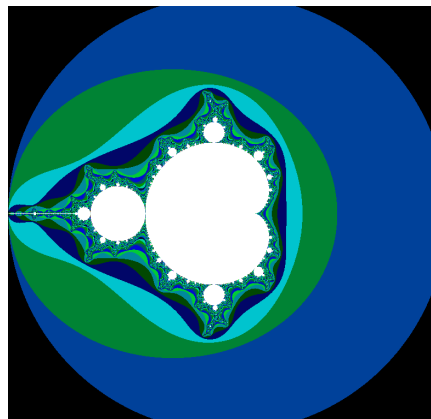


PAR Laboratory  
Assignment Lab 3:

**Iterative task  
decomposition with  
*OpenMP*:  
The Computation Of The  
Mandelbrot Set**



Guillem Dubé Quintín par3103  
Ricard Guixaró Tranco par3108  
Fall 2021-2022  
Date: 17/11/2022



UNIVERSITAT POLITÈCNICA  
DE CATALUNYA  
BARCELONATECH

# Index

<b>Index</b>	<b>1</b>
<b>1. Introduction</b>	<b>1</b>
<b>2. Task Decomposition analysis</b>	<b>3</b>
2.1. Task decomposition analysis with Tareador	3
a. Row Strategy	4
b. Point Strategy	9
<b>3. Implementing task decompositions in OpenMP</b>	<b>12</b>
3.1. Point strategy implementation using task	12
3.2. Point strategy with granularity control using taskloop	19
3.3. Row strategy implementation	25
<b>5. Conclusions</b>	<b>29</b>

# 1. Introduction

We are going to explore the tasking model in *OpenMP* to express **iterative task decompositions**, using the computation of the *Mandelbrot set*. But before that, we will start by exploring the most appropriate ones by using *Tareador*.

To start, we identify the loops that traverse the two dimensional space and the loop that checks if a point belongs to the *Mandelbrot set* in *lab3/mandel-seq.c*, inside the method *void mandelbrot()*.

We compile the sequential version: ***make madel-seq***. Next, we familiarize ourselves with the different output options we can use when running the program:

- ***./mandel-seq***: the *Mandelbrot set* is computed, but no output (image) is displayed or written to the disc.
- ***./mandel-seq -h***: the histogram for the values in the *Mandelbrot set* is also computed.
- ***./mandel-seq -d***: the *Mandelbrot set* is displayed.
- ***./mandel-seq -o***: the values for *Mandelbrot set* and/or histogram are written to disk, creating the *output\_seq.out* file.

In order to have a reference of the sequential execution, we execute the sequential version with *./mandel-seq -h -i 10000 -o* to get its execution time and the output file, to be used later to check the correctness of the different parallel versions we write. As we can see, the total execution time of the sequential version with a maximum of 10000 iterations is 3.048128s.

```
par3108@boada-1:~/lab3$ ./mandel-seq -h -i 10000 -o

Computation of the Mandelbrot set with:
  center = (0, 0)
  size = 2
  maximum iterations = 10000

Total execution time (in seconds): 3.048128

Mandelbrot set: Computed
Histogram for Mandelbrot set: Computed
Writing output file to disk: output_seq.out
```

Figure 1: part 1 of output of the command: *./mandel-seq -h -i 10000 -o*

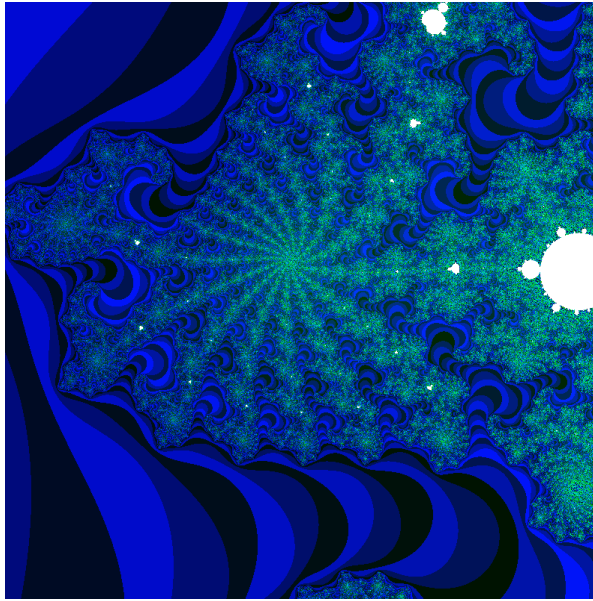


Figure 2: part 2 of the output of the command: `./mandel-seq -h -i 10000 -o`

## 2. Task Decomposition analysis

### 2.1. Task decomposition analysis with Tareador

Two different alternatives to generate tasks will be explored in this laboratory assignment, with different granularities\*:

- **Row decomposition:** a task will correspond with the computation of one or more (consecutive) rows of the *Mandelbrot set* (it will correspond with the distribution of iterations of the loop indexed by the row induction variable). Therefore, the size of each task will be bigger than the ones using the *Point decomposition method*.
- **Point decomposition:** a task will correspond with the computation of one or more (consecutive) points (row, col) in the same row of the *Mandelbrot set* (it will correspond with the distribution of iterations of the loop indexed by the col induction variable, for a fixed value of row).

To begin, we'll study the main characteristics of both forementioned alternatives (Row and Point decompositions) with *Tareador* and a granularity of one iteration per task.

## a. Row Strategy

We edit the sequential *mandel-tar.c* code partially instrumented in order to analyze the potential parallelism for the Row strategy, with granularity of one iteration of the row.

Loop per task, adding the lines:

```
- tareador_start_task("row");  
- tareador_end_task("row");
```

inside the outer loop (we can see the changes on the right).

We save the new file as *mandel-tar\_row.c* and add the appropriate target in the Makefile.

Then we compile it with ***make mandel-tar\_row*** and execute it interactively with the ***./run\_tareador.sh mandel-tar\_row*** and no execution options.

```
void mandelbrot(int height, int width, double real_min, double imag_min,  
               double scale_real, double scale_imag, int maxiter, int **output) {  
  
    // Calculate points and generate appropriate output  
    for (int row = 0; row < height; ++row) {  
        tareador_start_task("row");  
        for (int col = 0; col < width; ++col) {  
            complex z, c;  
  
            z.real = z.imag = 0;  
  
            /* Scale display coordinates to actual region */  
            c.real = real_min + ((double) col * scale_real);  
            c.imag = imag_min + ((double) (height-1-row) * scale_imag);  
            /* height-1-row so y axis displays  
             * with larger values at top  
             */  
  
            // Calculate z0, z1, .... until divergence or maximum iterations  
            int k = 0;  
            double lengthsq, temp;  
            do {  
                temp = z.real*z.real - z.imag*z.imag + c.real;  
                z.imag = 2*z.real*z.imag + c.imag;  
                z.real = temp;  
                lengthsq = z.real*z.real + z.imag*z.imag;  
                ++k;  
            } while (lengthsq < (N*N) && k < maxiter);  
  
            output[row][col]=k;  
  
            if (output2histogram) histogram[k-1]++;  
  
            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);  
                }  
            }  
        }  
        tareador_end_task("row");  
    }  
}
```

Figure 3: code *mandel-tar.c*

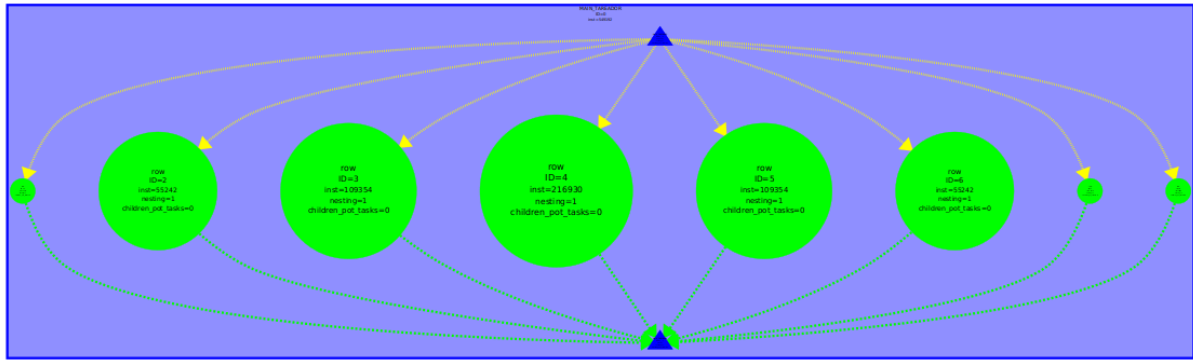


Figure 4: `./run-tareador.sh mandel-tar`

As we can see, there are no dependencies between the tasks, meaning the code is in parallel. We can also see that there is a considerable load imbalance when analyzing the parallel execution (with 8 CPUs) - the central tasks take longer than the corner ones. That's because of the do-while inside the row loop.



Figure 5: Task view of the mandel-tar

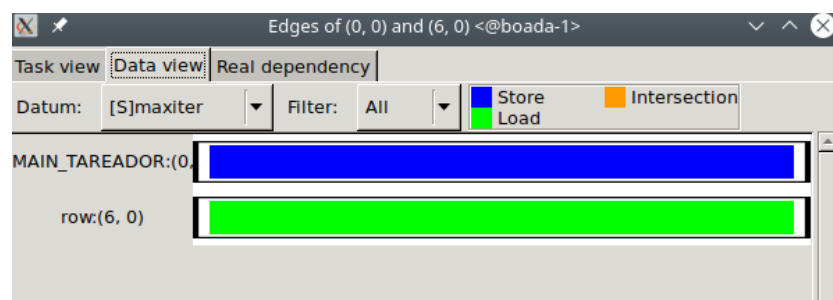


Figure 6: Data view of the code mandel-tar

- Next we will execute the binary with the -d option `./run-tareador.sh mandel-tar -d`.

We can see that, same as before, there is load imbalance due to the different number of instructions in each task, but in this case each task depends on the one before.

The -d command makes the execution of the code in sequential mode, so all dependencies are executed by the same thread.

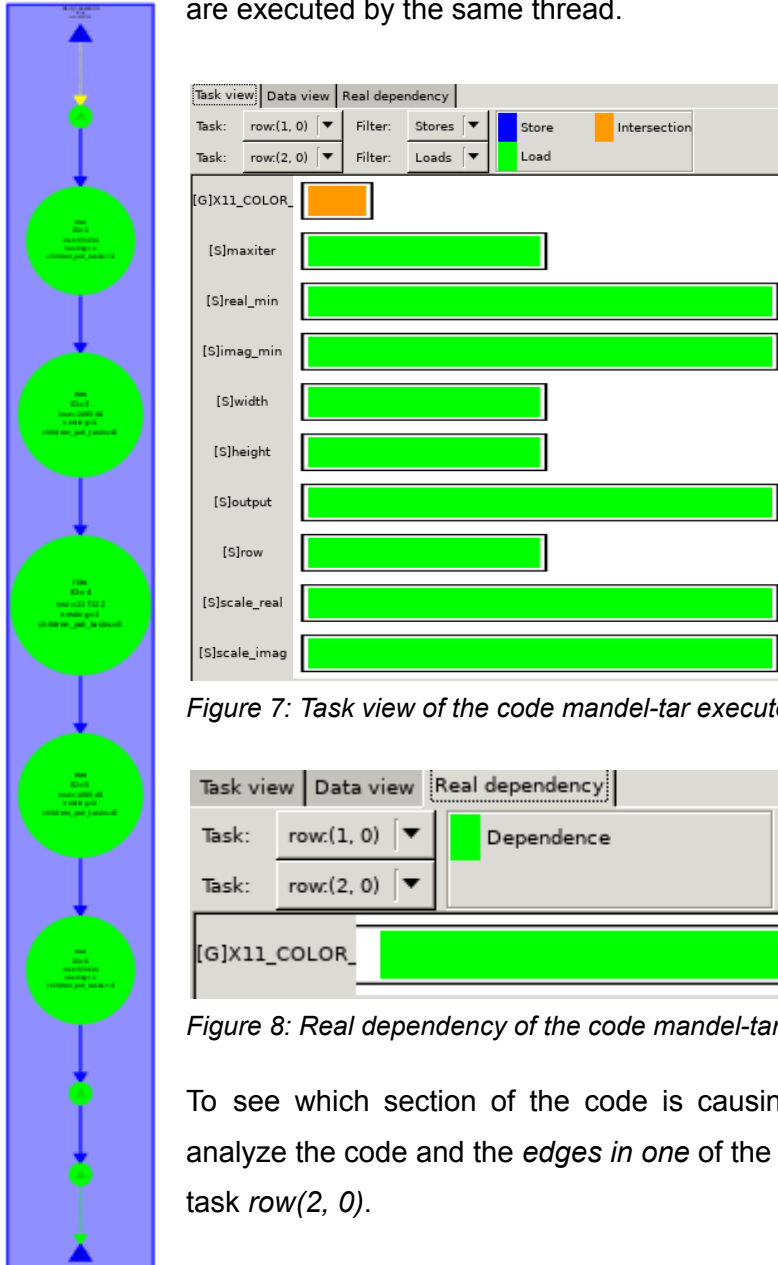


Figure 7: Task view of the code mandel-tar executed with the -d option.

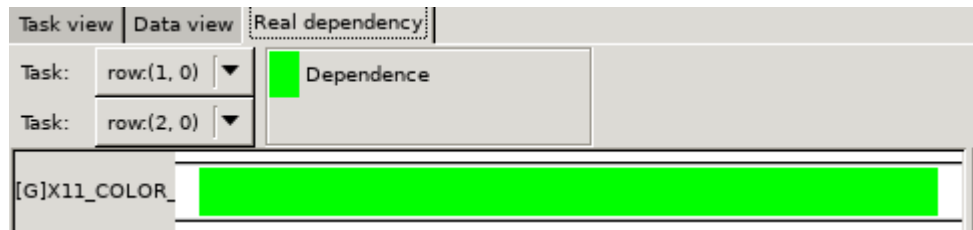


Figure 8: Real dependency of the code mandel-tar executed with the -d option.

To see which section of the code is causing the serialization of tasks we analyze the code and the *edges in one* of the tasks, for example task `row(2, 0)`.

Figure 9: diagram of the `./run-tareador.sh mandel-tar -d`.

We see that it depends on task `row(2,0)` and the variable causing the dependence is `[G]X11_COLOR_fake`. We can't truly see it in figure 7, but if we look at the *dependency view*

window, Figure 8, we can see all the possible variables that cause dependencies. So we can see it's related to the `_X11` display.

The only place `histogram` is used in is `void mandelbrot()` is the section of the code where the X11 display settings are set (now we understand the dependence analyzed above), meaning that the section of the code that should be protected when parallelizing with `OpenMP` is:

```
if (output2histogram) histogram[k-1]++;
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 10: `mandel-tar.c`, section of the code that must be protected because it's causing dependencies is "`histogram[k-1]++`".

This dependency can be protected with `#pragma omp critical`.

- Next we execute the binary with the `-h` option: `./run_tareador.sh mandel-tar_row -h`

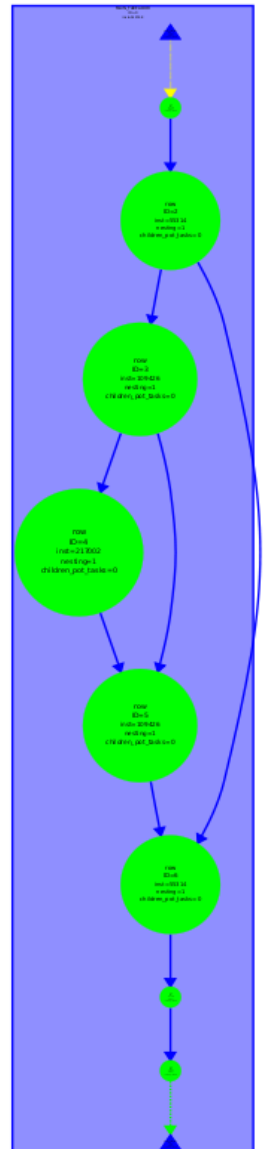
In this case a few dependencies are added, but the execution order of the tasks will be the same as in the `-d` version, and they will be executed sequentially, too.

Each chain of tasks in the task graph represents the tasks with the same value of `k` (iteration number when divergence is accomplished,  $k < maxiter$ ).

The load imbalance is the same as in both previous versions.

The variable that causes the dependencies is `histogram`. We confirm it by analyzing the edges in and the code, same as before.

Figure 11: execution of the command `./run-tareador.sh mandel-tar -h`





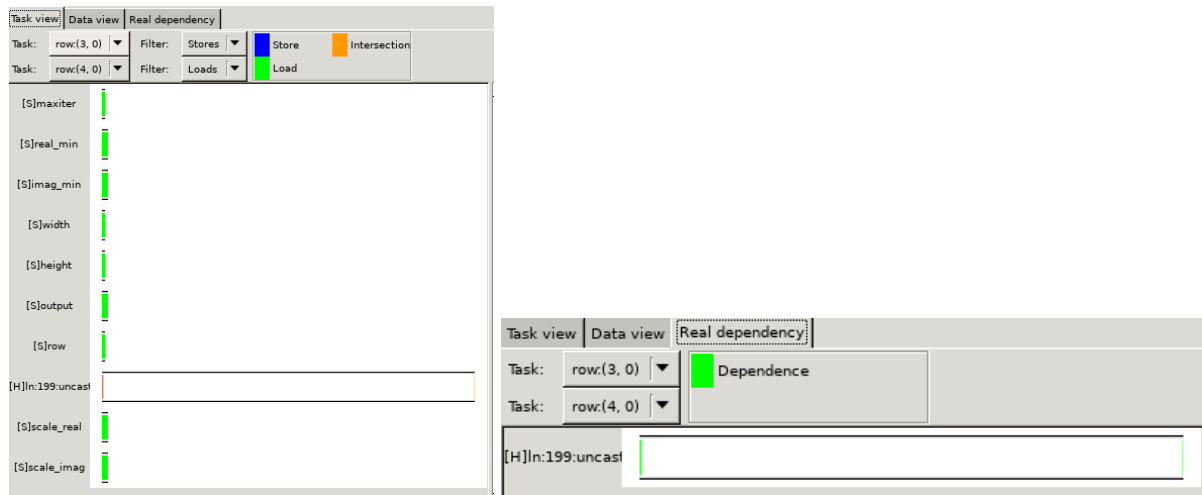


Figure 12,13: Task view and Real Dependency tabs from Mandel-tar executed with the -h option.

It could happen that many tasks access the same element of the variable histogram, that's why it creates a dependency.

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

Figure 14: variable histogram creating dependencies.

As a solution we could use a firstprivate clause, so the threads don't access the same variable, we also could use a `#pragma omp atomic`.

## b. Point Strategy

Same as before but now using *Point Strategy*. We edit the sequential mandel-tar.c code, adding the lines `tareador_start_task("point");` and `tareador_end_task("point");`.

```
void mandelbrot(int height, int width, double real_min, double imag_min,
               double scale_real, double scale_imag, int maxiter, int **output) {
    // Calculate points and generate appropriate output
    for (int row = 0; row < height; ++row) {
        for (int col = 0; col < width; ++col) {
            tareador_start_task("point");
            complex z, c;

            z.real = z.imag = 0;

            /* Scale display coordinates to actual region */
            c.real = real_min + ((double) col * scale_real);
            c.imag = imag_min + ((double) (height-1-row) * scale_imag);
            /* height-1-row so y axis displays
             * with larger values at top
             */

            // Calculate z0, z1, .... until divergence or maximum iterations
            int k = 0;
            double lengthsq, temp;
            do {
                temp = z.real*z.real - z.imag*z.imag + c.real;
                z.imag = 2*z.real*z.imag + c.imag;
                z.real = temp;
                lengthsq = z.real*z.real + z.imag*z.imag;
                ++k;
            } while (lengthsq < (N*N) && k < maxiter);

            output[row][col]=k;

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

            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);
                }
            }
            tareador_end_task("point");
        }
    }
}
```

Figure 15: code mandel-tar with point decomposition analysis with Tareador

After executing the command `./run-tareador mandel-tar`

We can see that there are no dependencies and a heavy load imbalance, just like in the *Row, no execution options* case (even though the imbalance in this, *Point, no execution options* case, is much bigger) by analyzing the task dependence graph (*Photo 5*) and simulation with 64 CPUs (*Figure 16*) seen below:

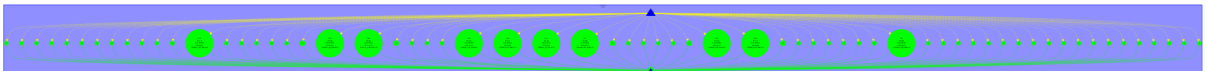


Figure 16: mandel-tar code with the point decomposition analysis with Tareador



Figure 17: Task view and Real dependency analysis with Tareador.

We now execute the binary with the -d option: `./run-tareador.sh mandel-tar -d`.

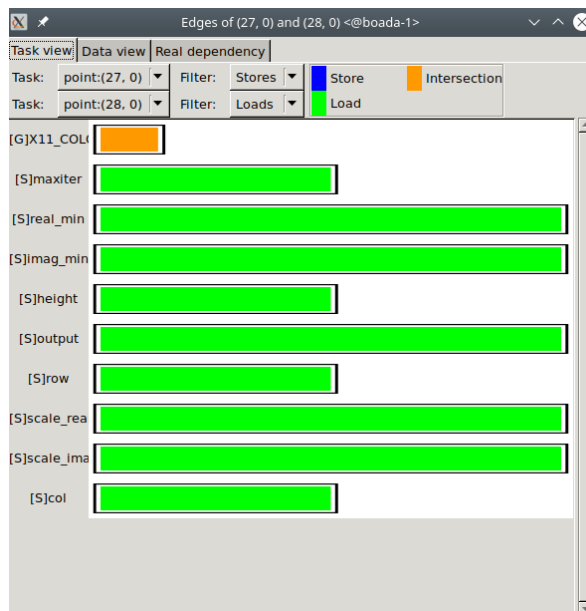


Figure 18: Task view of the code mandel-tar executed with the -d option.

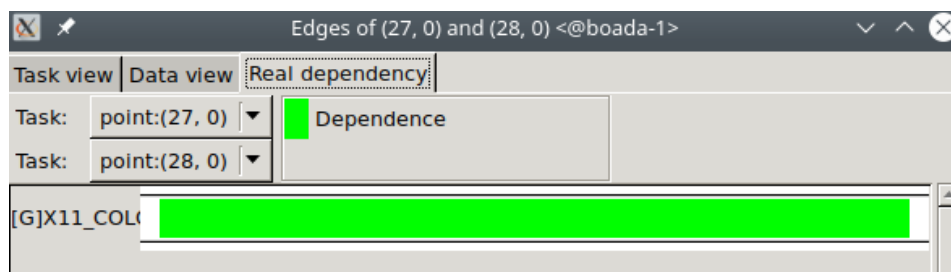


Figure 19: Real dependency of the code mandel-tar executed with the -d option.

Figure 20: mandel-tar code with the point decomposition analysis with Tareador

We can see that, same as before, there is load imbalance and each task depends on the one before. We observe that the characteristics are the same as in the Row *-d* option, but in this case there are 64 tasks, not 8, because the task is created inside both loops.

The section of the code causing the serialization of the tasks is also the same as in Row, *-d* option:

It will also be protected with a `#pragma omp critical` clause.

```
if (output2display) {
    /* Wait for user response, then exit program */
    if (setup_return == EXIT_SUCCESS) {
        interact(display, &win, width, height,
            real_min, real_max, imag_min, imag_max);
    }
    return EXIT_SUCCESS;
}
```

Figure 20: *mandel-tar.c*, section of the code that must be protected.

Finally, we execute the binary with the *-h* option: `./run-tareador mandel-tar -h`

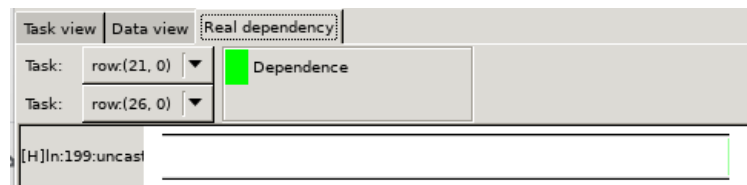


Figure 21: Real dependency of the code *mandel-tar* executed with the *-h* option.

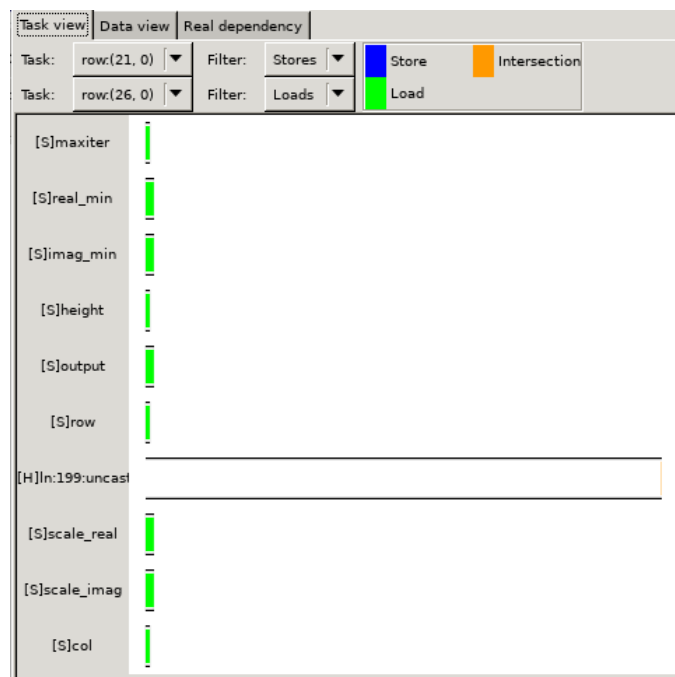


Figure 22, 23: Task view of the code *mandel-tar* executed with the *-d* option and *mandel-tar* code with the point decomposition analysis with Tareador.

## 3. Implementing task decompositions in *OpenMP*

### 3.1. Point strategy implementation using task

Now we are going to modify the file *mandel-omp.c* according to the dependencies that we found previously:

```
output[row][col]=k;
if (output2histogram) {
    #pragma omp atomic
    histogram[k-1]++;
}
if (output2display) {
    #pragma omp critical
    {
        /* 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);
        }
    }
}
```

We applied both `#pragma omp atomic` and `#pragma omp critical` to protect the dependencies in the code above.

Then, we execute two times the binary obtained after compiling with 1 and 2 threads. The resulting images are the following.

**OMP\_NUM\_THREADS=1 ./mandel-omp -d -h -i 10000**

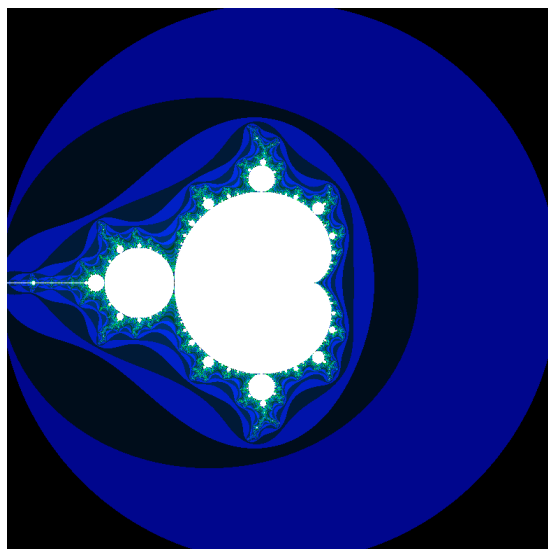
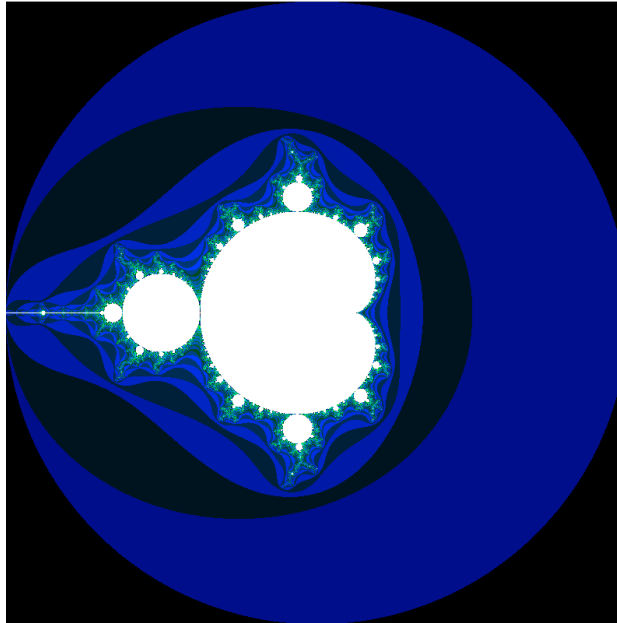


Figure 24: Execution of *mandel-omp* with 1 thread

**OMP\_NUM\_THREADS=2 ./mandel-omp -d -h -i 10000**



*Figure 25: Execution of mandel-omp with 2 threads*

As we can see in both images, the case is the same, meaning the parallelization is well done.

Now we will measure the decrease in time due to the parallel execution.

We make sure that the submit-omp.sh script contains the option -h -i -1000 -o, because we will need to make sure that the output files we obtain are correct.

**sbatch ./submit-omp.sh mandel-omp 1**

```
par3108@boada-1:~/lab33$ cat time-mandel-omp-1-boada-2
4.31user 0.01system 0:17.38elapsed 24%CPU (0avgtext+0avgdata 7152maxresident)k
144inputs+5088outputs (1major+1028minor)pagefaults 0swaps
```

*Figure 26: Time mandel-omp with 1 thread*

And total execution time = 17.23 s

**sbatch ./submit-omp.sh mandel-omp 8**

```
par3108@boada-1:~/lab33$ cat time-mandel-omp-8-boada-3
35.73user 0.80system 0:05.23elapsed 698%CPU (0avgtext+0avgdata 8344maxresident)k
144inputs+5088outputs (1major+3417minor)pagefaults 0swaps
```

*Figure 27: Time mandel-omp with 8 threads*

And total execution time = 4.99 s

Next, we check that the execution times and speed-ups are correct.

**sbatch submit-strong-omp.sh**

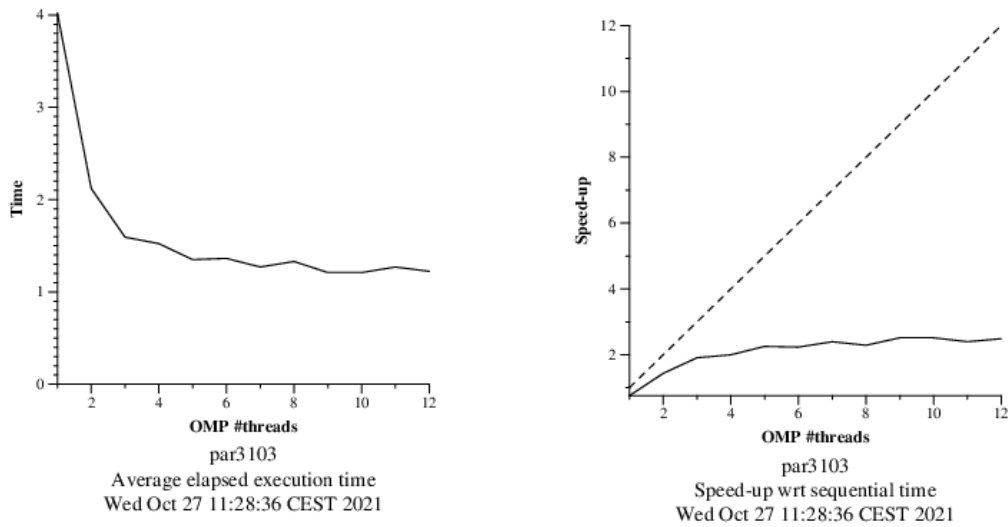


Figure 28: Scalability plot

From the pictures above, we can see that as we use more threads, we also reduce the execution time. But eventually it stops improving and stays around ~1.5 seconds.

Therefore, the scalability is not as good as we'd like it to be, because both execution times and speed-ups are ~constant once we reach ~6 threads.

Finally, we submit the binary to the *submit-extrae-sh* script in order to better understand how the execution actually works.

**sbatch submit-extrae.sh mandel-omp 8**

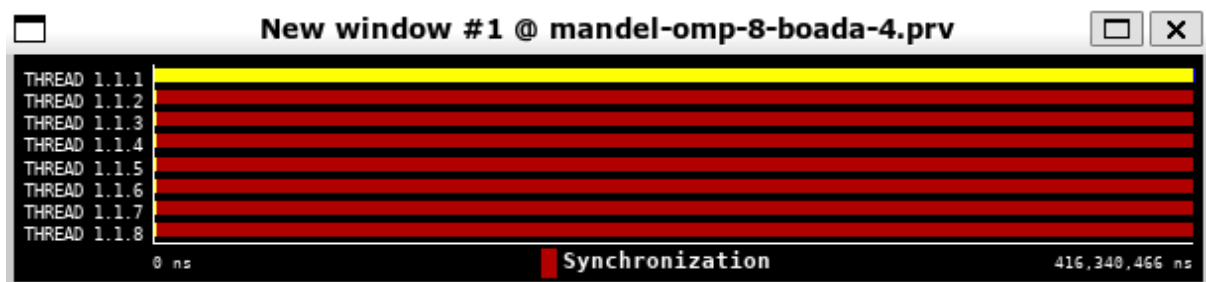


Figure 29: Execution of *mandel-omp.c* with Point Strategy using task and 8 threads

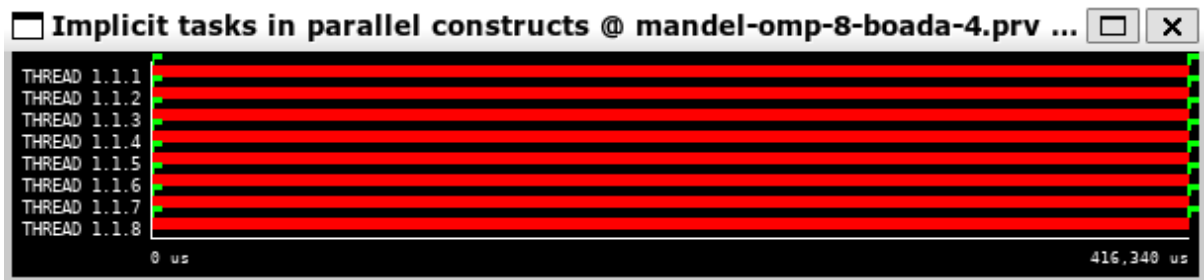


Figure 30: Implicit tasks originated in the parallel construct

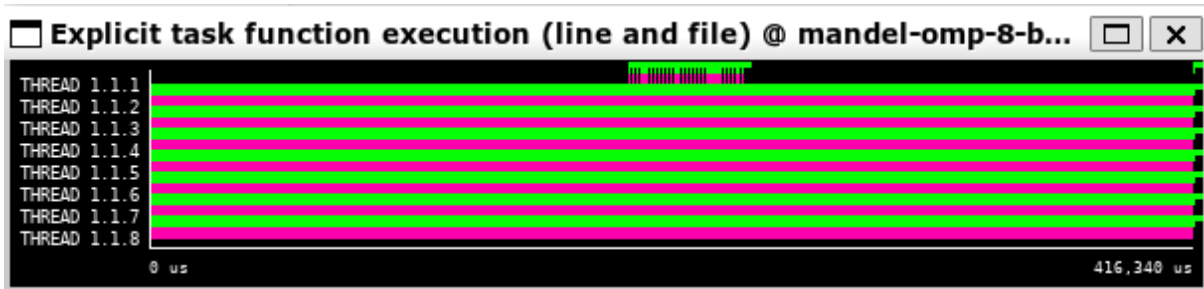


Figure 31: Explicit tasks execution

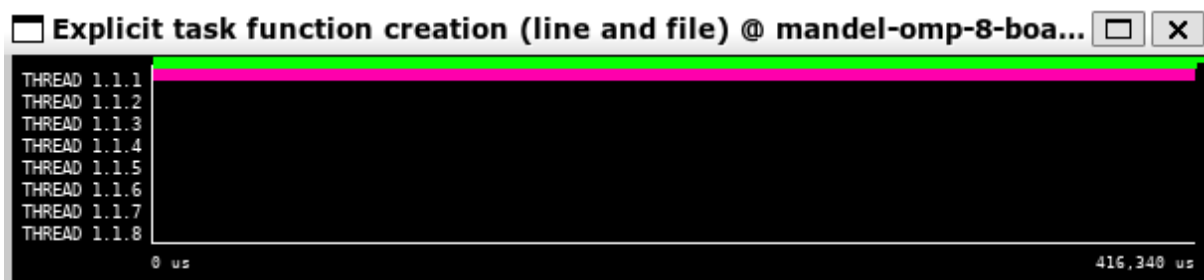


Figure 32: Explicit tasks creation

What we can see in the pictures above is that one thread creates all the tasks and the other ones just execute the tasks.



105 (mandel-omp.c, mandel-omp)	
THREAD 1.1.1	101
THREAD 1.1.2	14,132
THREAD 1.1.3	14,885
THREAD 1.1.4	14,288
THREAD 1.1.5	15,309
THREAD 1.1.6	14,216
THREAD 1.1.7	15,127
THREAD 1.1.8	14,342
Total	102,400
Average	12,800
Maximum	15,309
Minimum	101
StDev	4,817.84
Avg/Max	0.84

Figure 33: Explicit tasks creation and execution profile

From the previous figure, we see the statistics of the executed tasks, and we can obtain the number of tasks created/executed (102,400) and how are they distributed among all 8 threads.

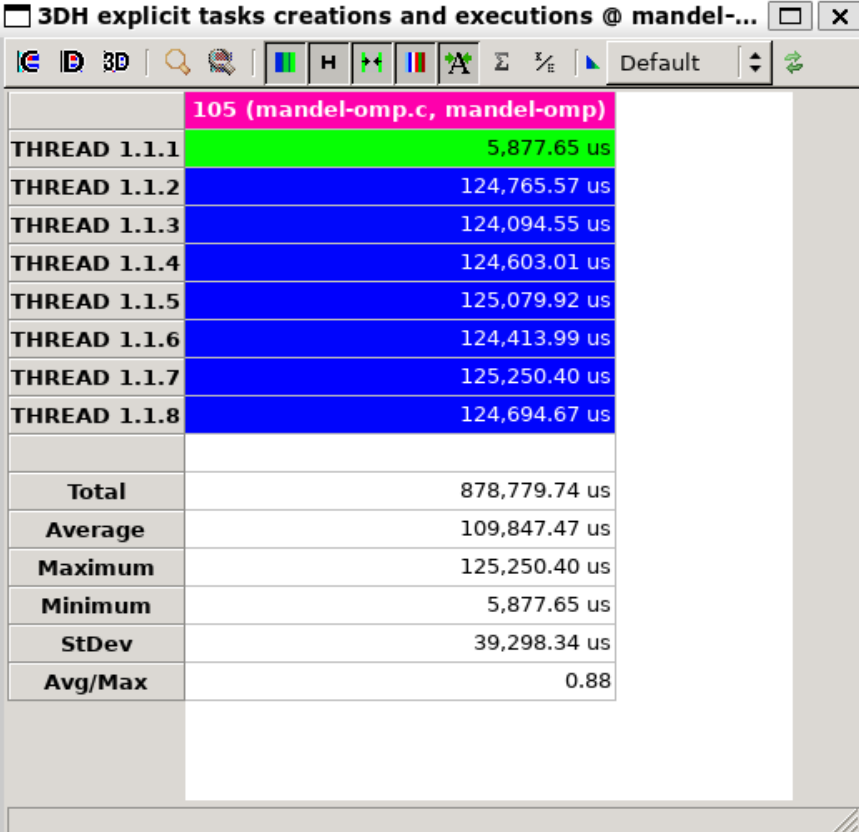
105 (mandel-omp.c, mandel-omp)	
THREAD 1.1.1	58.19 us
THREAD 1.1.2	8.83 us
THREAD 1.1.3	8.34 us
THREAD 1.1.4	8.72 us
THREAD 1.1.5	8.17 us
THREAD 1.1.6	8.75 us
THREAD 1.1.7	8.28 us
THREAD 1.1.8	8.69 us
Total	117.98 us
Average	14.75 us
Maximum	58.19 us
Minimum	8.17 us
StDev	16.42 us
Avg/Max	0.25

Figure 34: Explicit tasks creation and execution profile

In order to check if the load is well-balanced, we change the metric to show the average time spent executing a task by each thread. We see that only the thread 1.1.1 creates tasks, while the others execute them. The average time spent in each task is almost identical in all the threads (the thread 1.1.1 is creating the tasks, and has a higher average time).

In the next picture (*Figure 35*), we change again the metric to show this time the total time spent creating/executing the tasks, and we can conclude that the execution of tasks is well-balanced.

The total time spent creating/executing tasks is much more important than the number of tasks created, because it allows us to check whether the load is well-balanced or not, while the number of tasks doesn't help us determine it (one task could be more extensive than another).



105 (mandel-omp.c, mandel-omp)	
THREAD 1.1.1	5,877.65 us
THREAD 1.1.2	124,765.57 us
THREAD 1.1.3	124,094.55 us
THREAD 1.1.4	124,603.01 us
THREAD 1.1.5	125,079.92 us
THREAD 1.1.6	124,413.99 us
THREAD 1.1.7	125,250.40 us
THREAD 1.1.8	124,694.67 us
Total	878,779.74 us
Average	109,847.47 us
Maximum	125,250.40 us
Minimum	5,877.65 us
StDev	39,298.34 us
Avg/Max	0.88

*Figure 35: Explicit tasks creation and execution profile*

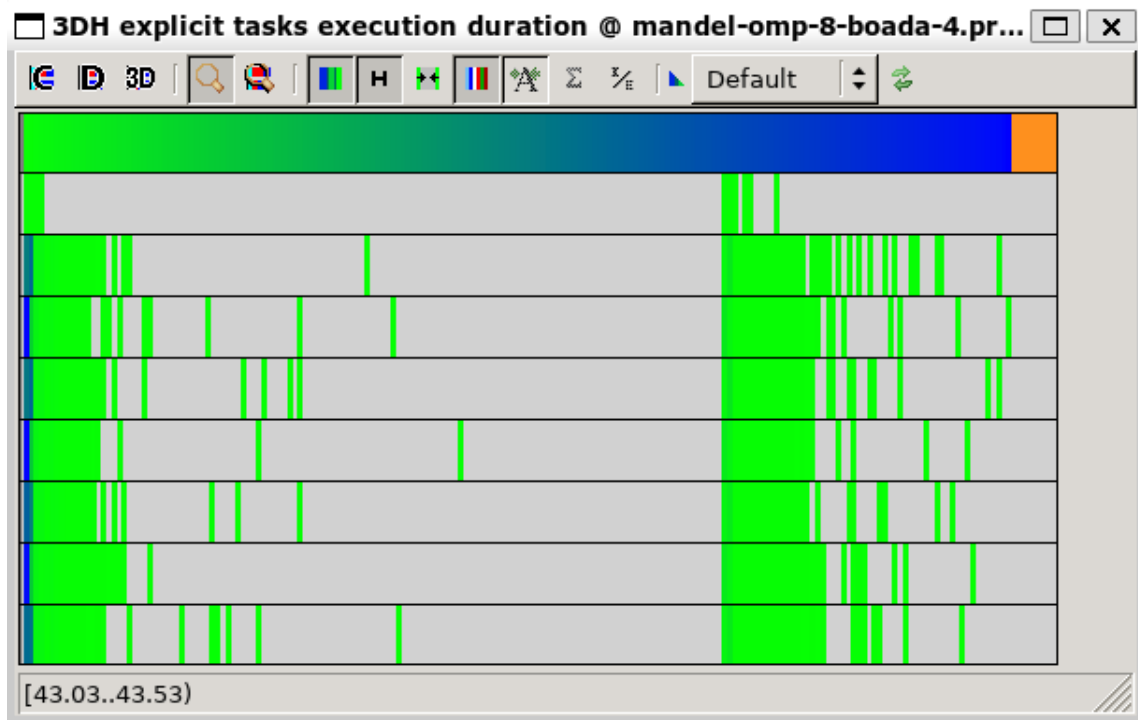


Figure 36: Explicit tasks execution duration histogram

To look at the overheads related with synchronization and tasks scheduling, we will use Hints → OpenMP → Thread state profile (efficiency).

2D thread state profile @ mandel-omp-8-boada-4.prv #1			
	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	47.35 %	0.00 %	52.65 %
THREAD 1.1.2	30.00 %	70.00 %	0.00 %
THREAD 1.1.3	29.84 %	70.16 %	0.00 %
THREAD 1.1.4	29.96 %	70.04 %	0.00 %
THREAD 1.1.5	30.08 %	69.92 %	0.00 %
THREAD 1.1.6	29.92 %	70.08 %	0.00 %
THREAD 1.1.7	30.12 %	69.88 %	0.00 %
THREAD 1.1.8	29.98 %	70.01 %	0.00 %
Total	257.25 %	490.09 %	52.66 %
Average	32.16 %	61.26 %	6.58 %
Maximum	47.35 %	70.16 %	52.65 %
Minimum	29.84 %	0.00 %	0.00 %
StDev	5.74 %	23.15 %	17.41 %
Avg/Max	0.68	0.87	0.13

Figure 37: Overheads related with synchronization and task scheduling

After analyzing all these pictures and diagrams, we can conclude that the load is well-balanced.

## 3.2. Point strategy with granularity control using taskloop

This time, we will be using taskloop instead of task, and the modified section of the code looks like this:

Note that we didn't specify the number of tasks (we would add `num_tasks(NUM_TASKS)` add to the *taskloop*), and therefore it will be OpenMP who will decide the granularity of the execution.

```
#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) {
      complex z, c;

      z.real = z.imag = 0;
```

**OMP\_NUM\_THREADS=1 ./mandel-omp -d -h -i 10000**

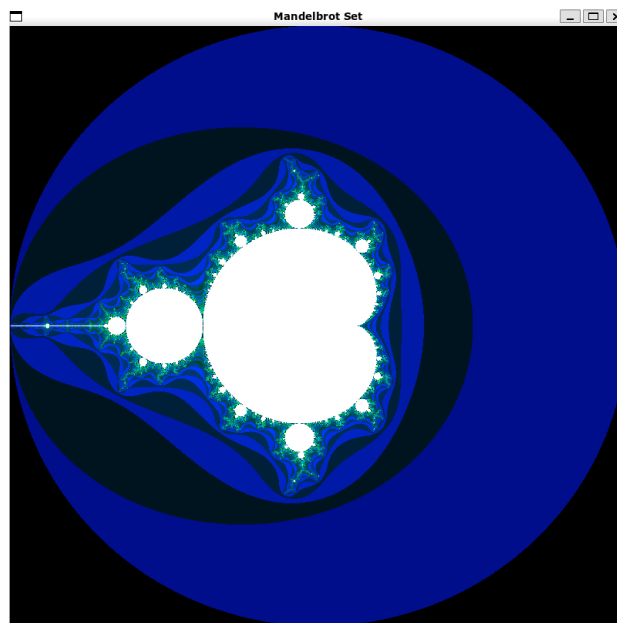
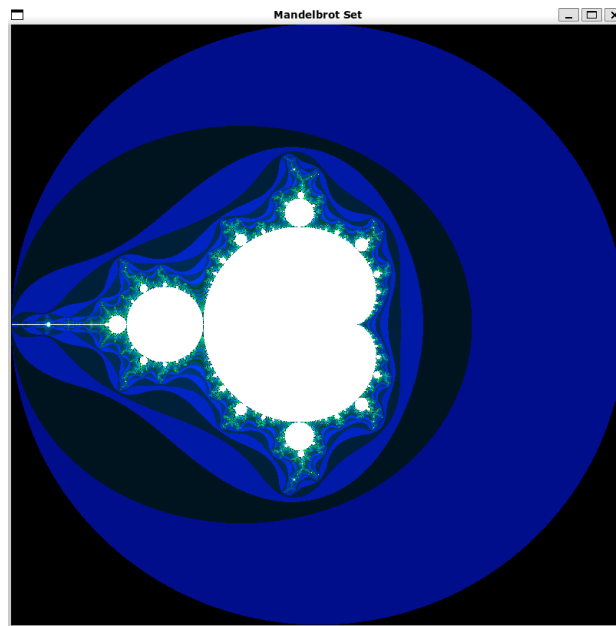


Figure 38: Execution of mandel-omp with 1 thread

**OMP\_NUM\_THREADS=2 ./mandel-omp -d -h -i 10000**



*Figure 39: Execution of mandel-omp with 2 threads*

Again, both pictures are identical, meaning that the Mandelbrot set is correct.

**sbatch ./submit-omp.sh mandel-omp 1**

```
par3108@boada-1:~/lab33$ cat time-mandel-omp-1-boada-4
3.92user 0.00system 0:04.04elapsed 97%CPU (0avgtext+0avgdata 7012maxresident)k
144inputs+5088outputs (1major+932minor)pagefaults 0swaps
```

And total execution time = 3.917473 s

**sbatch ./submit-omp.sh mandel-omp 8**

```
par3108@boada-1:~/lab33$ cat time-mandel-omp-8-boada-2
13.05user 0.10system 0:01.95elapsed 674%CPU (0avgtext+0avgdata 7832maxresident)k
144inputs+5088outputs (1major+2755minor)pagefaults 0swaps
```

And total execution time = 1.783076 s

Next, we check that the execution times and speed-ups are correct.

**sbatch submit-strong-omp.sh**

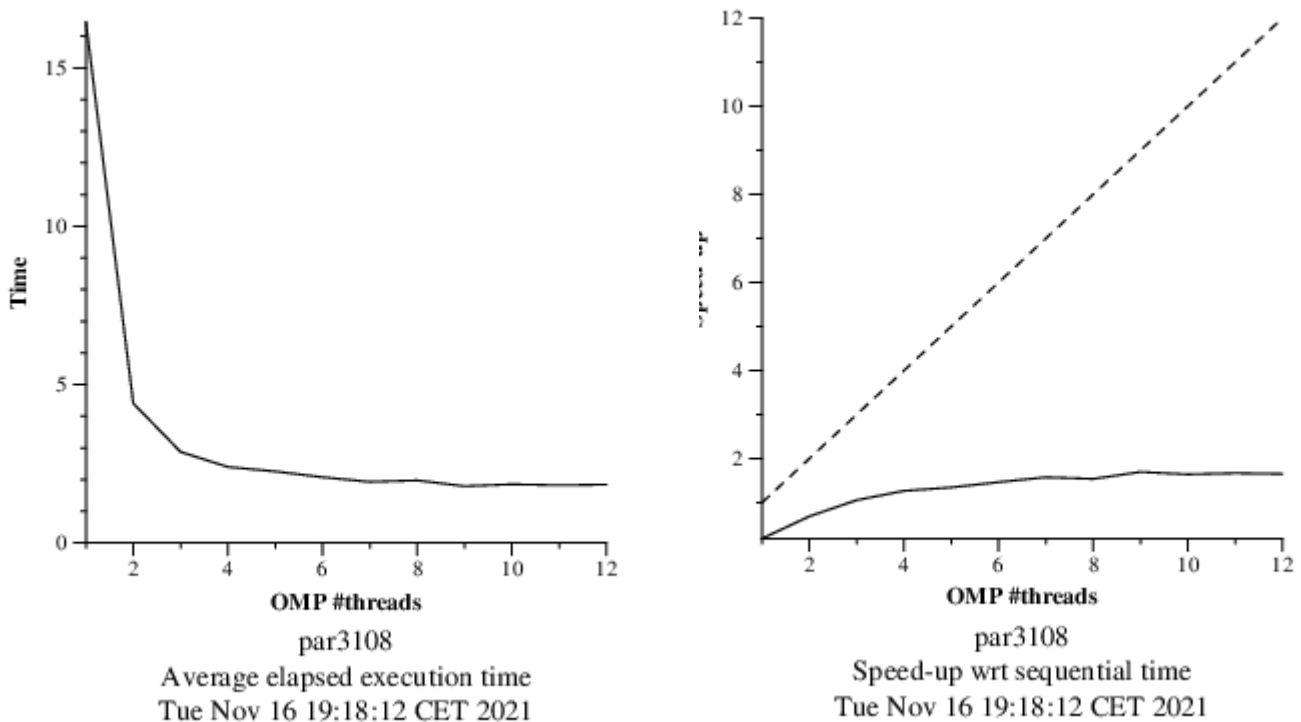


Figure 40: Scalability plot

From the scalability plot above, we can deduce that the execution times are slightly better than the ones obtained using *task* version. Yet again, both execution times and speed-ups are ~constant once we reach ~8 threads.

Next, we submit the binary to the *submit-extrae-sh* script in order to better understand how the execution actually works.

**sbatch submit-extrae.sh mandel-omp 8**

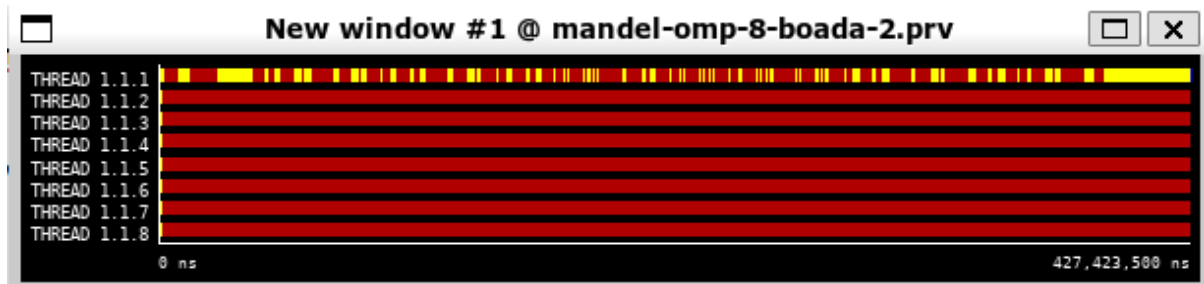


Figure 41: Execution of *mandel-omp.c* with Point Strategy using *taskloop* and 8 threads



Figure 42: Implicit tasks originated in the parallel construct

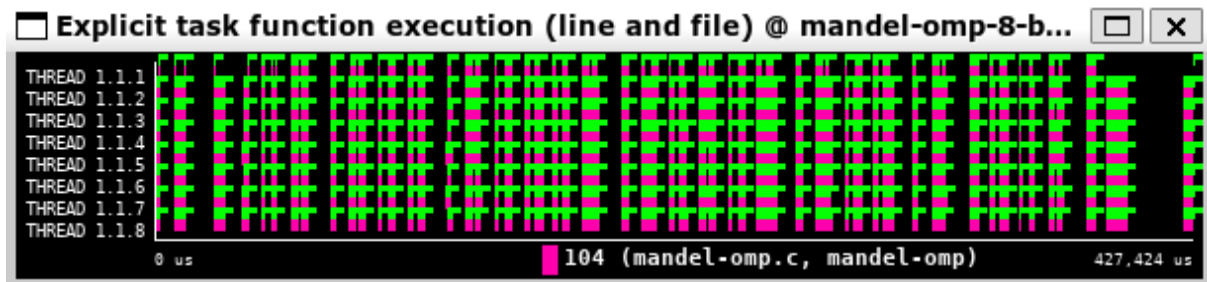


Figure 43: Explicit tasks execution

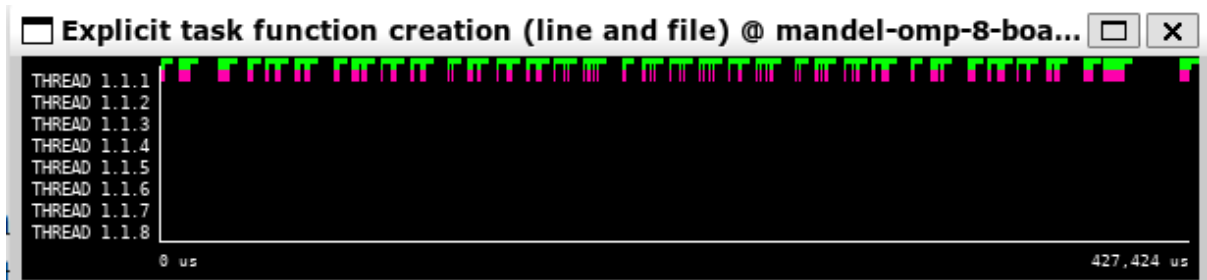


Figure 44: Explicit tasks creation

Again, we see that one thread creates all the tasks and the other ones just execute them.

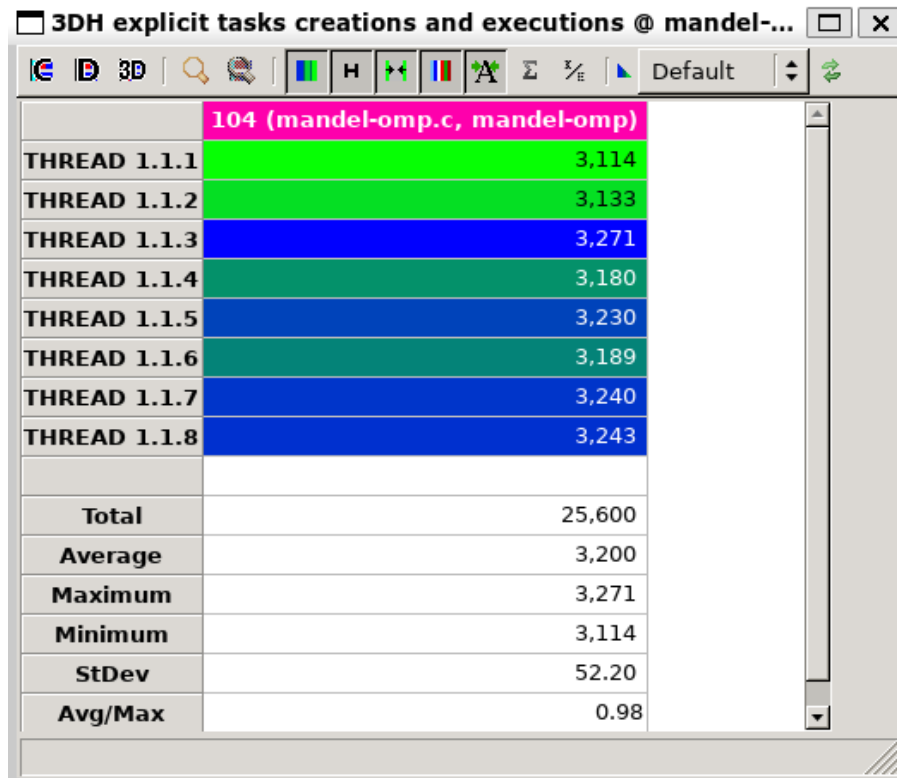


Figure 45: Explicit tasks creation and execution profile

From the previous figure (Figure 45), we see the statistics of the executed tasks, and we can obtain the number of tasks created/executed (25,600) and how are they distributed among all 8 threads.

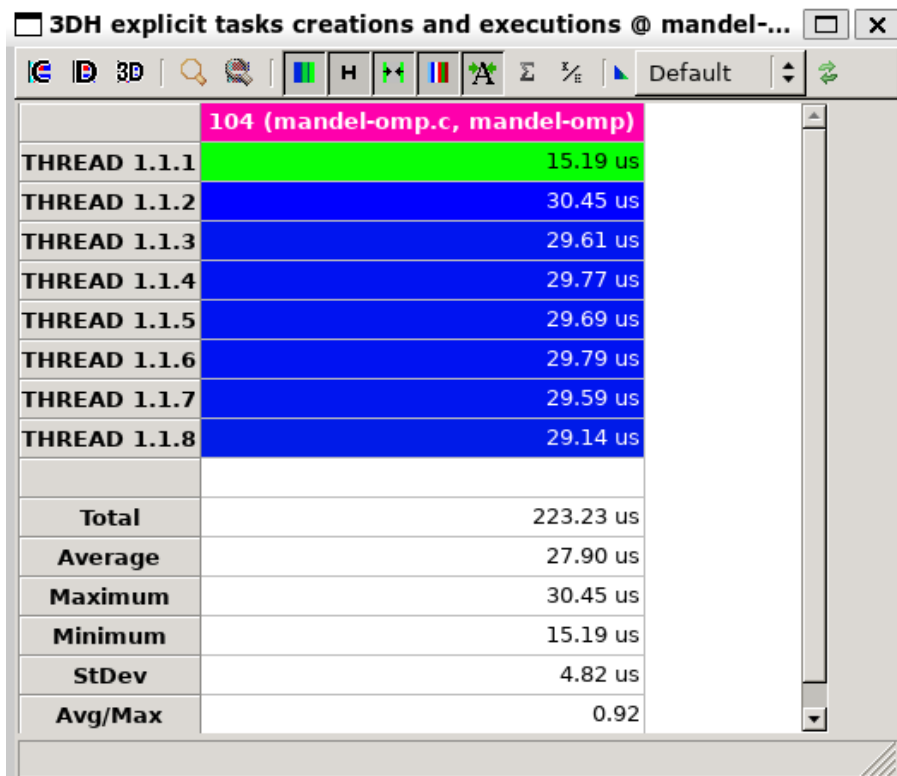


Figure 46: Explicit tasks creation and execution profile



In order to check if the load is well-balanced, we change the metric to show the average time spent executing a task by each thread. We see that only the thread 1.1.1 creates tasks, while the others execute them. The average time spent in each task is almost identical in all the threads (the thread 1.1.1 is creating the tasks, and has a higher average time).

In the next picture (Figure 47), we change again the metric to show this time the total time spent creating/executing the tasks, and we can conclude that the execution of tasks is well-balanced.

104 (mandel-omp.c, mandel-omp)	
THREAD 1.1.1	47,297.39 us
THREAD 1.1.2	95,394.77 us
THREAD 1.1.3	96,839.11 us
THREAD 1.1.4	94,678.03 us
THREAD 1.1.5	95,896.83 us
THREAD 1.1.6	95,000.49 us
THREAD 1.1.7	95,883.17 us
THREAD 1.1.8	94,494.85 us
Total	715,484.63 us
Average	89,435.58 us
Maximum	96,839.11 us
Minimum	47,297.39 us
StDev	15,942.55 us
Avg/Max	0.92

Figure 47: Explicit tasks creation and execution profile

The conclusions for both versions (*task* and *taskloop*) regarding load balance and granularity are the same.

In order to see which kind of task synchronization is being used by the program (*taskawait* or *taskgroup*) we generate the following diagrams:

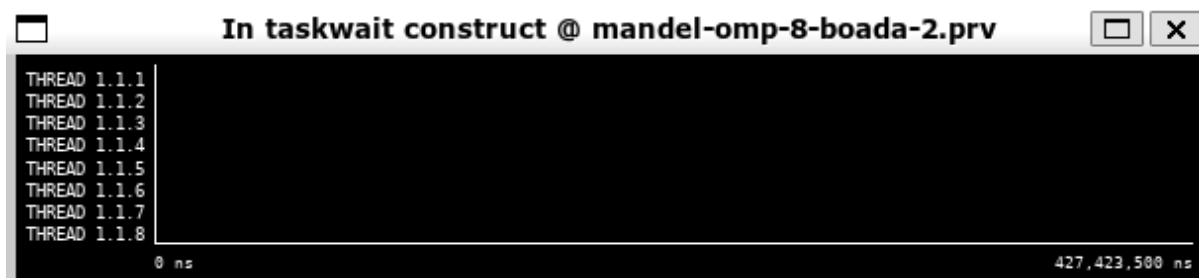


Figure 48: Taskwait constructs

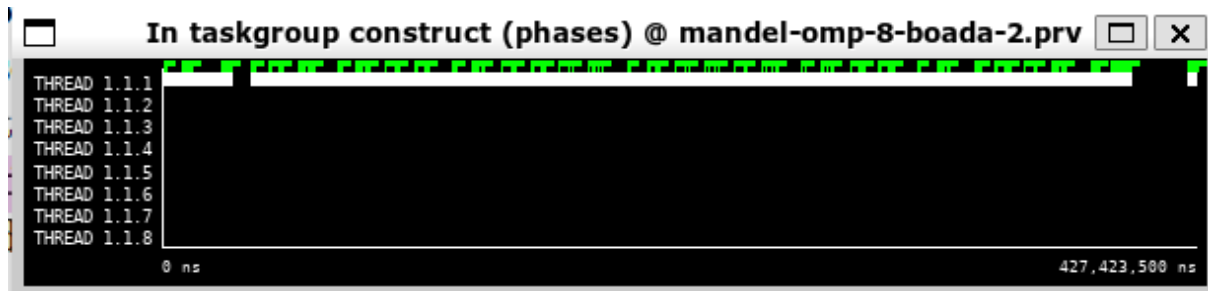


Figure 49: taskgroup constructs

As the pictures below show, taskgroup synchronization is being used in this program and is taking place in thread 1.1.1.

### 3.3. Row strategy implementation

In this final section, we are going to modify the file *mandel-omp.c* to really obey the Row strategy.

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp task firstprivate(row)
    for (int col = 0; col < width; ++col) {
        complex z, c;

        z.real = z.imag = 0;
```

OMP\_NUM\_THREADS=1 ./mandel-omp -d -h -i 10000

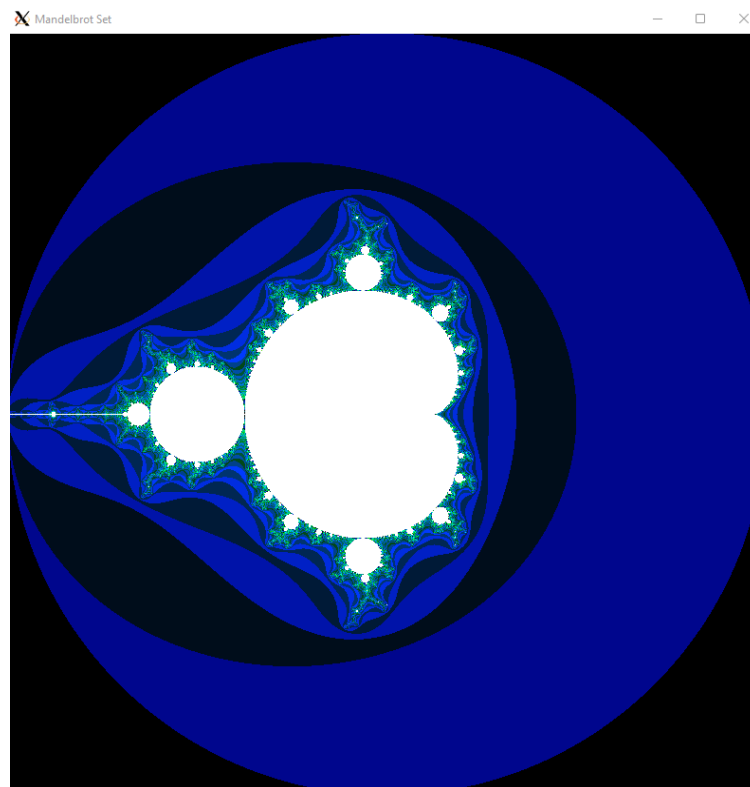


Figure 50: Execution of mandel-omp with 1 thread

The Mandelbrot set obtained is correct and after submitting the binary with the *submit-omp-sh* script, we get better results than the ones from the other versions (total execution time = 0.5798 s).

**sbatch submit-omp.sh mandel-omp 8**

```
par3108@boada-1:~/lab3$ cat submit-omp.sh.0137462
make: 'mandel-omp' is up to date.

Computation of the Mandelbrot set with:
  center = (0, 0)
  size = 2
  maximum iterations = 10000

Total execution time (in seconds): 0.579809

Mandelbrot set: Computed
Histogram for Mandelbrot set: Computed
Writing output file to disk: output_omp_8.out
```

And from the Scalability plot below we can see that the *Row* version has a really low execution time once we start using more threads. And the speedup obtained is not constant, like the ones from the other versions, this one is lineal.

**sbatch submit-strong-omp.sh**

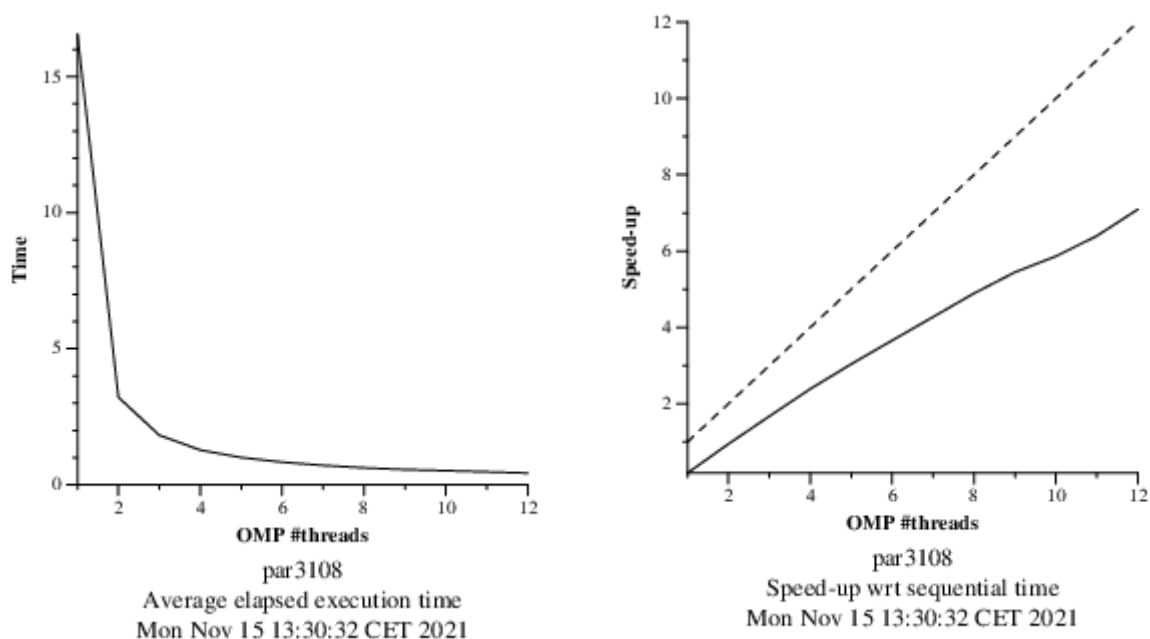


Figure 51: Scalability plot

Finally, we submit the binary to the `submit-extræ-sh` script in order to better understand how the execution actually works.

**sbatch submit-extræ.sh mandel-omp 8**



Figure 52: Implicit tasks originated in the parallel construct



Figure 53: Explicit tasks execution

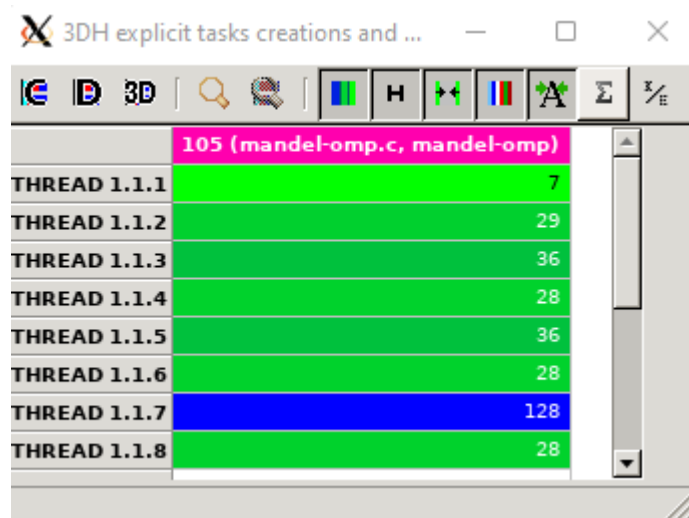


Figure 54: Explicit tasks creation and execution profile

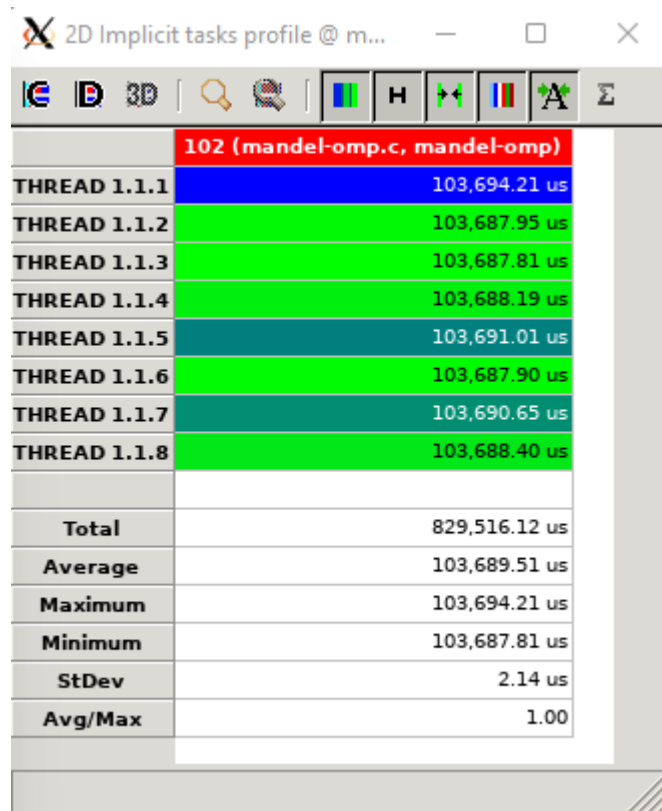


Figure 55: Implicit tasks creation and execution profile

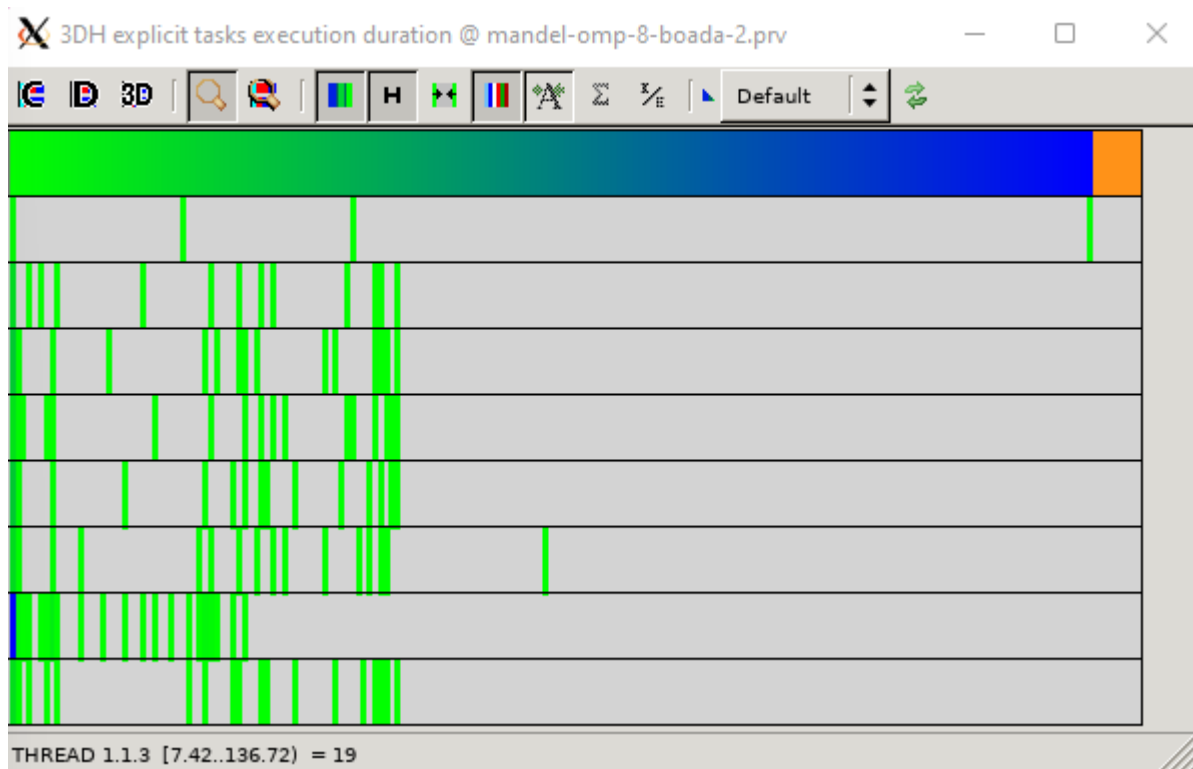


Figure 56: Explicit tasks execution duration diagram

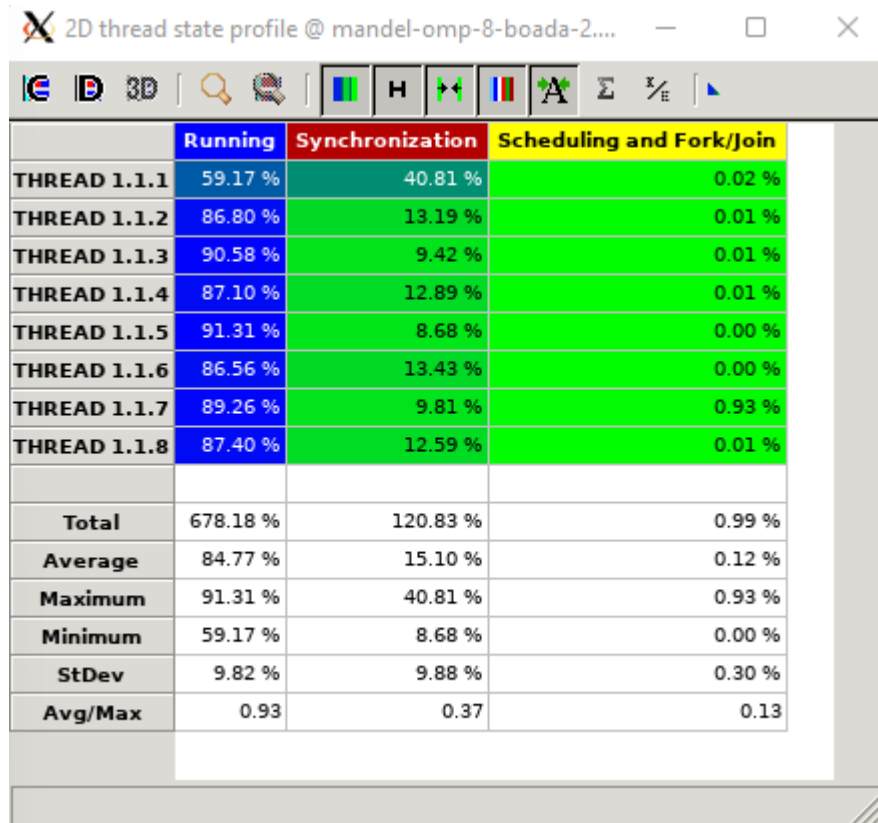


Figure 57: Overheads related with synchronization and task scheduling

## 5. Conclusions

In this laboratory, by using a program that computes the *Mandelbrot* set, we've studied the tasking model in *OpenMP* with the objective of understanding the iterative task decompositions.

We've seen two strategies for the task generation: *Point* and *Row*. We started by analyzing their behavior with the *Tareador* tool. There we saw the dependencies that need to be kept in mind, the load imbalances and simulations of executions with different number of threads. We also saw how the *Mandelbrot* set computation code can be embarrassingly parallel.

Then we continued by implementing the task decompositions in *OpenMP* and analyzing the overall performance of *Point*, developing three different versions: one where we defined tasks with the *task* pragma, one with *taskloop* and one with *task*, both with the *firstprivate* clause and the last one with *taskloop nogroup*. The *taskloop* version was the one with the best performance. Next we studied the *Row* strategy.

In conclusion, we can say that the *Row* strategy works better than the *Point* strategy when executing the computation of the *Mandelbrot* set due to the fact that it has quite a higher speed-up and fewer overheads during execution.