Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

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

Izan Cordobilla Blanco (par1304)

Alejandro Salvat Navarro (par1120)

Grau en enginyeria informàtica

Quadrimestre de primavera 2021-2022

INDEX

1. INTRODUCTION

In this laboratory assignment we are going to explore the tasking model in OpenMP to express iterative task decompositions. Before that, we will start exploring the most appropriate ones by using Tareador. The program that we will use consists in the computation of the Mandelbrot set, a particular set of points, in the complex domain, whose boundary generates a distinctive and easily recognisable two dimensional fractal shape (it is shown in Figure 1). For each point c in a delimited two-dimensional space, the complex quadratic polynomial recurrence zn+1 = z 2 n + c is iteratively applied n to determine if it belongs or not to the Mandelbrot set.

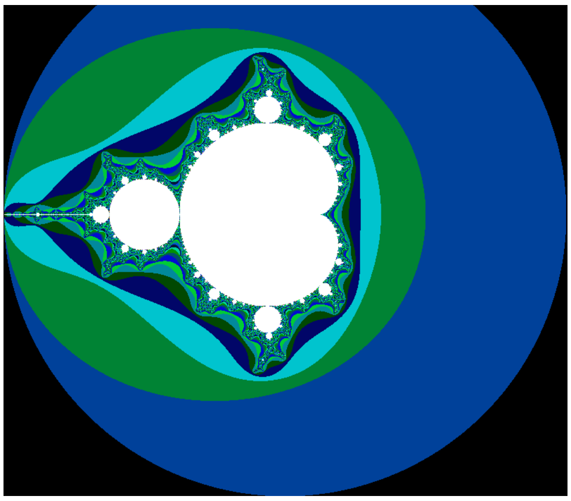


Figure 1: Mandelbrot Set Representation

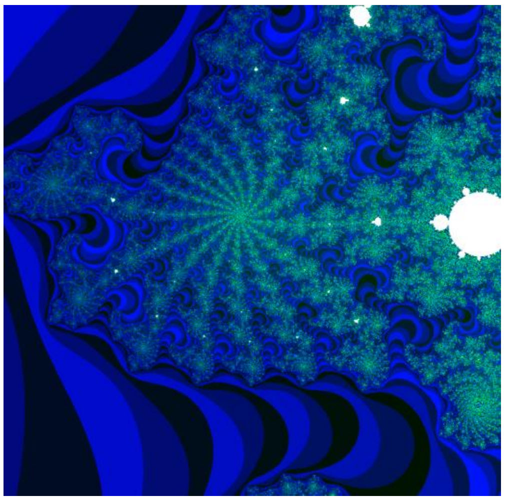


Figure 2: " Execution with ./mandel-seq -d -c -0.737 0.207 -s 0.01 -i 100000"

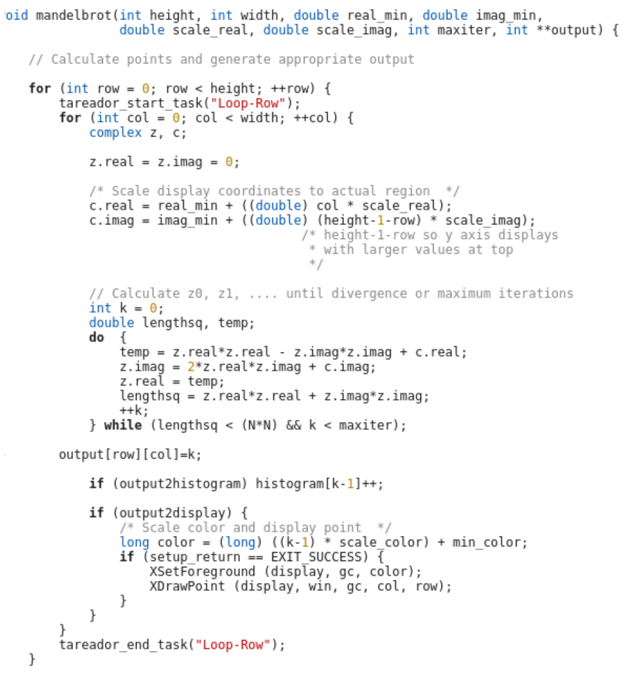
2. TASK DECOMPOSITION ANALYSIS FOR THE MANDELBROT SET COMPUTATION

Firstly, in this section we will study the main characteristics of each proposed task decomposition strategy (Row and Point) using Tareador, in both cases with a granularity of one iteration per task.

To understand which are the changes that we will have to apply to the code in order to achieve this first part, we have to understand in which consists the two strategies:

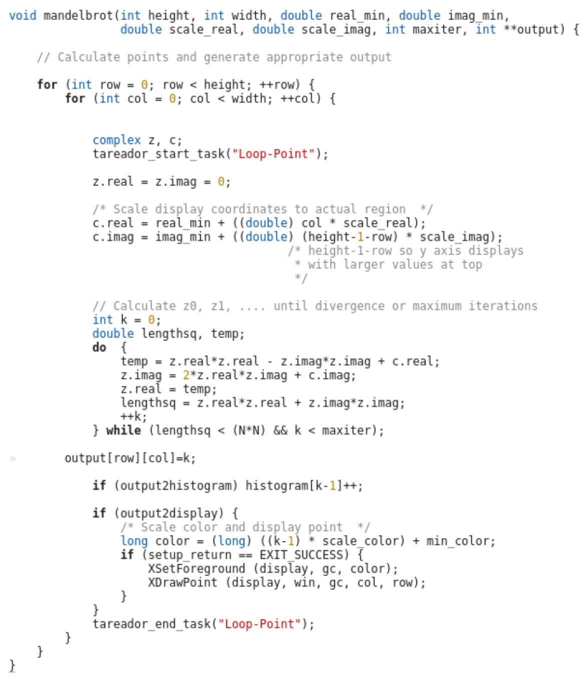
- Row strategy: a task will correspond with the computation of one or more (consecutive) rows of the Mandelbrot set.
- Point strategy: a task will correspond with the computation of one or more (consecutive) points in the same row of the Mandelbrot set.

Now, we have to start looking, understanding and changing the *mandel-tar.c* code in order to analyze the potential parallelism for the Row and Point strategies with Tareador. In the following figures we can see the changes in *mandel-tar.c*.



```
oid 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("Loop-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("Loop-Row");
    }
```

Figure 3: mandel-tar.c with Row Strategy



```
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) {


            complex z, c;
            tareador_start_task("Loop-Point");

            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("Loop-Point");
        }
    }
}
```

Figure 4: mandel-tar.c with Point Strategy

After that, we have executed the *mandel-tar* binary using the *./run-tareador.sh* script indicating three different options:

- The first execution doesn't have any option.
- The second one has the -d flag (display flag).
- The third one executes using the -h flag (histogram flag).

<u>No options</u>

Executing the *mandel-tar* binary with the ./run-tareador-sh script without any options shows us the following TDGs:

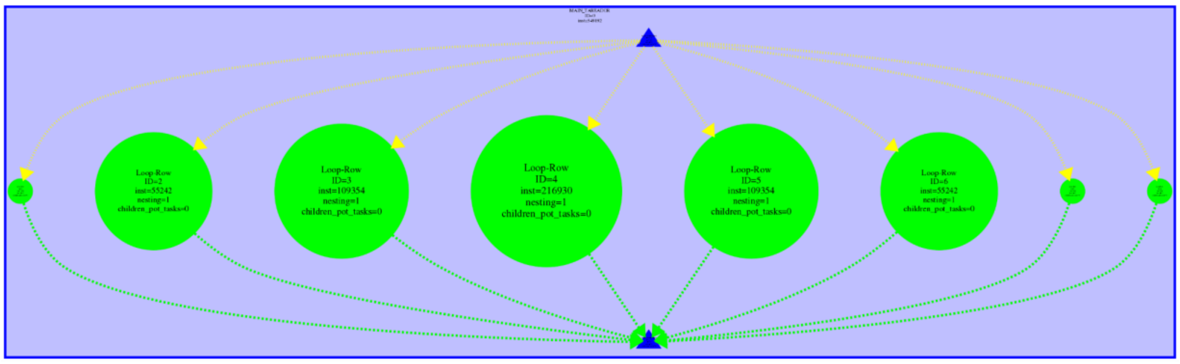


Figure 5: TDG of Row Decomposition without options

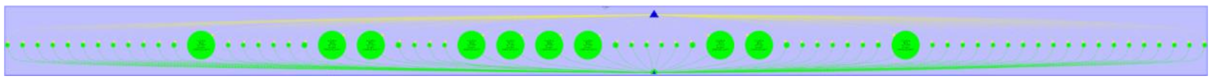


Figure 6: TDG of Point Decomposition without options

By observing the two images we can conclude that for both Task Dependency Graphs the tasks are embarrassingly parallel (the loop iterations are all parallelizable without dependencies) but they don't have the same amount of work (some tasks are bigger than others). In addition, the Point Strategy has more iterations than the Row one. This is normal because the Tareador's tasks in the Point Strategy are inside the innermost loop, while the Row ones are outside that loop.

“-d option”

Executing the *mandel-tar* binary with the ./run-tareador-sh script using the “-d” option generates the following TDGs:

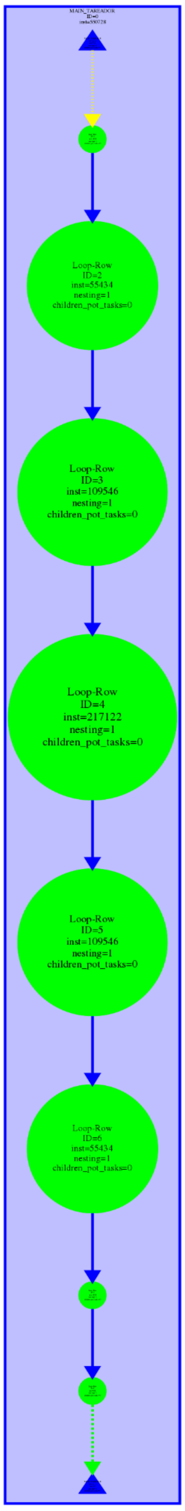


Figure 7: TDG of Row Decomposition with option -d

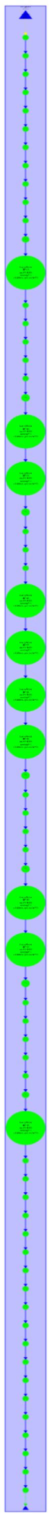
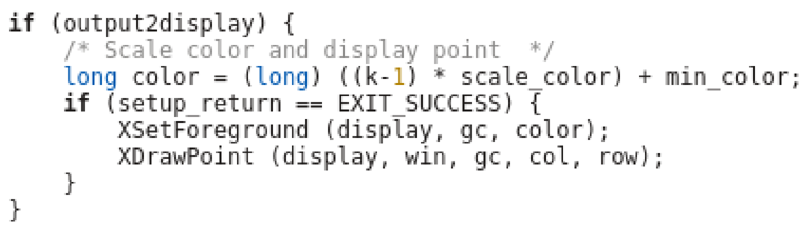


Figure 8: TDG of Point Decomposition with option -d

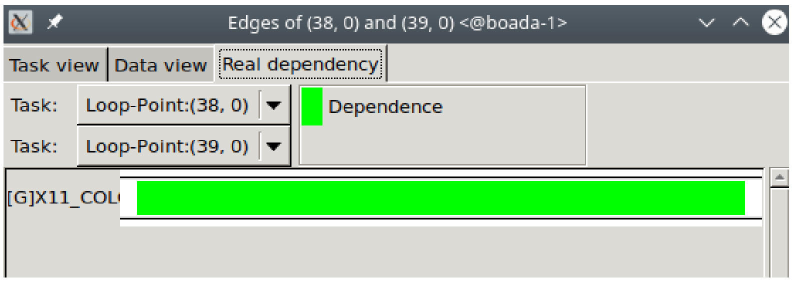
As we can see in the previous images, the TDGs have changed. Now, the task sizes remain the same but for both strategies the code is executing sequentially. This is due to the code that executes when we apply the -d option (we can see it in the next figure).



```
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 9: Code that executes with -d option

As we can see, when the -d option is activated the *output2display* variable is true and the program enters into this if. With Tareador we will analyze this part of the code in order to discover which variables generate dependencies:



Edges of (38, 0) and (39, 0) <@boada-1>

Task view | Data view | Real dependency

Task: Loop-Point:(38, 0)

Task: Loop-Point:(39, 0)

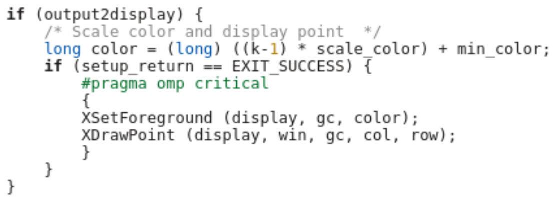
🟩 Dependence

[G]X11_COL

Figure 10: Variables that create dependencies using -d option

The XSetForeground and the XDrawPoint functions are the ones that generate dependencies between tasks.

To solve this, we will protect this part of the code of possible unwanted accesses and data races with the #pragma omp critical. Using this directive will ensure that all the threads wanting to enter this region will wait at the beginning until no other thread is executing a critical region. To see the resulting code, you can look at the following image.



```
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 11: Code executed with -d option protected with critical

<u>“-h option”</u>

Executing the *mandel-tar* binary with the ./run-tareador-sh script using the “-h” option generates the following TDGs:

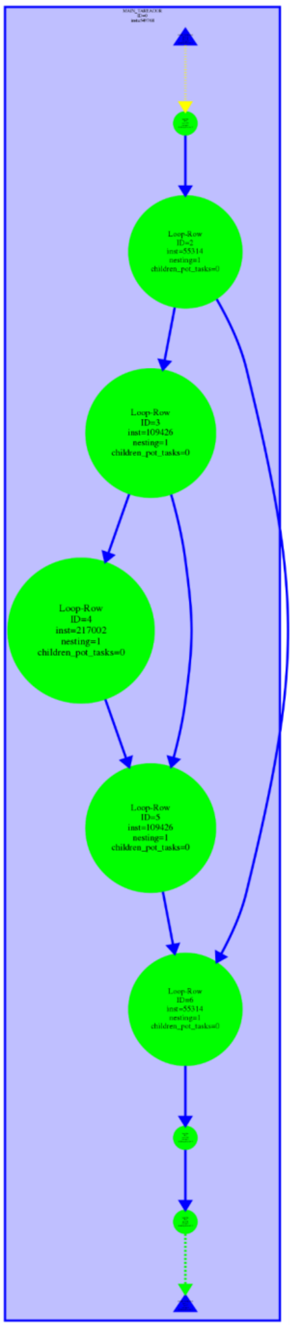


Figure 12: TDG of Row Decomposition with option -h

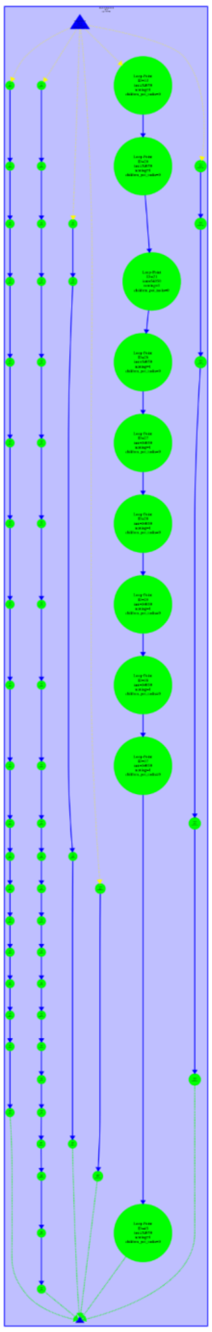
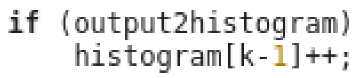


Figure 13: TDG of Point Decomposition with option -h

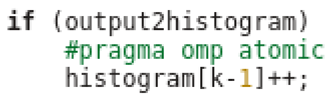
Comparing these “-h” option TDGs with the previous ones we can assert that they’re a lot different than them. We can observe that there are dependencies between different tasks and now we will discover why by taking a look at the code that executes when the “-h” option is enabled (shown in the next image).



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

Figure 14: Code that executes with -h option

As we can see in this code, the reason why there are dependencies is because there is an update. To protect this part, we have decided to use a #pragma omp atomic:



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

Figure 15: Code executed with -h option protected with atomic

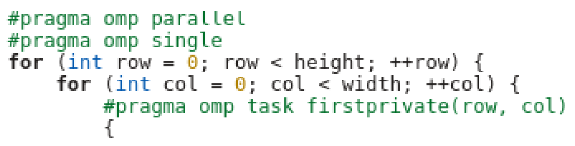
Using the atomic directive forces threads to do all their execution before another thread enters the atomic region and treats the variable as an indivisible part.

3. IMPLEMENTING TASK DECOMPOSITIONS IN OpenMP

3.1. Point decomposition

3.1.1. Point strategy using task

Firstly, we will start by applying the simplest way to create a task in OpenMP for the computation of each point in the Mandelbrot set: each iteration of the col loop will be executed as an independent task. Executing the code that uses this simple method we obtain these results:



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

Figure 16: Point decomposition using task

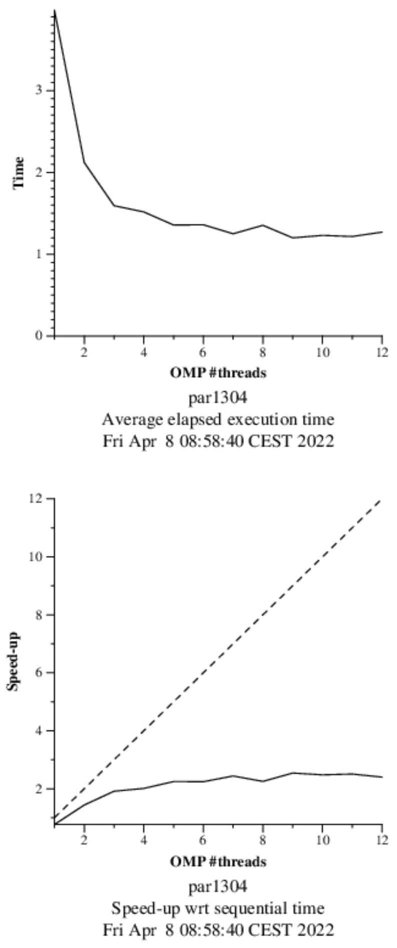
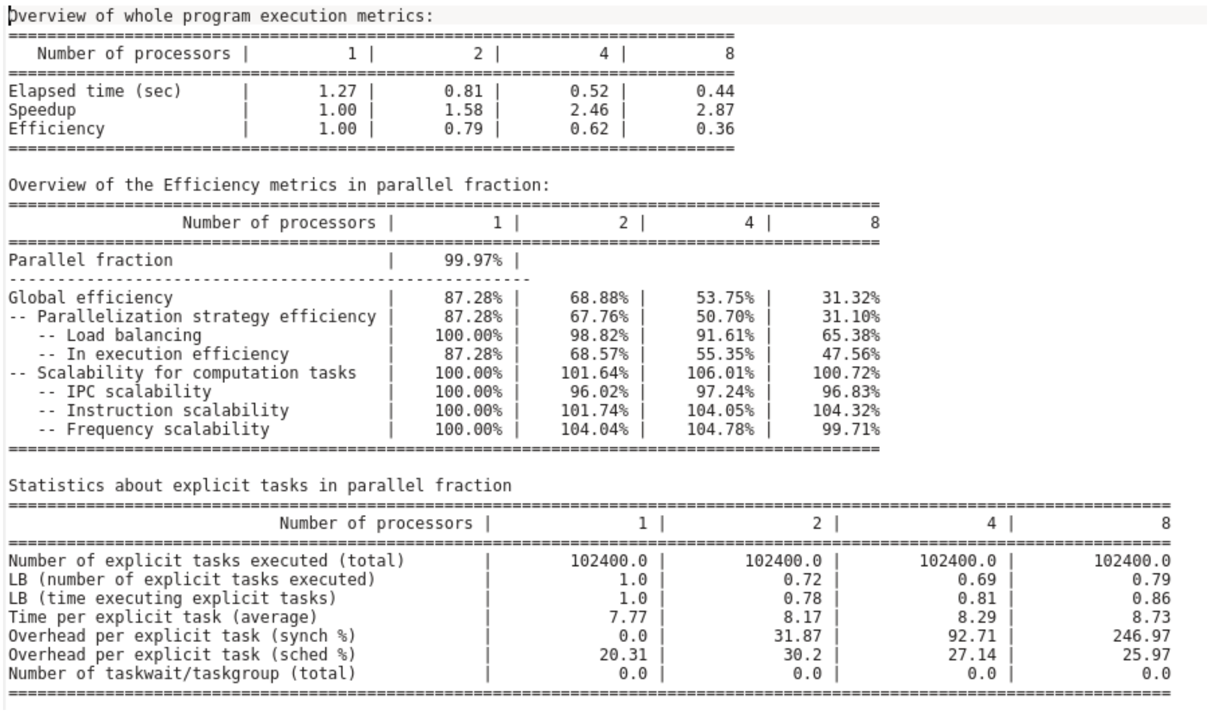


Figure 17: Execution time and speedup of point using task

Observing these plots we can affirm that using this version of the Point Strategy is far from the optimum performance of the program. It presents a bad scalability, the execution time stops reducing from the third thread and the speed-up doesn't improve with the increment of threads.

The parallelization strategy that provokes these bad results for the efficiency is *#pragma omp critical.* As we can see in the following table (in the statistics about explicit tasks in parallel fraction section) and in the window of the paraver, the main problem is the overhead per explicit task (synchronization). It grows exponentially with the augment of threads. We know that the problem are the overheads, so we have to increase the granularity.



```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |           1 |           2 |           4 |           8
===========================================================================
Elapsed time (sec)      |        1.27 |        0.81 |        0.52 |        0.44
Speedup                 |        1.00 |        1.58 |        2.46 |        2.87
Efficiency              |        1.00 |        0.79 |        0.62 |        0.36
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |           1 |           2 |           4 |           8
==========================================================================================
Parallel fraction                      |      99.97% |
------------------------------------------------------
Global efficiency                      |      87.28% |      68.88% |      53.75% |      31.32%
-- Parallelization strategy efficiency |      87.28% |      67.76% |      50.70% |      31.10%
   -- Load balancing                   |     100.00% |      98.82% |      91.61% |      65.38%
   -- In execution efficiency          |      87.28% |      68.57% |      55.35% |      47.56%
-- Scalability for computation tasks   |     100.00% |     101.64% |     106.01% |     100.72%
   -- IPC scalability                  |     100.00% |      96.02% |      97.24% |      96.83%
   -- Instruction scalability          |     100.00% |     101.74% |     104.05% |     104.32%
   -- Frequency scalability            |     100.00% |     104.04% |     104.78% |      99.71%
==========================================================================================

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 |              0.72 |              0.69 |              0.79
LB (time executing explicit tasks)               |               1.0 |              0.78 |              0.81 |              0.86
Time per explicit task (average)                 |              7.77 |              8.17 |              8.29 |              8.73
Overhead per explicit task (synch %)             |               0.0 |             31.87 |             92.71 |            246.97
Overhead per explicit task (sched %)             |             20.31 |              30.2 |             27.14 |             25.97
Number of taskwait/taskgroup (total)             |               0.0 |               0.0 |               0.0 |               0.0
========================================================================================================================
```

Table 1: Modelfactors table of point using task

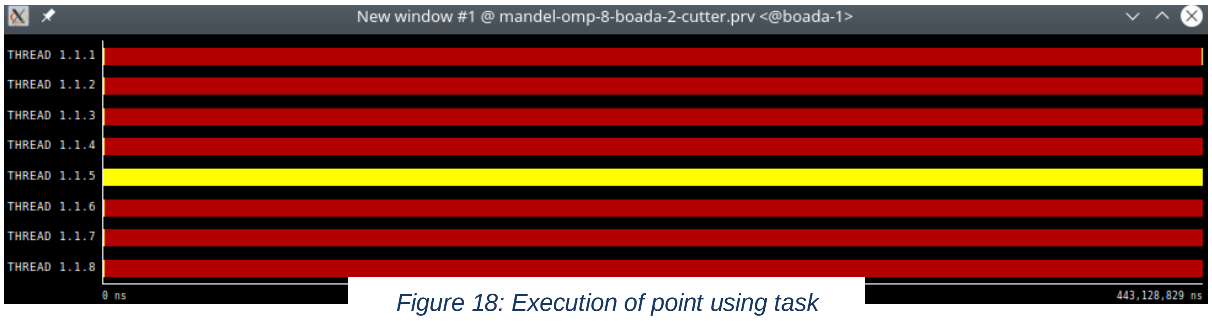
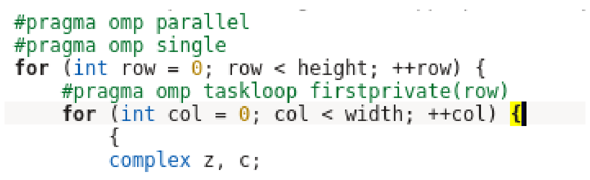


Figure 18: Execution of point using task

3.1.2. Point strategy with granularity control using taskloop

Now, let's try implementing the same strategy but using the *taskloop* clause. Executing this new code will show us if we made an improvement in performance.



```
#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;
```

Figure 19: Point decomposition using taskloop

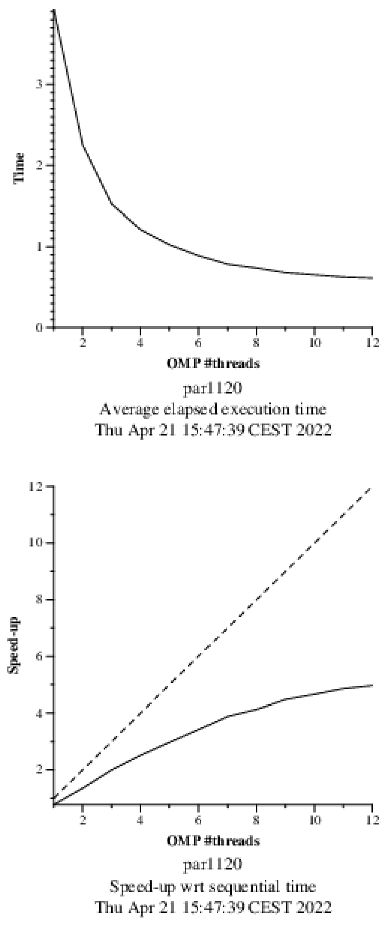
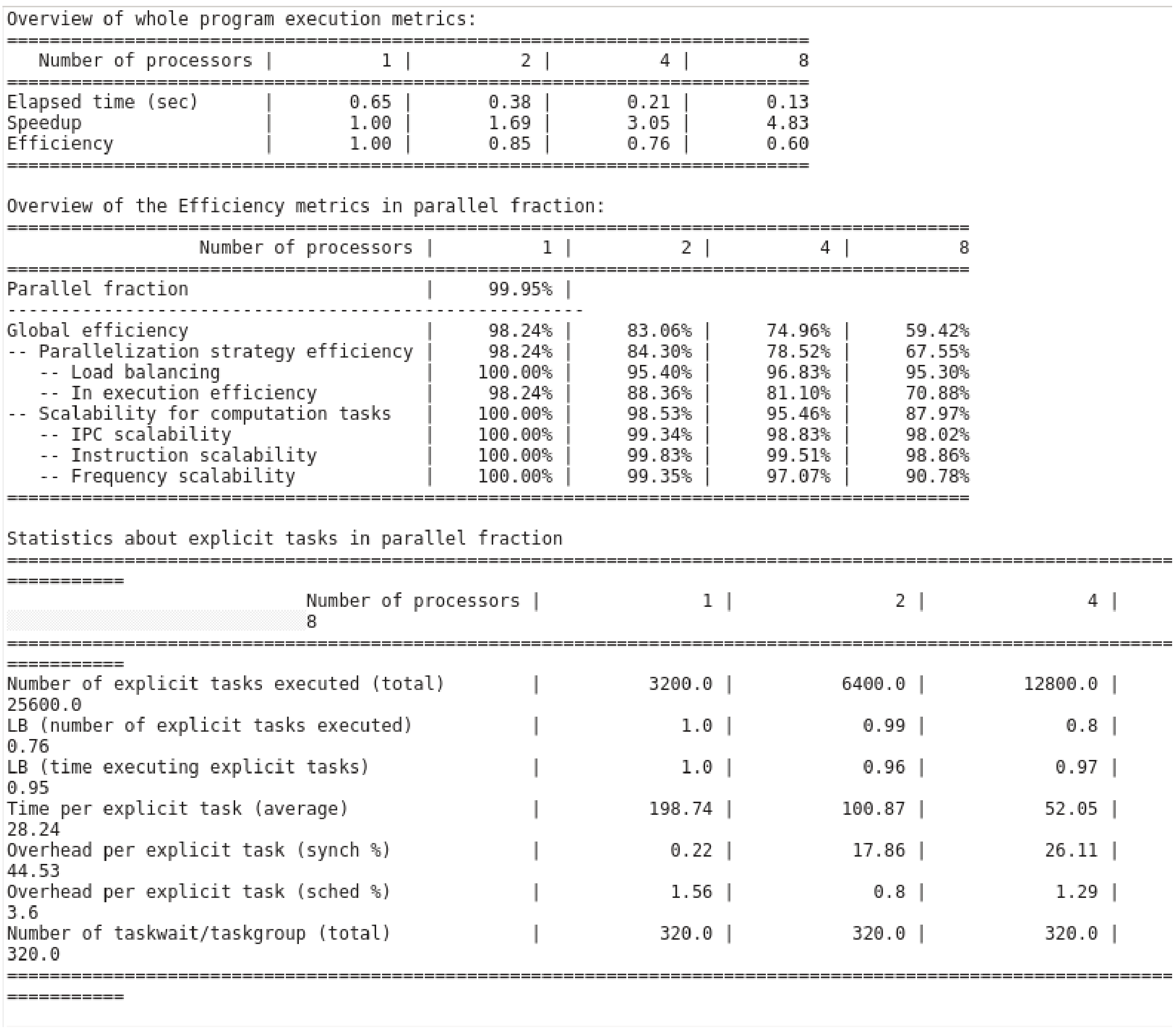


Figure 20: Execution time and speedup of point using taskloop

As we can see in the plots, the speedup increases continuously as the number of threads increases, but the last threads only perceive a minuscule improvement.

Even so, it’s scalability is better than using the same strategy without taskloop.



```
Overview of whole program execution metrics:
============================================================================
   Number of processors |          1 |          2 |          4 |          8
============================================================================
Elapsed time (sec)      |       0.65 |       0.38 |       0.21 |       0.13
Speedup                 |       1.00 |       1.69 |       3.05 |       4.83
Efficiency              |       1.00 |       0.85 |       0.76 |       0.60
============================================================================

Overview of the Efficiency metrics in parallel fraction:
=========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
=========================================================================================
Parallel fraction                      |     99.95% |
-----------------------------------------------------
Global efficiency                      |     98.24% |     83.06% |     74.96% |     59.42%
-- Parallelization strategy efficiency |     98.24% |     84.30% |     78.52% |     67.55%
   -- Load balancing                   |    100.00% |     95.40% |     96.83% |     95.30%
   -- In execution efficiency          |     98.24% |     88.36% |     81.10% |     70.88%
-- Scalability for computation tasks   |    100.00% |     98.53% |     95.46% |     87.97%
   -- IPC scalability                  |    100.00% |     99.34% |     98.83% |     98.02%
   -- Instruction scalability          |    100.00% |     99.83% |     99.51% |     98.86%
   -- Frequency scalability            |    100.00% |     99.35% |     97.07% |     90.78%
=========================================================================================

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 |            0.99 |             0.8 |
0.76
LB (time executing explicit tasks)               |             1.0 |            0.96 |            0.97 |
0.95
Time per explicit task (average)                 |          198.74 |          100.87 |           52.05 |
28.24
Overhead per explicit task (synch %)             |            0.22 |           17.86 |           26.11 |
44.53
Overhead per explicit task (sched %)             |            1.56 |             0.8 |            1.29 |
3.6
Number of taskwait/taskgroup (total)             |           320.0 |           320.0 |           320.0 |
320.0
==============================================================================================================
===========
```

Table 2: Modelfactors table of point using taskloop

We can see that, unlike in the previous table, the load balancing does not plummet after 8 threads and that in execution efficiency drops at a much slower rate. The problem is caused by the "in execution efficiency".

The number of explicit tasks is much lower than in the previous table and increases with the number of threads, on the other hand, the average time per explicit task is much higher. Finally, we see that the % of overheads are much lower thanks to the implementation of taskloop.

The reason for the high %synch in the overheads is the *taskgroup* implicit in the taskloop, which generates dependencies by causing waits for the completion of a task and all its descendants.

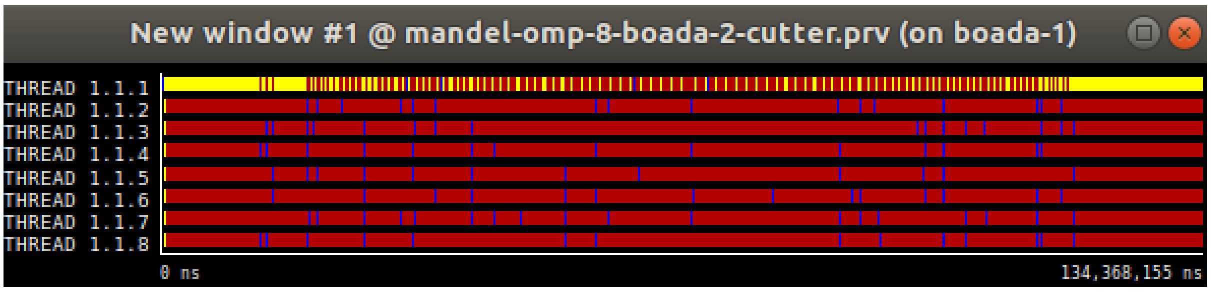


Figure 21: Execution of point using taskloop

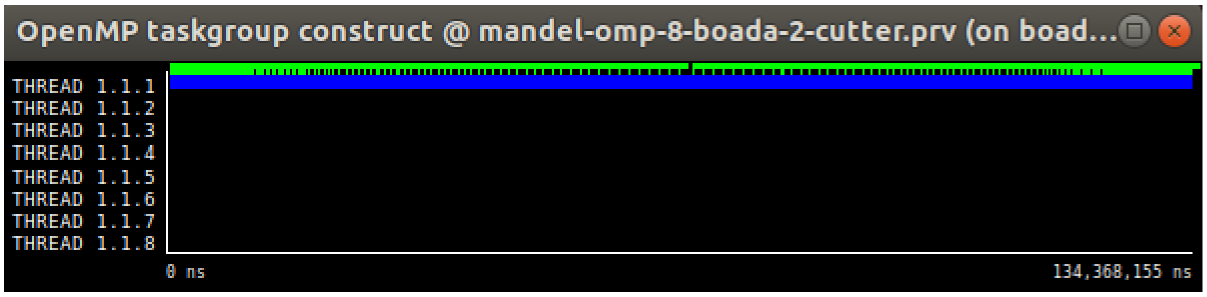


Figure 22: Execution of point using taskloop (taskgroups)

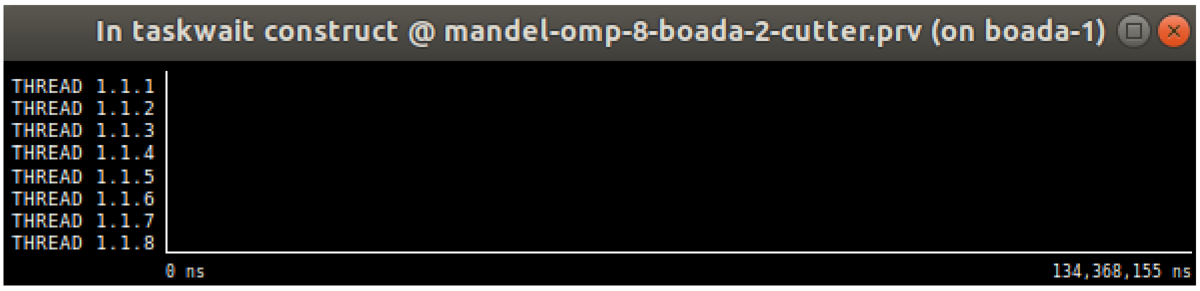
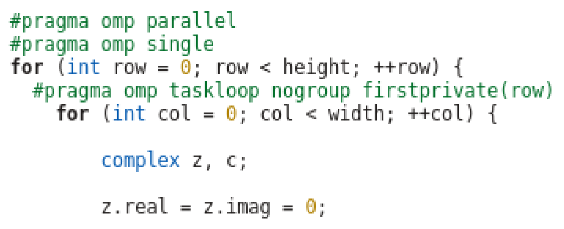


Figure 23: Execution of point using taskloop (taskwaits)

The upper image belongs to the execution of the program with *taskloop*, the middle one to the *taskgroups* in it, and the lower one to the *taskwait*. As we can see, the ones responsible for the program execution being dominated by synchronization tasks are the taskgroups implicit in the taskloop, since there is no presence of taskwait.

These taskgroups are not necessary, so we can eliminate them with the nogroup causula, with which we obtain the following results.



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

        complex z, c;

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

Figure 24: Point decomposition using taskloop nogroup

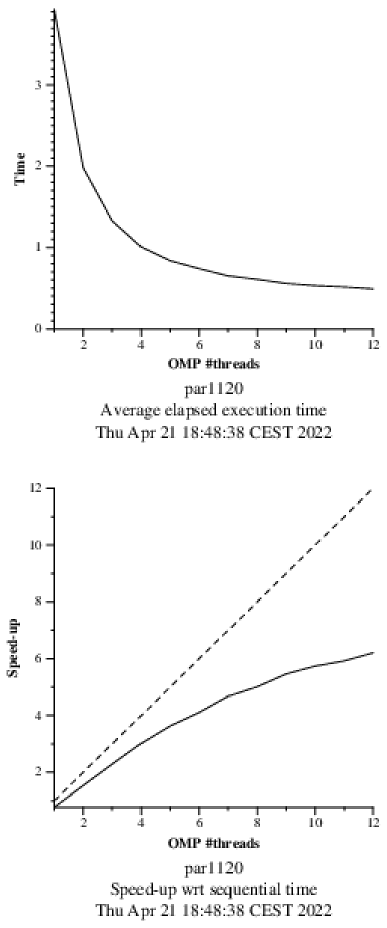


Figure 25: Execution time and speedup of point using taskloop nogroup

Thanks to the elimination of taskgroups, we eliminate their corresponding part of the program runtime, thus achieving better scalability.

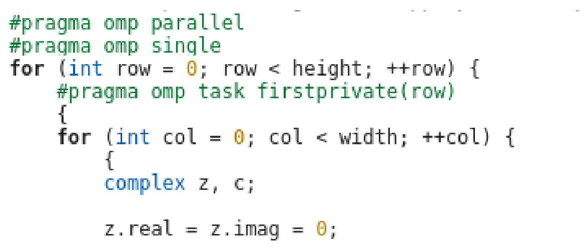
```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |         1 |         2 |         4 |         8
===========================================================================
Elapsed time (sec)      |      0.65 |      0.34 |      0.18 |      0.11
Speedup                 |      1.00 |      1.93 |      3.65 |      6.08
Efficiency              |      1.00 |      0.97 |      0.91 |      0.76
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |         1 |         2 |         4 |         8
==========================================================================================
Parallel fraction                      |    99.94% |
------------------------------------------------------
Global efficiency                      |    98.70% |    95.33% |    90.25% |    75.16%
-- Parallelization strategy efficiency |    98.70% |    96.88% |    94.33% |    87.65%
   -- Load balancing                   |   100.00% |    99.35% |    98.19% |    98.21%
   -- In execution efficiency          |    98.70% |    97.51% |    96.06% |    89.25%
-- Scalability for computation tasks   |   100.00% |    98.40% |    95.68% |    85.75%
   -- IPC scalability                  |   100.00% |    99.63% |    99.18% |    97.88%
   -- Instruction scalability          |   100.00% |    99.83% |    99.50% |    98.86%
   -- Frequency scalability            |   100.00% |    98.93% |    96.95% |    88.62%
==========================================================================================

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 |              0.77 |              0.46 |
0.95
LB (time executing explicit tasks)               |               1.0 |              0.99 |              0.98 |
0.98
Time per explicit task (average)                 |            199.17 |            101.19 |             52.03 |
29.03
Overhead per explicit task (synch %)             |               0.0 |              1.42 |              3.28 |
11.22
Overhead per explicit task (sched %)             |              1.32 |              1.79 |              2.74 |
2.88
Number of taskwait/taskgroup (total)             |               0.0 |               0.0 |               0.0 |
0.0
==============================================================================================================
===========
```

Table 3: Modelfactors table of point using taskloop nogroup

In this table, corresponding to the execution of the program with point strategy using taskloop with the nogroup clause, we see that now, the number of *taskwait/taskgroup* is 0, achieving an improvement in efficiency and speedup.

3.2. Row decomposition

Now that we have finished improving the point strategy, we will try changing it to row decomposition to see how it performs both with and without taskloop. In the following images, we can see the implementation without the *taskloop* clause and the results obtained by executing the code:



```
#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;
```

Figure 26: Row decomposition using task

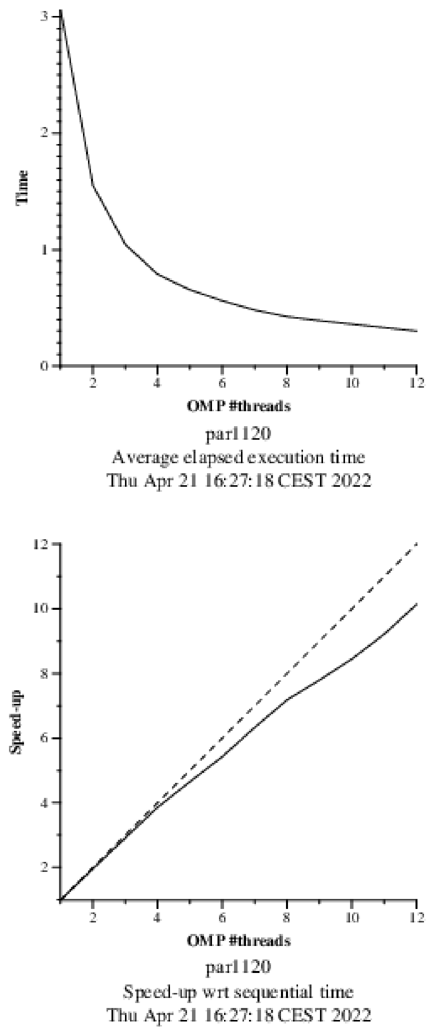
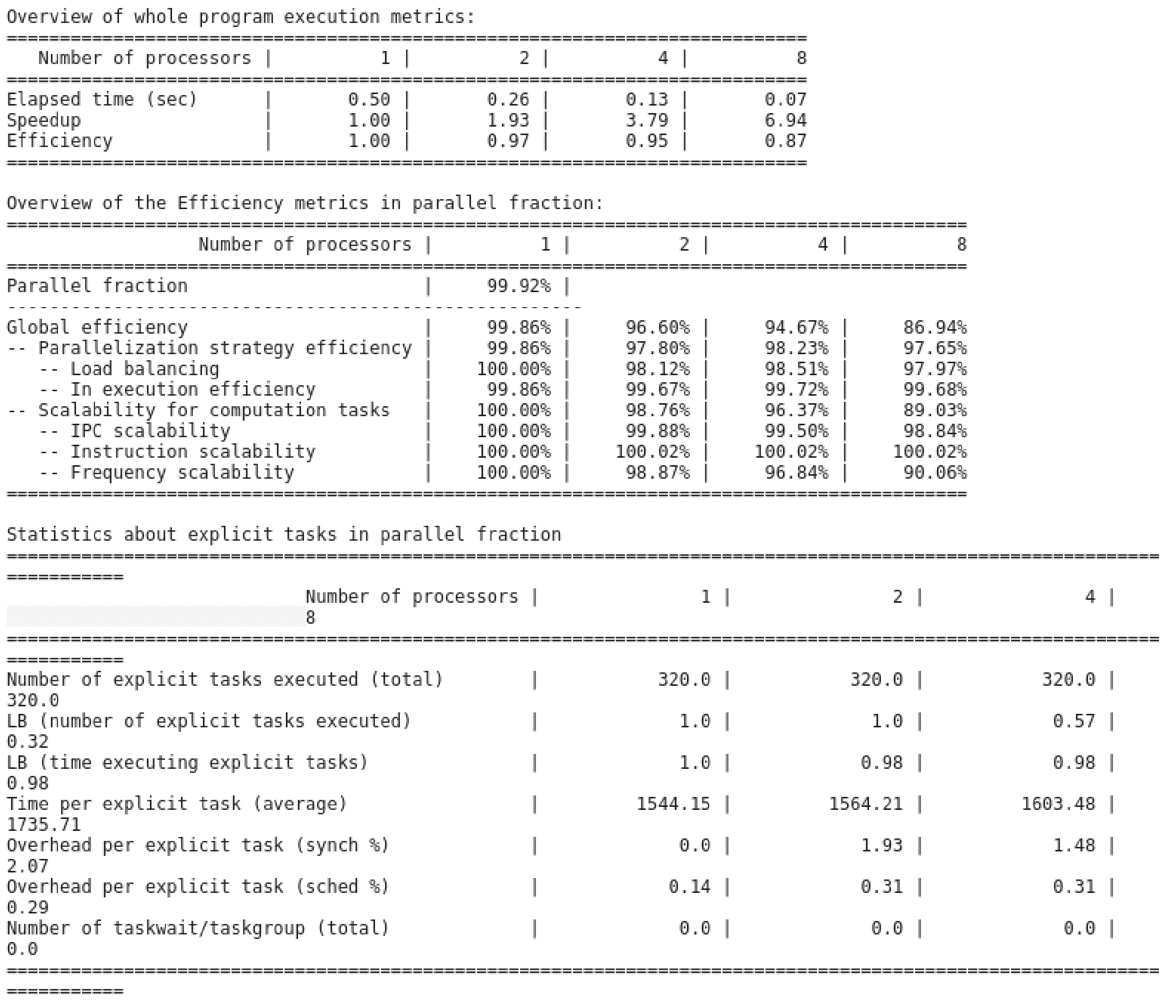


Figure 27: Execution time and speedup of point using task

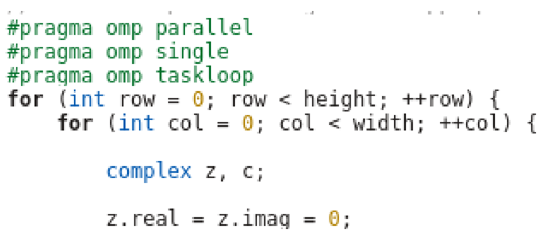
To begin with, we can see an abysmal improvement in terms of scalability with respect to the 3 previous versions of point strategy. In addition, in the following table we can also see that the efficiency has improved a lot. We can also notice that the time spent in overheads is practically null, being this the reason why row strategy is so close to what would be considered an optimal performance for the program.



```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |          1 |          2 |          4 |          8
===========================================================================
Elapsed time (sec)      |       0.50 |       0.26 |       0.13 |       0.07
Speedup                 |       1.00 |       1.93 |       3.79 |       6.94
Efficiency              |       1.00 |       0.97 |       0.95 |       0.87
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
==========================================================================================
Parallel fraction                      |     99.92% |
------------------------------------------------------
Global efficiency                      |     99.86% |     96.60% |     94.67% |     86.94%
-- Parallelization strategy efficiency |     99.86% |     97.80% |     98.23% |     97.65%
   -- Load balancing                   |    100.00% |     98.12% |     98.51% |     97.97%
   -- In execution efficiency          |     99.86% |     99.67% |     99.72% |     99.68%
-- Scalability for computation tasks   |    100.00% |     98.76% |     96.37% |     89.03%
   -- IPC scalability                  |    100.00% |     99.88% |     99.50% |     98.84%
   -- Instruction scalability          |    100.00% |    100.02% |    100.02% |    100.02%
   -- Frequency scalability            |    100.00% |     98.87% |     96.84% |     90.06%
==========================================================================================

Statistics about explicit tasks in parallel fraction
============================================================================================================
===========
                            Number of processors |                1 |                2 |                4 |
                            8
============================================================================================================
===========
Number of explicit tasks executed (total)        |            320.0 |            320.0 |            320.0 |
320.0
LB (number of explicit tasks executed)           |              1.0 |              1.0 |             0.57 |
0.32
LB (time executing explicit tasks)               |              1.0 |             0.98 |             0.98 |
0.98
Time per explicit task (average)                 |          1544.15 |          1564.21 |          1603.48 |
1735.71
Overhead per explicit task (synch %)             |              0.0 |             1.93 |             1.48 |
2.07
Overhead per explicit task (sched %)             |             0.14 |             0.31 |             0.31 |
0.29
Number of taskwait/taskgroup (total)             |              0.0 |              0.0 |              0.0 |
0.0
============================================================================================================
===========
```

Table 4: Modelfactors table of row using task

Let's test the execution of the program with row strategy using taskloop to see how it performs.



```
#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;

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

Figure 28: Row decomposition using taskloop

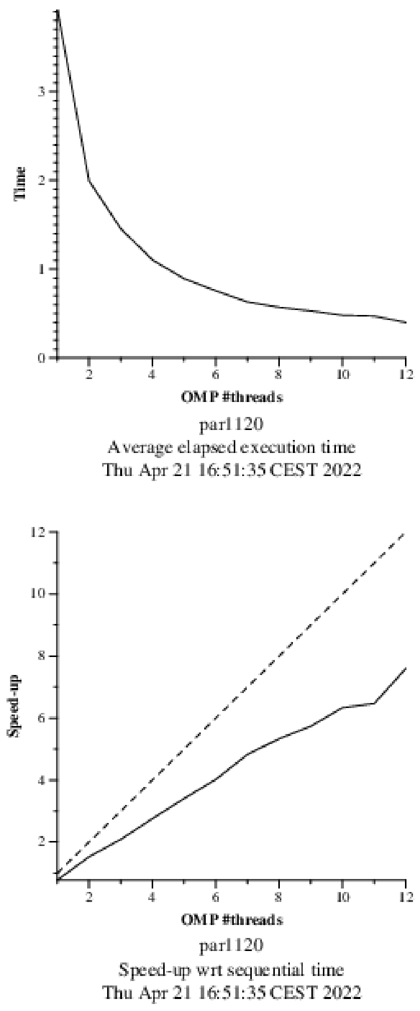
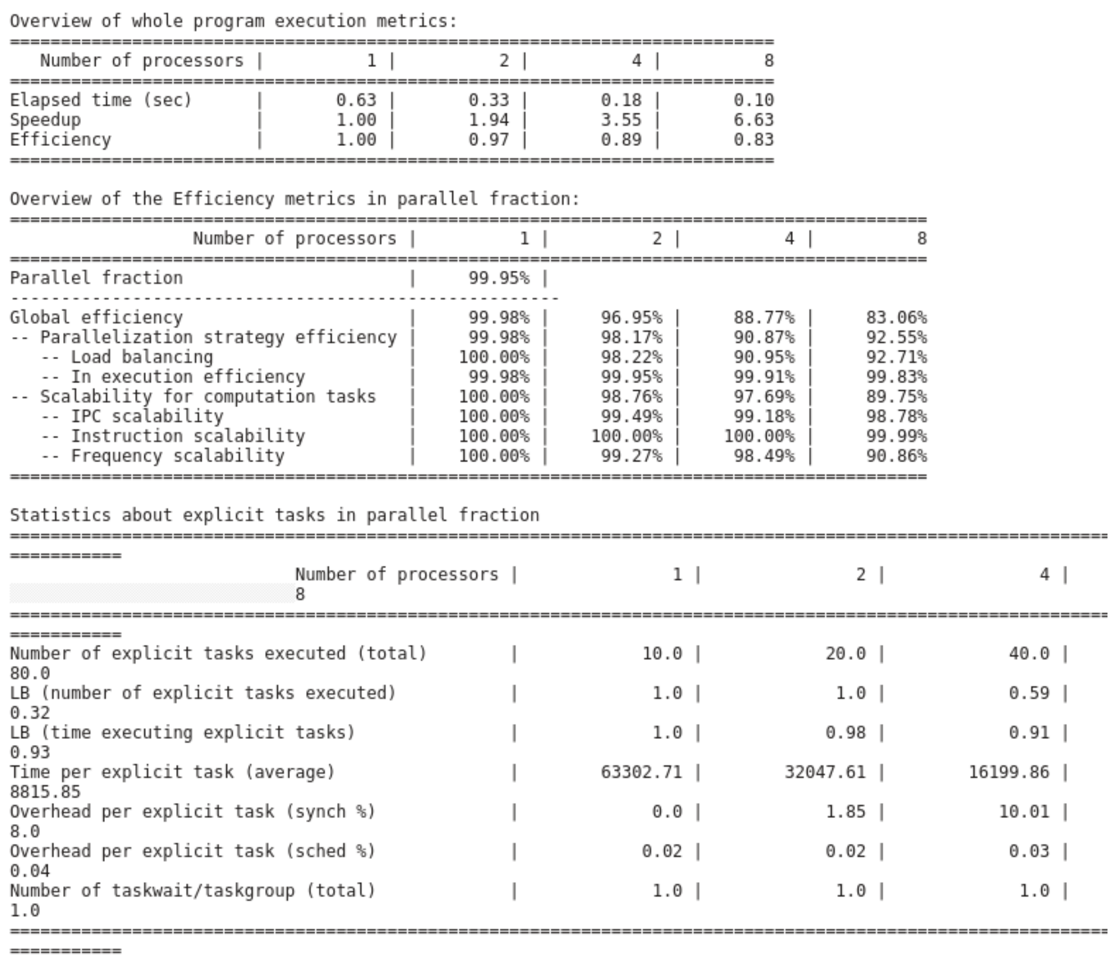


Figure 29: Execution time and speedup of point using taskloop

We can see that, in the case of row strategy, using taskloop what it does is to worsen the performance of the program, since this strategy invests very little time in synchronization and, with taskloop, we cause it to invest more time than necessary with the synchronization overheads.

Even so, the performance and scalability is still much better than the three versions of point strategy previously seen.



```
Overview of whole program execution metrics:
==========================================================================
   Number of processors |          1 |          2 |          4 |         8
==========================================================================
Elapsed time (sec)      |       0.63 |       0.33 |       0.18 |      0.10
Speedup                 |       1.00 |       1.94 |       3.55 |      6.63
Efficiency              |       1.00 |       0.97 |       0.89 |      0.83
==========================================================================

Overview of the Efficiency metrics in parallel fraction:
=========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
=========================================================================================
Parallel fraction                      |     99.95% |
-----------------------------------------------------
Global efficiency                      |     99.98% |     96.95% |     88.77% |     83.06%
-- Parallelization strategy efficiency |     99.98% |     98.17% |     90.87% |     92.55%
   -- Load balancing                   |    100.00% |     98.22% |     90.95% |     92.71%
   -- In execution efficiency          |     99.98% |     99.95% |     99.91% |     99.83%
-- Scalability for computation tasks   |    100.00% |     98.76% |     97.69% |     89.75%
   -- IPC scalability                  |    100.00% |     99.49% |     99.18% |     98.78%
   -- Instruction scalability          |    100.00% |    100.00% |    100.00% |     99.99%
   -- Frequency scalability            |    100.00% |     99.27% |     98.49% |     90.86%
=========================================================================================

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)                 |          63302.71 |          32047.61 |          16199.86 |
8815.85
Overhead per explicit task (synch %)             |               0.0 |              1.85 |             10.01 |
8.0
Overhead per explicit task (sched %)             |              0.02 |              0.02 |              0.03 |
0.04
Number of taskwait/taskgroup (total)             |               1.0 |               1.0 |               1.0 |
1.0
==========================================================================================================
===========
```

Table 5: Modelfactors table of row using taskloop

4. OPTIONAL

In this optional section we will explore the behavior of the Point and Row strategies for different granularities. To do this, we must adapt the mandel-omp.c file so that we can run the submit-numtasks-omp.sh script to do this work.

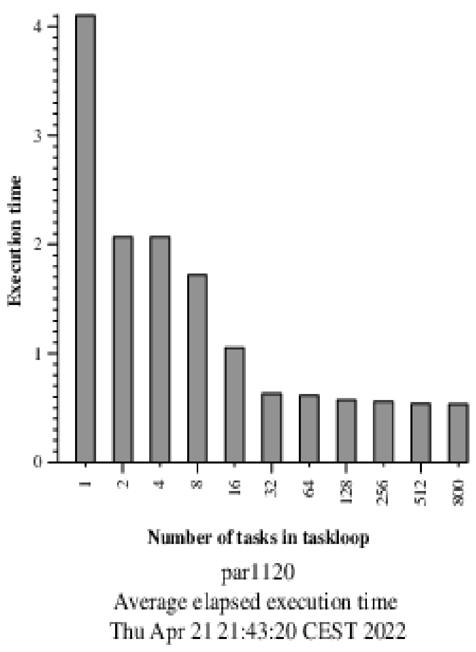


Figure 30: Execution time of row using taskloop

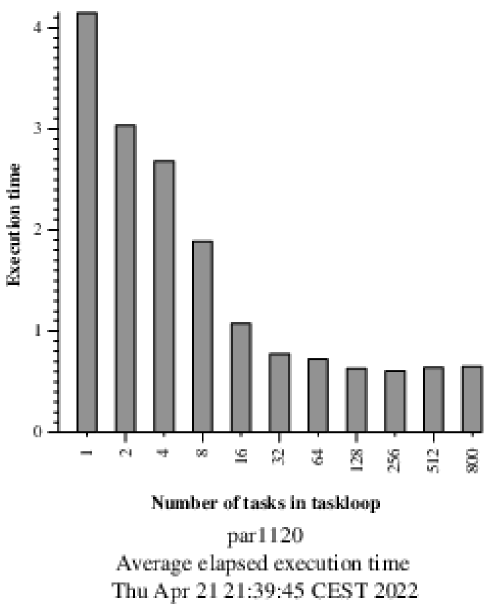


Figure 31: Execution time of point using taskloop

The change we had to make in mandel-omp.c was to read the fifth argument passed by terminal, which we passed to the mandelbrot function, so that we could set it as a parameter of the num_tasks clause.

We can see that as the number of tasks increases, as they can be parallelized, the execution time is significantly reduced. But at the point where it is no longer possible to parallelize in more processes, increasing the number of tasks is not only not an improvement, but it is detrimental due to the overheads.