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

Group 2208

Course 2020/2021 Q1

Arnau Lamiel
Oscar Cañabate

arnau.lamiel@estudiantat.upc.edu
oscar.canabate@estudiantat.upc.edu

Index

-

1. Introduction

In this work we will explain the different parallelization strategies for the computation of Mandelbrot Set.

Mandelbrot Set is a particular set of points in the complex domain in which each point is defined by a certain function that will leave the points in a certain way, such that 2 dimensional fractal shapes are made.

So every 'c' complex point in the function: $fc(z) = z^2 + c$ does not diverge when iterated from z = 0. For instance, for which the sequence $fc(0), fc(fc(0)), etc$... remains bounded in absolute value.
The name and definition of the Set is given by Adrien Douady, to tribute the mathematician Benoit Mandelbrot.

There are different ways to compute the Mandelbrot Set, in which these fractal shapes are generated. In PAR, the algorithm used is the 'Exterior distance estimation', which computes the distance from point 'c' to the nearest point on the boundary of the set.

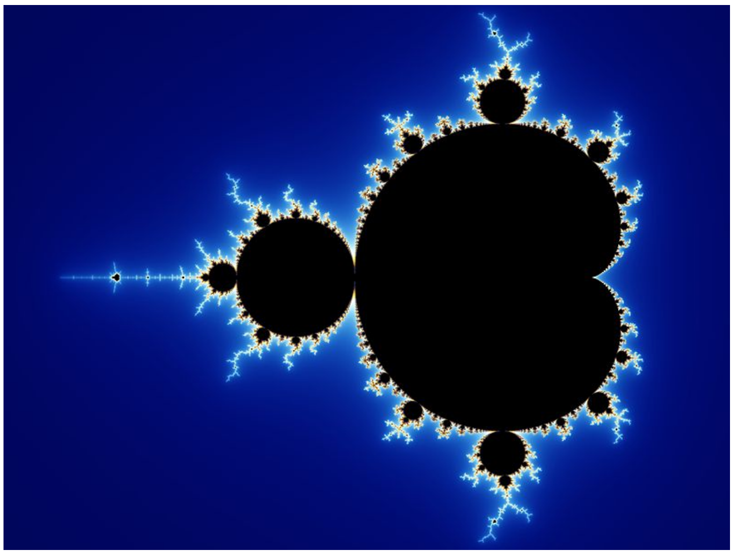


Figure 1. Mandelbrot Set representation

2. Parallelization Strategies

There are 2 main different parallelization strategies (Row and Point), and in this point we will explore it's differences.

We analyzed the 2 main strategies using Tareador. These two strategies, Row and Point decomposition.
With *Tareador*, we generated the dependency graphs with a certain number of iterations for each version.
In addition, there was not only the execution of the program, but also a display version that generated fractal shapes explained on the introduction, and generated the TDG's.
Now, we will see the TDG generated by mandel-tar displayed and no-displayed version for each strategy;

When we use row strategy, for each row of the matrix we are creating an individual task, and tasks are bigger, as seen in Figure 2.
But if we use point strategy, we are creating an individual task per every element of the matrix, as seen in Figure 3.

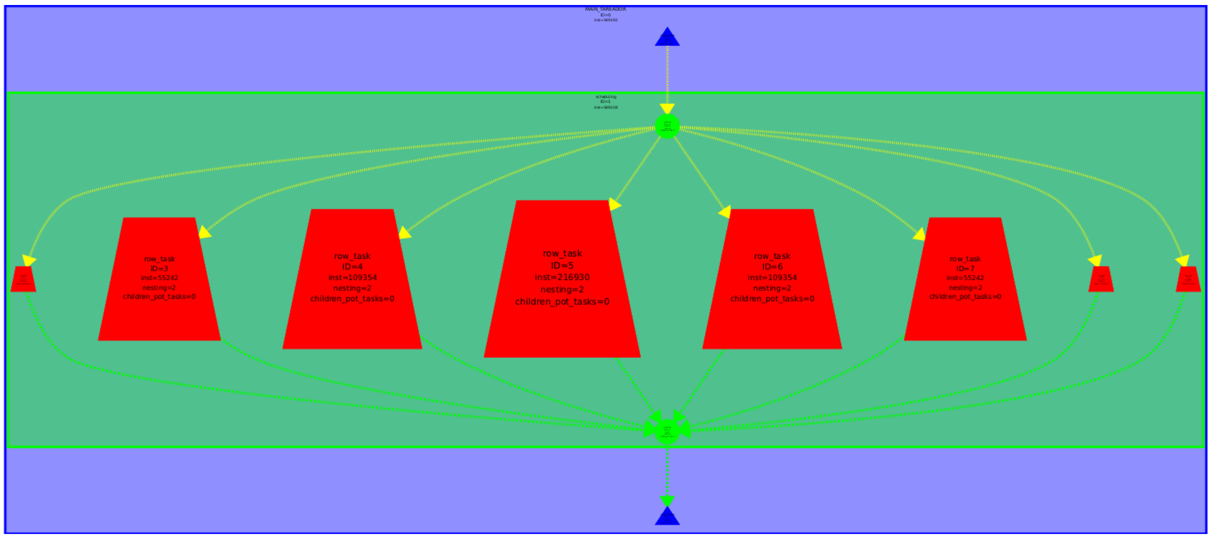


Figure 2: Row decomposition with Tareador (only name of binary)

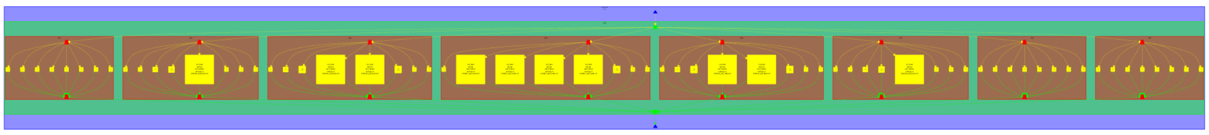


Figure 3: Point decomposition with Tareador (only name of binary)

Both strategies make "horizontal" graphs, meaning that both of them are very parallelizables. But there are tasks bigger than other ones, depending on the time it takes to determine whether a point belongs to the set.

When it comes to which strategy is better, the answer is that it depends. As we see in the graphs, point decomposition tasks cost much less time, and the big tasks are the critical path for the execution of the program, but the number of tasks generated is significantly big.
In the row decomposition strategy, tasks are much bigger, so the critical path will also cost much more, but the number of tasks is reduced in many steps.
We can conclude that, with an infinite number of processors, Point decomposition is much better. But if we don't have a big number of processors, it is better row decomposition because it will reduce the overhead of task creation.

As we can see in Figure 4, this looks like a sequential generation of tasks, which is obviously worst than the previous ones, so we do not consider it as a good option.

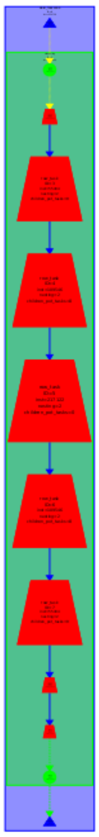


Figure 4: *Row decomposition with Tareador (name of binary & -d)*

In Figure 5 and 7, we can observe that the Point distribution is better than the Row distribution with the *-h* command, because it divides the task into more little tasks and can end the tasks quickly, and the workload is less than in the row decomposition. But this task decomposition is worse than the first one, because it is quite sequential, and the first ones (Figures 2,3) was much more parallelizable.

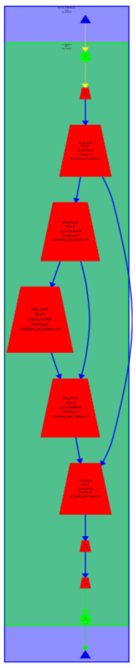


Figure 5: Row decomposition with Tareador (name of binary & -h)

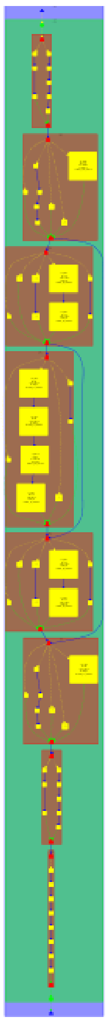
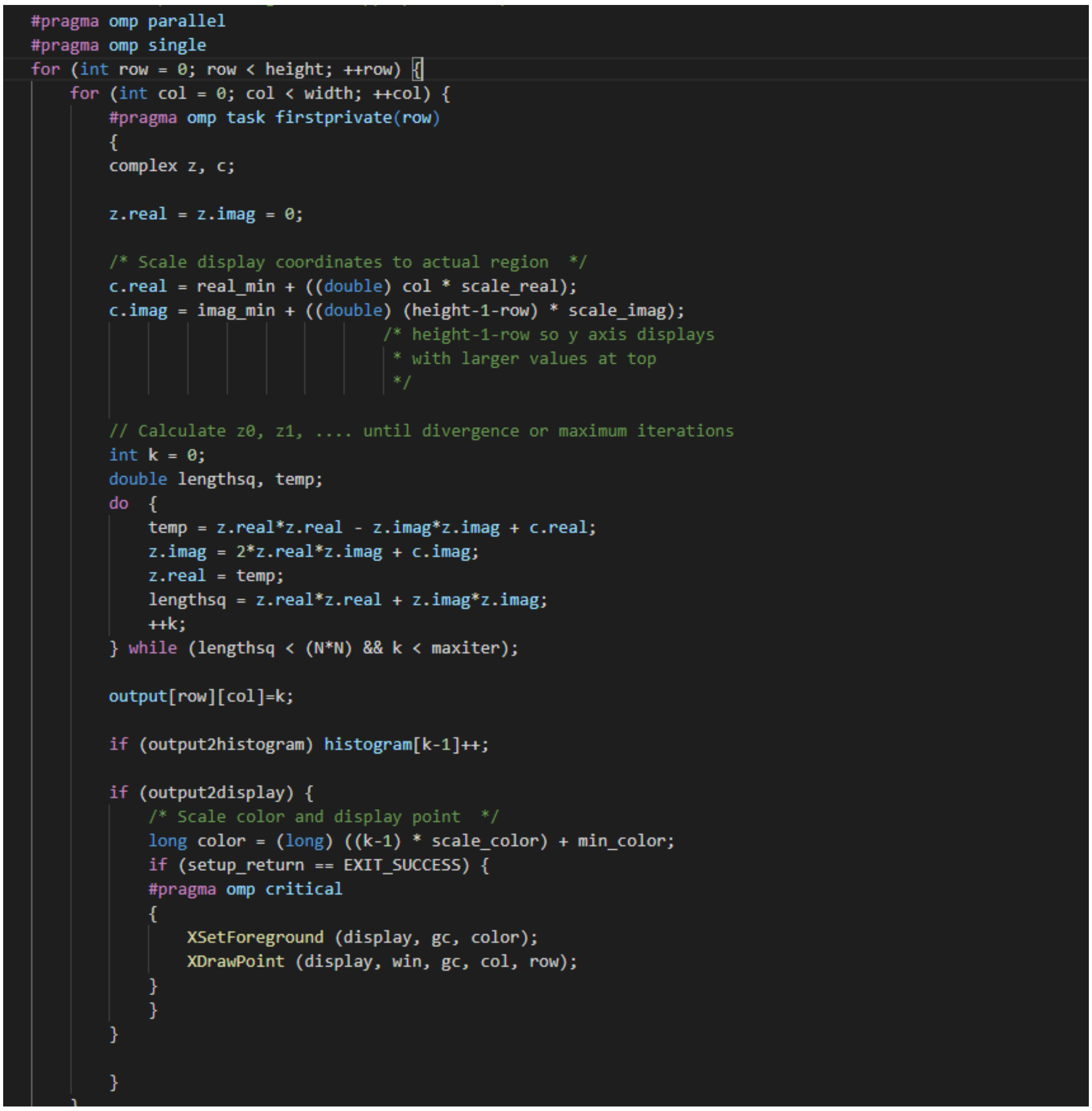


Figure 6: Point decomposition with Tareador (name of binary & -h)

In order to parallelize the program, we used *critical* clause from OpenMP, to have consistency when we update the variable, and improve parallelism.
We can see it at Figure 7.



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

        }
```

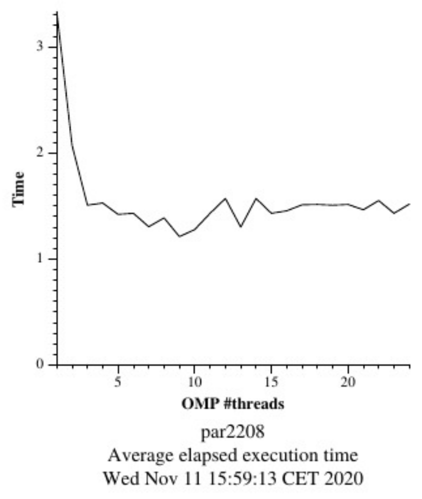
Figure 7: Code of the loop with critical

3. Performance analysis

Point decomposition in OpenMP

Task decomposition

In order to parallelize the code in an easy way, we could use a Point task-based strategy. The firstprivate clause avoids data races.
Code is the same as in Figure 7.
As we can see in Figure 8, in the scalability and speed-up plots for this version, we can see that there is no strong scalability, and the performance of this version is not good enough to consider it. We can observe that in the speed-up plot, this is so low and decreases for a bigger number of threads.Time is almost constant.



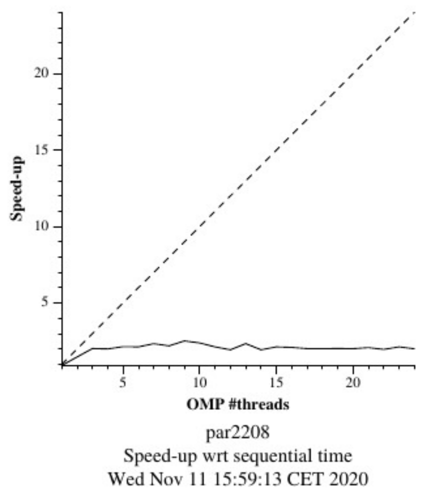
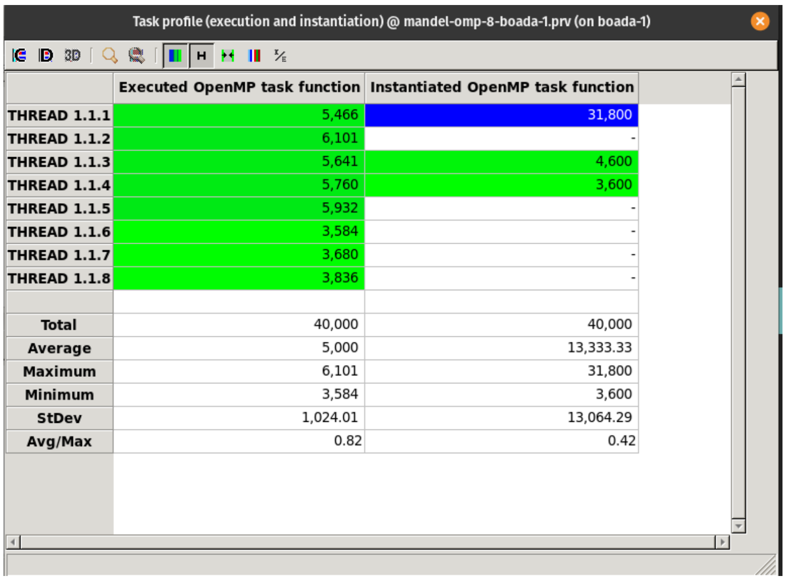


Figura 8: Strong Scalability from Point_v1.c (scalability and speed-up)

In the Figure above (Figure 9) we can see the statistics of how many tasks are being executed and created in each thread. We can observe that there have been created 40.000 tasks, resulting of the height*width task (200*200).



Task profile (execution and instantiation) @ mandel-omp-8-boada-1.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	5,466	31,800
THREAD 1.1.2	6,101	-
THREAD 1.1.3	5,641	4,600
THREAD 1.1.4	5,760	3,600
THREAD 1.1.5	5,932	-
THREAD 1.1.6	3,584	-
THREAD 1.1.7	3,680	-
THREAD 1.1.8	3,836	-
Total	40,000	40,000
Average	5,000	13,333.33
Maximum	6,101	31,800
Minimum	3,584	3,600
StDev	1,024.01	13,064.29
Avg/Max	0.82	0.42

Figure 9: Screenshot from paraver program for Point_v1

In the next figure (Figure 10) we can see that all threads are creating tasks, but 3 of them have task function instantiation. The execution time is 735.257 us.

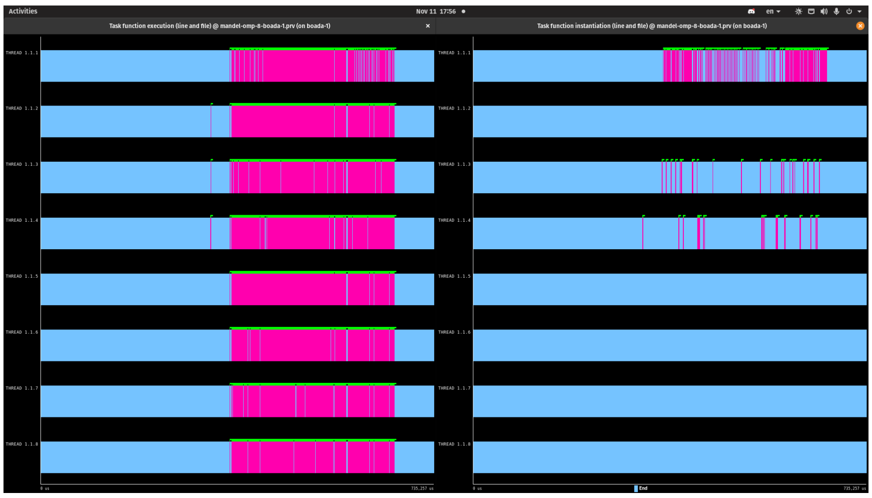
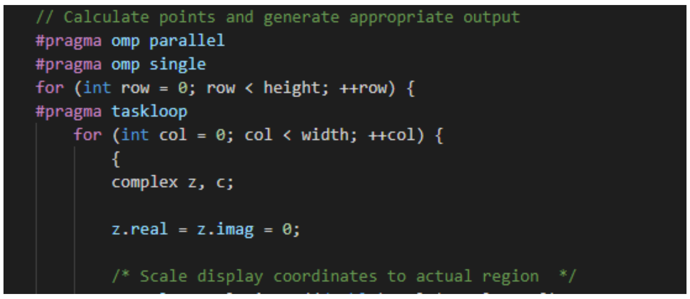

Task function execution (line and file) @ mandel-omp-8-boada-1.prv (on boada-1)
Task function instantiation (line and file) @ mandel-omp-8-boada-1.prv (on boada-1)
735.257 us

Figure 10: Chronological table from *paraver* program for the Point_v1

Granularity control using Taskloop decomposition

Now, we want to control the granularity with *taskloop* clause of OpenMP. For the taskloop, we used the code in Figure 11.



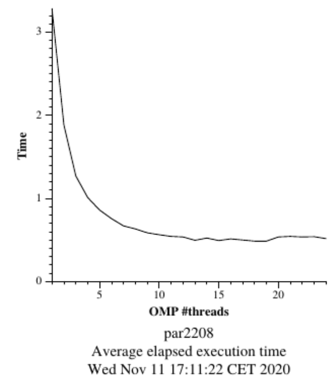
```
// Calculate points and generate appropriate output
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
#pragma taskloop
    for (int col = 0; col < width; ++col) {
        {
        complex z, c;

        z.real = z.imag = 0;

        /* Scale display coordinates to actual region  */
```

Figure 11: Code for the taskloop decomposition

In Figure 12 we can observe the plots for scalability and speed-up. We can see that the speedup is quite better than the previous one, because the taskloop is executed in the internal loop. And we can see that the time is lesser with a bigger number of threads, which is good and necessary.
We can conclude that is more worth it than the task but is not good enough.



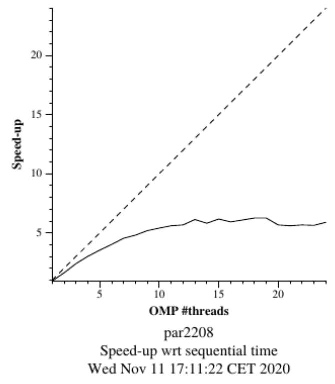
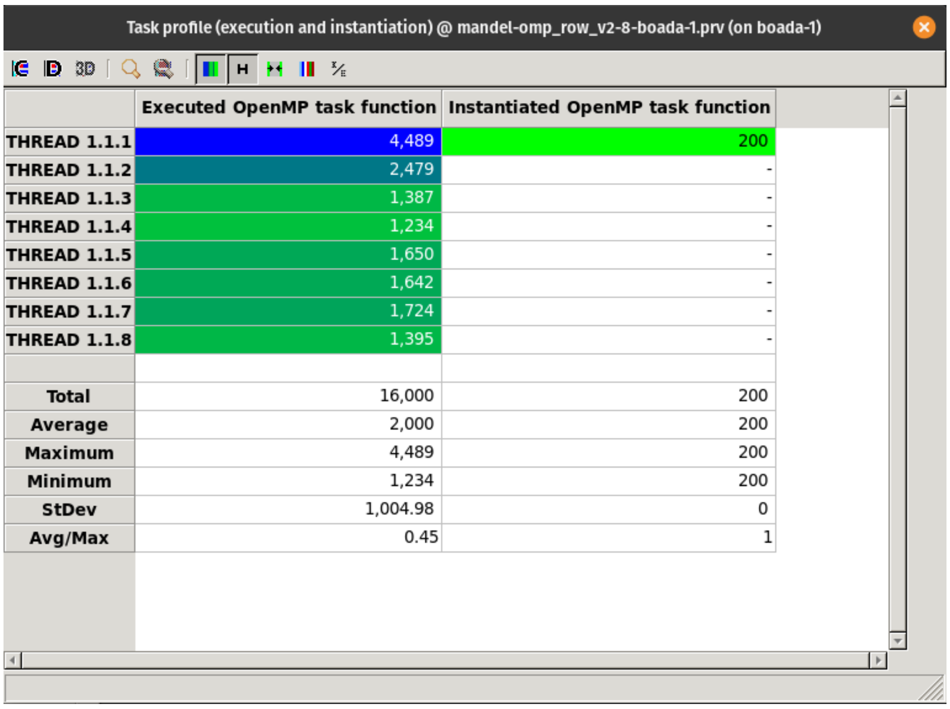


Figure 12: *Strong Scalability from Point_v2.c (* scalability and speed-up)

At Figure 13 and Figure 14, we observe that only one thread is doing a task instantiation, and there are 16.000 tasks created for all threads. The number of tasks is reduced, getting a better performance. The execution time is reduced (660.759 us)


Task profile (execution and instantiation) @ mandel-omp_row_v2-8-boada-1.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	4,489	200
THREAD 1.1.2	2,479	-
THREAD 1.1.3	1,387	-
THREAD 1.1.4	1,234	-
THREAD 1.1.5	1,650	-
THREAD 1.1.6	1,642	-
THREAD 1.1.7	1,724	-
THREAD 1.1.8	1,395	-
Total	16,000	200
Average	2,000	200
Maximum	4,489	200
Minimum	1,234	200
StDev	1,004.98	0
Avg/Max	0.45	1

Figure 13: *Screenshot from paraver program for Point_v2*

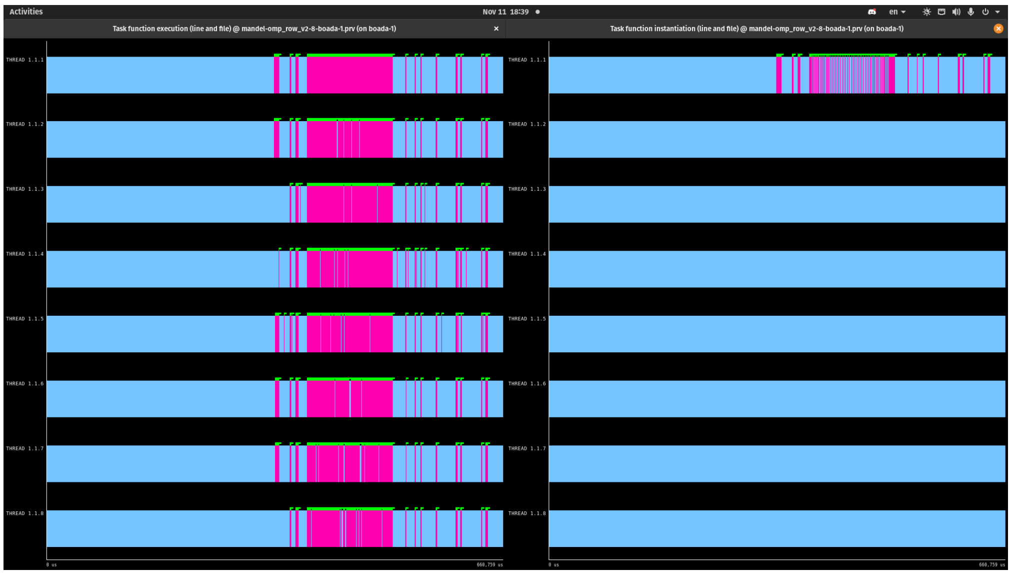
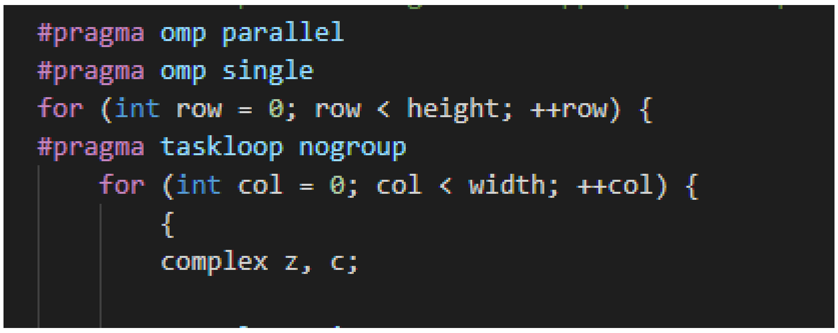

Activities
Nov 11 18:39
Task function execution (line and file) @ mandel-omp_row_v2-8-boada-1.prv (on boada-1)
Task function instantiation (line and file) @ mandel-omp_row_v2-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 us
660,759 us

Figure 14: Chronological table from *paraver* program for the Point_v2

Now, we will analyze the *taskloop no group* clause. The code is shown in Figure 15.



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

Figure 15: Code for taskloop no group

In Figure 16, we can see the plots of scalability and speed-up. And we can observe that the speed up is better now than the previous two versions, and the scalability too (with a little difference with the common taskloop). With 13-14 threads, it reaches its maximum speed-up.

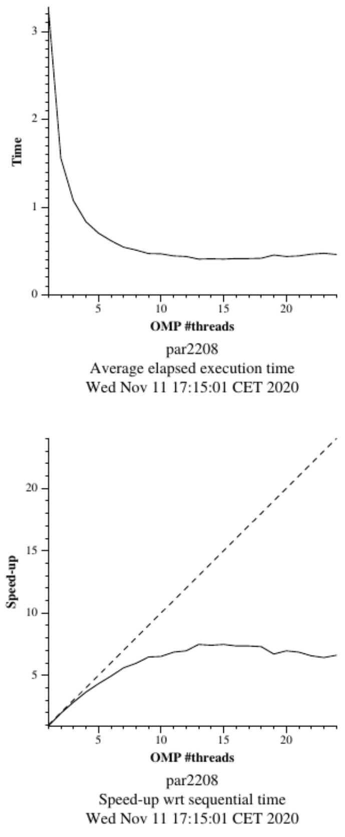
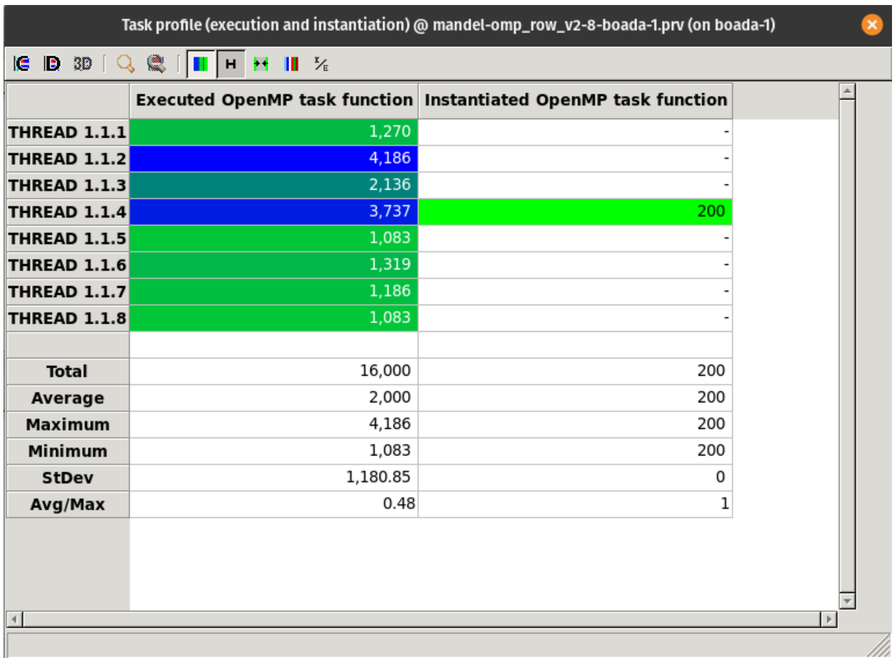


Figure 16: Strong Scalability for Point_v3.c (scalability and speed-up)

For the Figures 17 and 18, we can see that is similar to common taskloop. We created 16.000 tasks and just one thread is doing the task function instantiation, but it is another thread than the previous version. The main difference is the size of the tasks. But we have to observe that the execution time is bigger than the lasts options (813,261 us)



Task profile (execution and instantiation) @ mandel-omp_row_v2-8-boada-1.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	1,270	-
THREAD 1.1.2	4,186	-
THREAD 1.1.3	2,136	-
THREAD 1.1.4	3,737	200
THREAD 1.1.5	1,083	-
THREAD 1.1.6	1,319	-
THREAD 1.1.7	1,186	-
THREAD 1.1.8	1,083	-
Total	16,000	200
Average	2,000	200
Maximum	4,186	200
Minimum	1,083	200
StDev	1,180.85	0
Avg/Max	0.48	1

Figura 17: *Screenshot from paraver program for Point_v3*

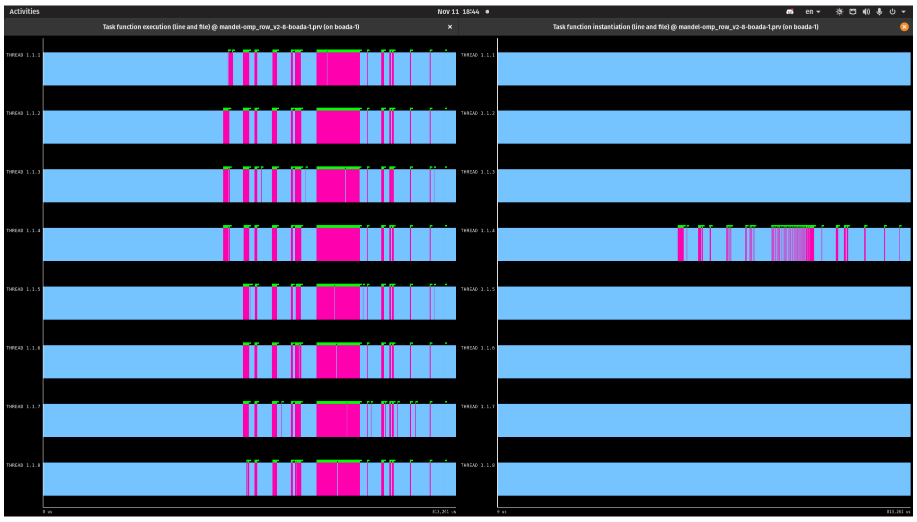
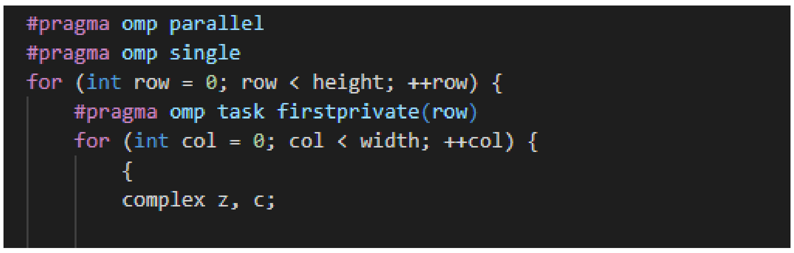

Task function execution (line and file) @ mandel-omp_row_v2-8-boada-1.prv (on boada-1)
Task function instantiation (line and file) @ mandel-omp_row_v2-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 us
813,261 us

Figure 18: Chronological table from *paraver* program for the Point_v3

Row decomposition in OpenMP

Task decomposition

We have analyzed the Point decomposition, and now we are going to analyze the Row decomposition with the same task decomposition strategies.
In this case we use only the creation of tasks, as we did in the Point task decomposition. We can see it at Figure 19.



```
#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;
```

Figure 19: Code for task decomposition

When we observe the plots (Figure 20), we have our definite confirmation that this type of parallelization is much better than the Point ones. The Speed-Up is much better, is almost perfect, and the scalability too, because the time reduces for a bigger number of processors.

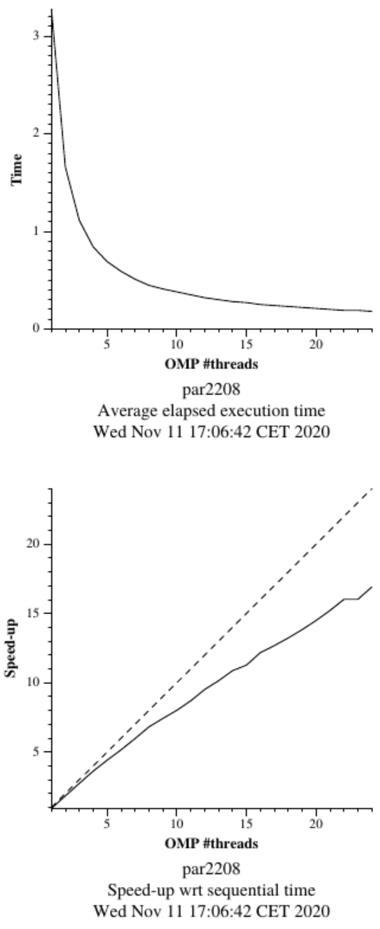
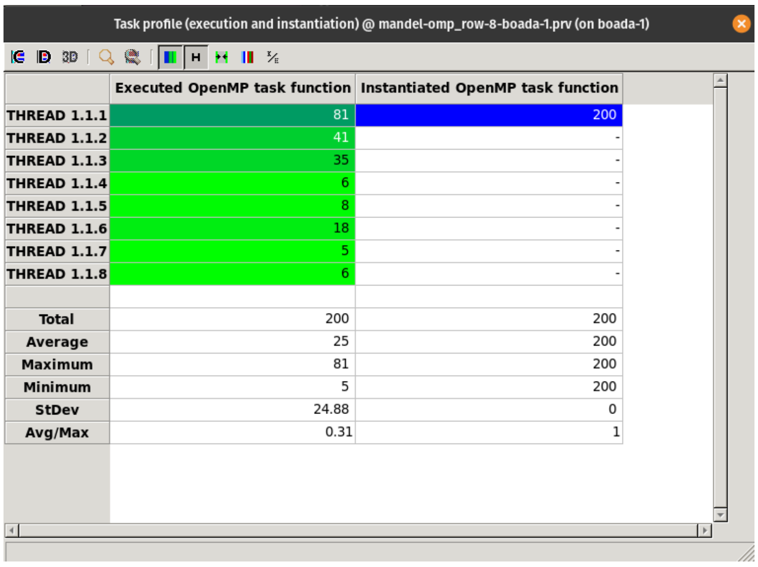


Figure 20: Strong Scalability for Point_v3.c (scalability and speed-up)

As we can see in the next Figures 21 and 22, it is similar to the Point_v2 version because thread 1 is instantaining task function, but is different because it just did it one time, and it creates 200 tasks. These tasks are so little, compared to Point strategy. The important thing here is that the execution time is 432.333 us, lesser than all the previouses.



Task profile (execution and instantiation) @ mandel-omp_row-8-boada-1.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	81	200
THREAD 1.1.2	41	-
THREAD 1.1.3	35	-
THREAD 1.1.4	6	-
THREAD 1.1.5	8	-
THREAD 1.1.6	18	-
THREAD 1.1.7	5	-
THREAD 1.1.8	6	-
Total	200	200
Average	25	200
Maximum	81	200
Minimum	5	200
StDev	24.88	0
Avg/Max	0.31	1

Figure 21: Screenshot from paraver program for Row_v1

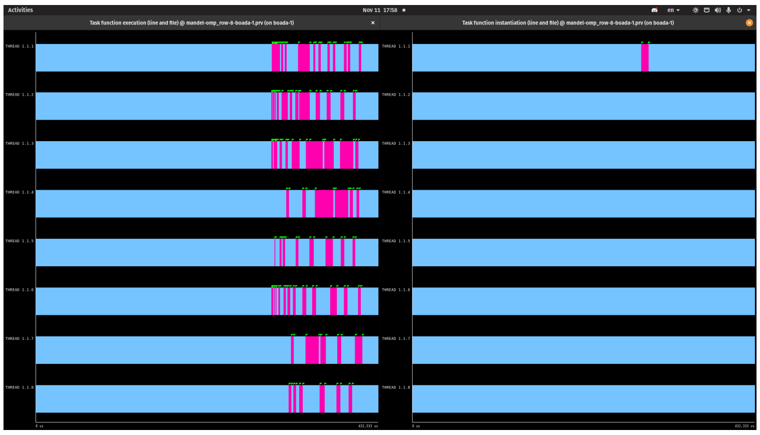
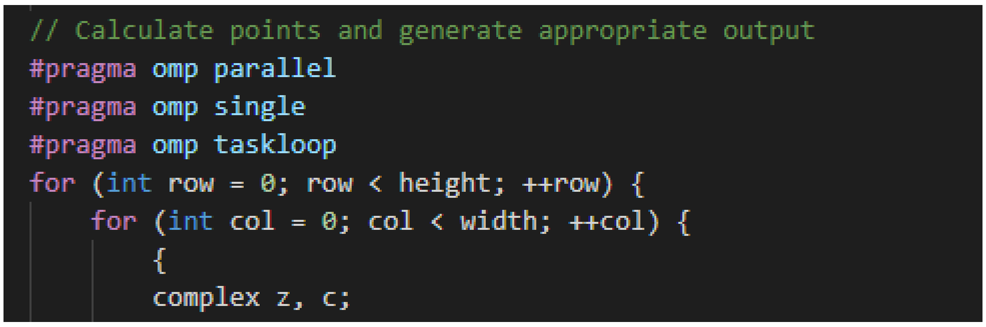


Figure 22: Chronological table from *paraver* program for the Row_v1

Granularity control using Taskloop decomposition

Now we are going to analyze the taskloop, code in Figure 23.



```
// Calculate points and generate appropriate output
#pragma omp parallel
#pragma omp single
#pragma omp taskloop
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        {
        complex z, c;
```

Figure 23: Code for taskloop

In the next Figure (24), we can see the plots of speedup and scalability. We can see that it is almost perfect too, but is worse than the task one, because there are a number of threads that the speed-up decreases, and the scalability is not so perfect.

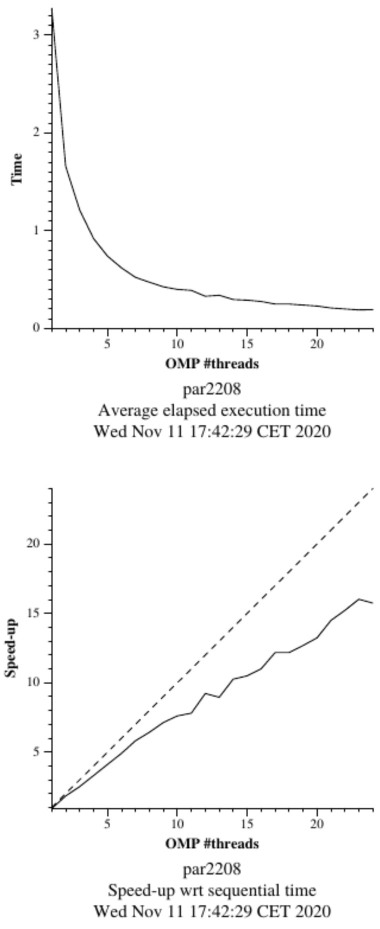
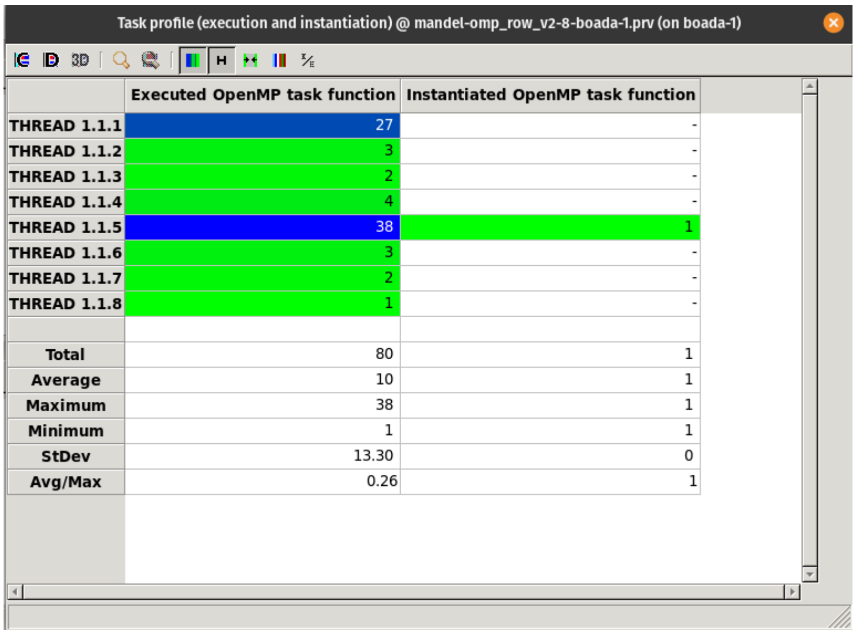


Figure 24: Strong Scalability for Row_v2.c (scalability and speed-up)

In the figures above (25 and 26), we can see the task profile. We observed that all threads are executing tasks, but not all the threads have the same workload. Just one thread (in this case is the 5th) is initializing a task function, and just one time like in the Figure 22, but for a smaller period (it just makes 1 task). The execution time is 437.184 us, practically the same as Figure 22 (a little worse), and we created just 80 tasks.



Task profile (execution and instantiation) @ mandel-omp_row_v2-8-boada-1.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	27	-
THREAD 1.1.2	3	-
THREAD 1.1.3	2	-
THREAD 1.1.4	4	-
THREAD 1.1.5	38	1
THREAD 1.1.6	3	-
THREAD 1.1.7	2	-
THREAD 1.1.8	1	-
Total	80	1
Average	10	1
Maximum	38	1
Minimum	1	1
StDev	13.30	0
Avg/Max	0.26	1

Figure 25: Screenshot from paraver program for Row_v2

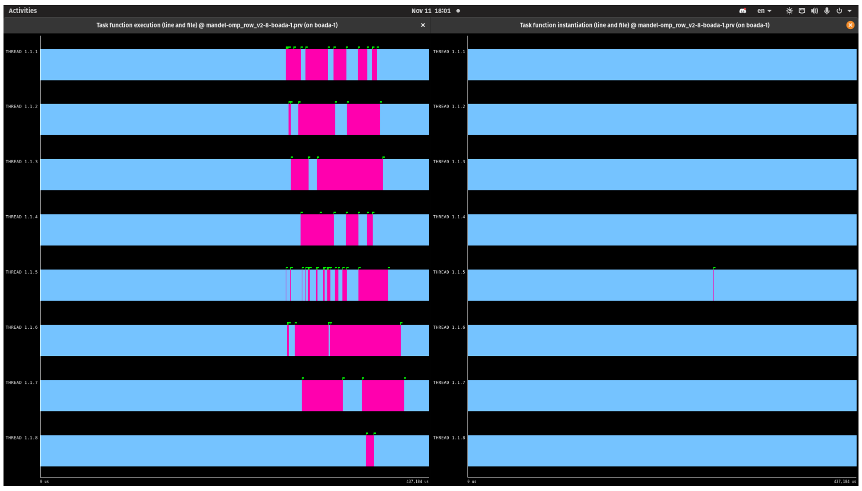
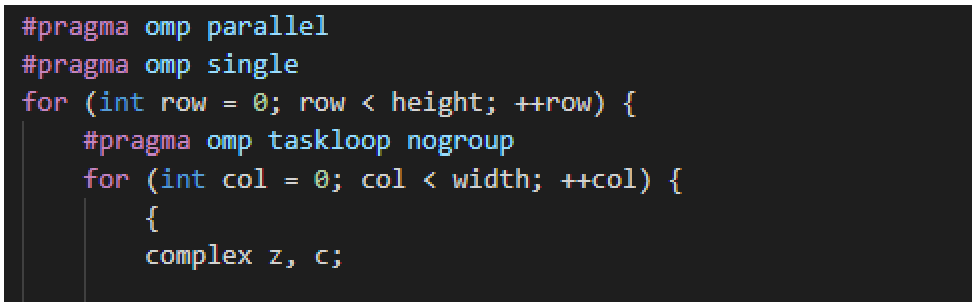


Figure 25: Chronological table from *paraver* program for the Row_v2

For the last part of the analysis of the performance, we make the *taskloop nogrup* clause for the Row decomposition, code above, in Figure 26.



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

Figure 26: Code for taskloop nogroup

We will study now the plots that are generated by this code (Figure 27).
Like in the *taskloop* clause, we can observe that these plots are almost perfect, but worse than the *task.* Maybe it is a little better, but is practically identical. We can see that for a certain number of threads it becomes constant or decreases. The scalability is good too, like all the Row decomposition strategies.

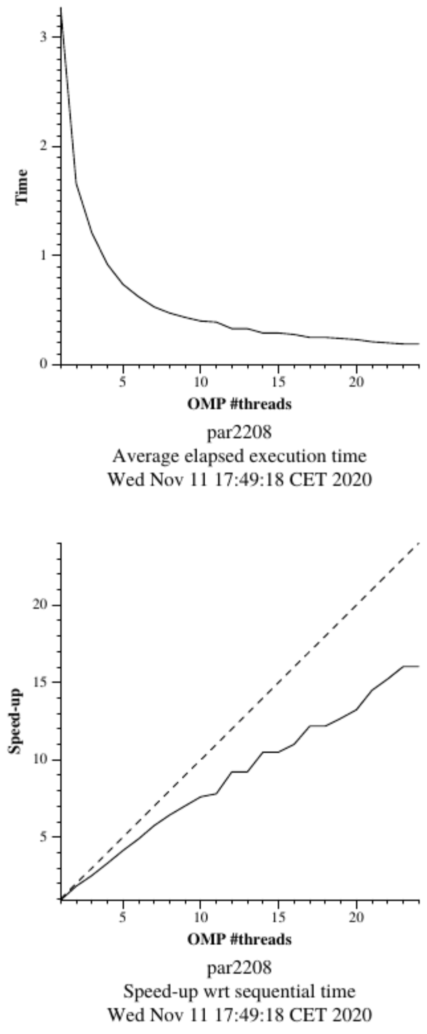
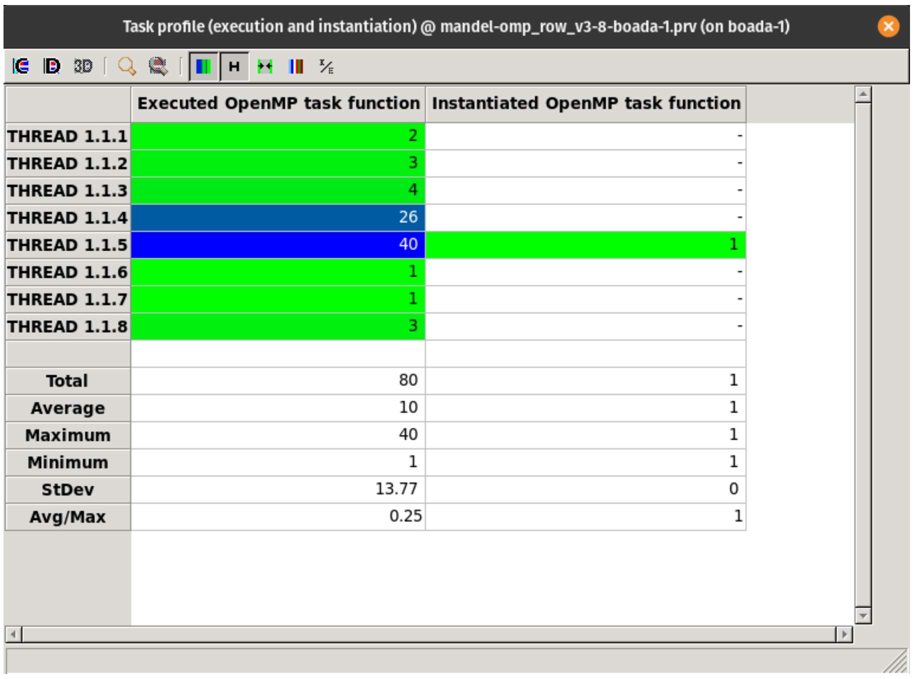


Figure 27: Strong Scalability for Row_v2.c (scalability and speed-up)

In the figures 28 and 29, we can observe that, like in the Figure 25 and 26, we created 1 instantiated task function in a thread, and we created in total 80 tasks. We can affirm that in that case, it is practically the same. The execution time is 513.114 us, so is notably worse than the previouses version.


Task profile (execution and instantiation) @ mandel-omp_row_v3-8-boada-1.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	2	-
THREAD 1.1.2	3	-
THREAD 1.1.3	4	-
THREAD 1.1.4	26	-
THREAD 1.1.5	40	1
THREAD 1.1.6	1	-
THREAD 1.1.7	1	-
THREAD 1.1.8	3	-
Total	80	1
Average	10	1
Maximum	40	1
Minimum	1	1
StDev	13.77	0
Avg/Max	0.25	1

Figure 28: Screenshot from paraver program for Row_v3

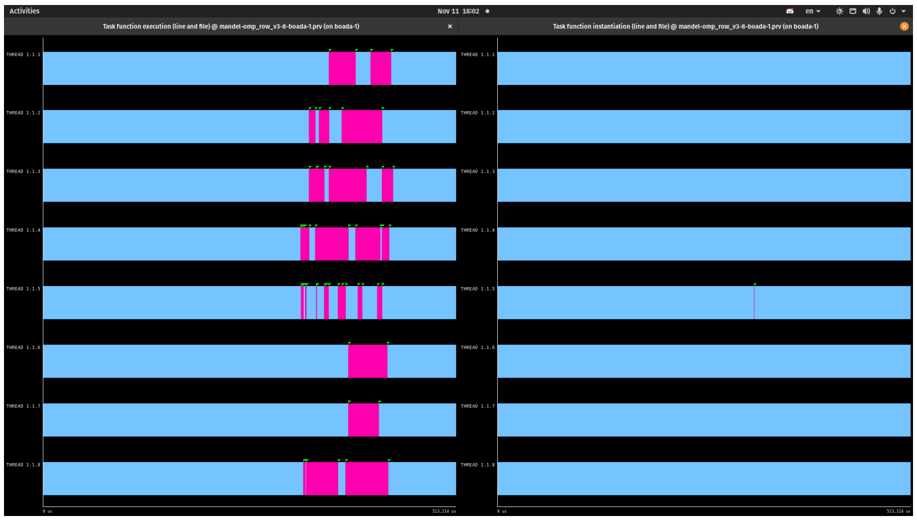

Activities
Nov 11 18:02
Task function execution (line and file) @ mandel-omp_row_v3-8-boada-1.prv (on boada-1)
Task function instantiation (line and file) @ mandel-omp_row_v3-8-boada-1.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 us
513,114 us

Figure 29: Chronological table from *paraver* program for the Row_v3

4. Conclusions

In this work, we analyzed the different strategies to parallelize a code which obtains Mandelbrot Set.
On the one hand, if we keep in mind the first part of this work, ‘Parallelization Strategies’, done with *Tareador*, we can see that if we use a huge number of threads/processors, it seems that Point decomposition is better than Row.
On the other hand, if we observe the second part ‘Performance analysis’, we can think that Row decomposition is much better, because of the Speed-up and the Scalability, which are best in this case than the Point one.
So, we can conclude that, depending on the grainsize and the strategy, we will decide if one is better or worse. We have to determine which are our expectations in order to decide which implementation is better for our project, but in this work we will have to go deeper if we want to analyse that.