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

E. Ayguadé, J. R. Herrero, P. Martínez-Ferrer,
J. Morillo, J. Tubella and G. Utrera Spring
2021-22

Maria Montalvo Falcón (par 4214)
Victor Pla Sanchis (par4219)
Group: 42
Date: 21/04/22
Course: 2021-2022 Q2

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Index

1

Introduction

In the following two laboratory sessions we are going to work with the Mandelbrot set code. The main purpose of these two sessions would be to analyse deeply how the implementation of the parallelisation could affect, limit or improve the performance of the initial sequential program. For this, initially we will understand the sequential code and apply two different implementations with *Tareador:* row strategy and point strategy.
Then, we will start to work with the *Paraver* tool and we will see the behaviours of: point strategy with task, point strategy with taskloop and row strategy. In this part, we will generate different diagrams and plots explaining the time execution, parallelisation, work per thread or speed-up among others to really understand the performance of the different implementations.

Finally in the optional part, we will study the granularity of the implementations and support our conclusions with plots and *Paraver* tools. After all, we will describe our main conclusions of the work.

2

Task decomposition strategies

In this section the main tool would be *Tareador,* so we can analyse the different strategies. The first strategy seen in section 2.1 will be the implementation with row tasks meanwhile the second one in section 2.2 will be the point strategy. In both sections we will show the dependency graphs and comment about the main conclusions of the strategies.

2.1 Row strategy implementation

The first strategy we came up with is the **row strategy**. This way to parallelise the mandelbrot program allows us to create a task for every row of the output image. In order to create this option we have added to the code the lines: tareador_start_task("ROW TASK") to indicate the starting point of the task and tareador_end_task("ROW TASK") at the end of the same as the following image shows (see figure 2.1.1).

```
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 TASK");
        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("RUN TASK");
    }
}
```

Figure 2.1.1 Row strategy code with Tareador

When we execute without any option, the program can be well parallelised. There is no dependency between row tasks, despite this, we can see on Figure 2.1.2 that the task computation is not that well shared. That inequality in proportions of work depends on the subset of the points that each row task has to compute. The rows in the middle are the ones that have to calculate and draw more points, whether the ones on the beginning and in the end have less computational work; it makes sense as the Mandelbrot set drawing fits with that description.

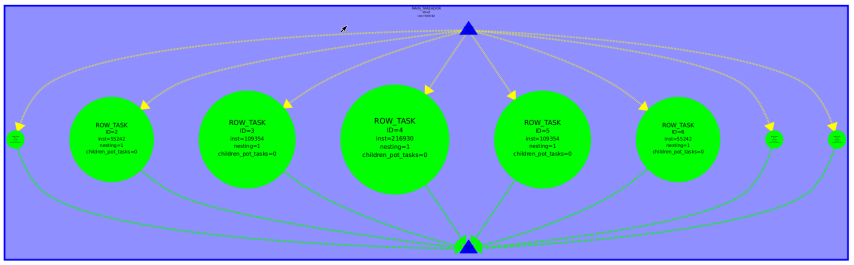

Figure 2.1.2 Row TDG without options

When we decided to execute our program with -d option (to display the results on the screen as an image), we saw that now we have created some dependencies on TDG. That's because now we are executing a new part of the code, that code executes 2 methods of X11 libraries (Xlib.h, Xutil.h and Xos.h) that creates a data dependency. As we see below, now our program becomes a sequential program, because all our tasks depend on the last one that is executed, we must ensure to parallelise the current tasks to get an improvement with -d option.

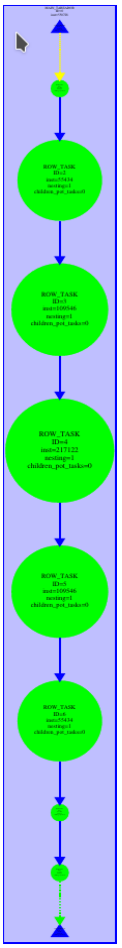

Figure 2.1.3 Row TDG with option -d

Tareador warned us that the variable named “color” is causing that dependency between row tasks see Figure 2.1.4. That variable is declared inside of the double for loop, exactly inside of the if statement that it’s executed with the -d option. Variable “color” generates dependency because it is used on the XSetForeground method and in XDrawPoint, both methods are creating that task dependency mentioned.

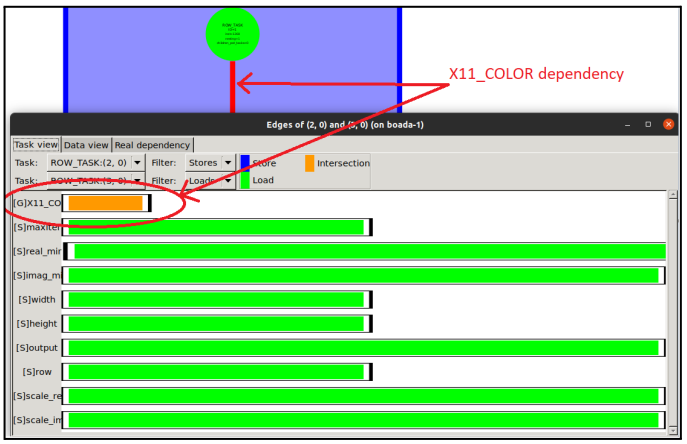

Figure 2.1.4 Dependency on variable X11_COLOR

Finally, when we execute the mandelbrot program with the option -h we still have a sequential program as the execution with the option -d, but we now have some new dependencies (it behaves as sequential because of the critical path). We are executing a new if statement that modifies a global variable, named “histogram”. Not all tasks modify the same element of “histogram”, just the tasks that compute the same value on the mandelbrot set. So these dependencies depend on the subset of the mandelbrot set that we want to compute. We see in the image below how a TDG will look like with the option -h.

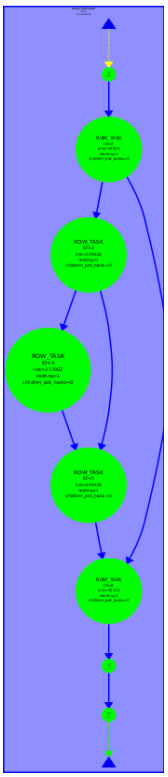

Figure 2.1.5 Row TDG with option -h

2.2 Point strategy implementation

After analysing the first code with row strategy implementation, we moved on to start analysing the point strategy one. In this case, the modifications of the code follows as below.
We have added the line tareador_start_task("POINT TASK") on the inner loop to indicate the starting point of the task and the line tareador_end_task("POINT TASK") at the ending of the loop in order to mark the end, see Figure 2.2.1.

```
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 TASK");
            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 TASK");
        }
    }
}
```

Figure 2.2.1 Point strategy code with Tareador

By executing the code modified, the number of tasks created are higher than in the row strategy as we see on Figure 2.2.2. This is because as we have said, in this case the creation of tasks is in the inner loop. Moreover, as in the case with row strategy implementation, we can see that tasks do not depend on the others, and that there are some tasks that are computing more points than the other ones. It is because there are tasks that are in a critical area of calculation of the fractal and have to compute more points.

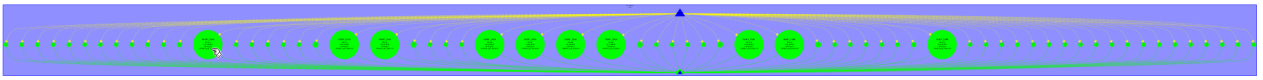

Figure 2.2.2 Point TDG without options

When executing the point strategy implementation with -d option, it occurs as in the case of row

strategy one, the execution turns up to become sequential (in this case, with more tasks because the point strategy is applied). The reason why this happens, is because of the variable “color” as we have explained earlier.

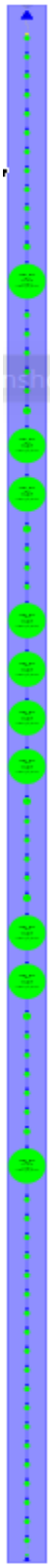

Figure 2.2.3 Point TDG with option -d

Finally, the last execution was made with the point strategy implementation and -h option. In this case, the TDG created was a bit different from the row strategy one. One could say that there is a bit of parallelisation on the code, but in fact, this code is sequential as the big tasks are making the critical path.

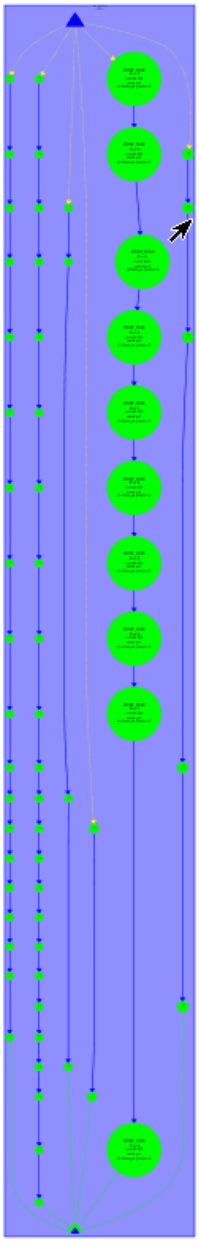

Figure 2.2.4 Point TDG with option -d

To conclude after studying all the behaviours, we can say that in general the row strategy implementation is better, as the code is behaving in a similar way for both, row and point with options -d and -h and in the case of the point strategy implementation the overhead of the creation of tasks and the synchronisation between them is much higher than in the row strategy one.

3

Implementation in OpenMP and performance analysis

In this section we will see the implementation of both strategies that we described in section 2 with OpenMP directives. The performance of the implementation would be analysed at the same time that strong and weak scalability is studied. In addition, we are going to synchronise the data shared between threads in order to avoid problems of data race condition, with the better option of parallelising.

3.1 Point strategy implementation using task implementation

Given the OpenMP template code, in this first part, we had to modify the code to protect the data shared between threads to avoid data race conditions. As we saw, we have to assure the correct writing on histogram[k-1]; to do that we used *#pragma omp atomic* directive because it is more efficient than *#pragma omp critical* one. Also, we had to preserve the correct access to variable "color" on X11 library calls: XSetForeground and XDrawPoint. For that, we used the *#pragma omp critical* instead of *#pragma omp atomic,* because atomic can't be applied to a pragma region but only to a variable.

```
if (output2histogram) {
    #pragma omp atomic
    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 3.1.1 Point strategy using task with data protection modifications

Following these modifications, we have compiled and executed the program to check if the result was correct. Also, we have compared the output with the cmp command on the terminal to see if there was any difference.

```
par4219@boada-1:~/lab3$ cmp output_seq.out output_omp_8.out
par4219@boada-1:~/lab3$ cmp output_seq.out output_omp_1.out
par4219@boada-1:~/lab3$
```

Figure 3.1.2 cmp command for output of point strategy with task

After checking this and seeing the modifications were correct, we submitted the work to analyse the time execution and saw how the code behaves with 1 up to 8 threads and with all options.

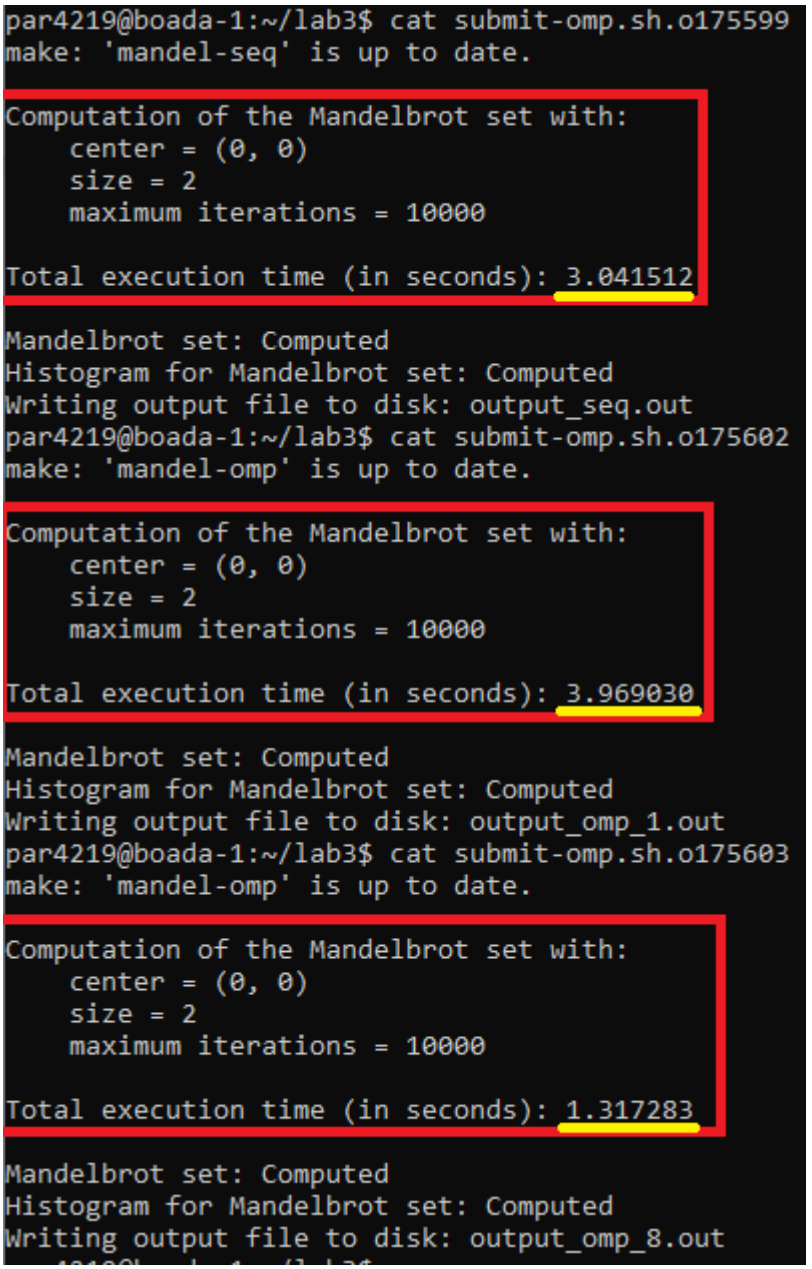

Figure 3.1.3 Time executions of point strategy using task

Point task	Sequential Mandelbrot	Mandelbrot omp 1 thread	Mandelbrot omp 8 threads
Execution time	3.0415 seconds	3.9690 seconds	1.3172 seconds

Table 3.1.1 Time executions of point strategy using tasks (units in seconds)

Analysing Table 3.1.1 we can see the overhead created when parallelising by comparing sequential Mandelbrot code and Mandelbrot omp with one thread. We can see that the increase in time with the parallelisation with one thread does not reach the second. Moreover, the performance of the program with parallel code and 8 threads decreased to 1.3 seconds which is a real improvement.
Also, we can see in the figure below the plots with the information of the performance of the program from 1 up to 12 threads. In the average elapsed execution time we can see that the time reaches a ceiling between 1.3 and 1.5 seconds. This time can't be decreased by adding more threads, it means that in a point, it remains constant. The reason why this happens could be the increasing overheads of the creation of tasks and eachs synchronisations.
On the other hand and as expected, if the time reaches a ceiling the same happens with the speedup. The speedup that could be reached in this case, is rounding the value of 2, and as for the time execution, it can't be improved by adding more threads as we can see on Figure 3.1.4. So we can say that in this case, the strong scalability of the program is far from being optimal.

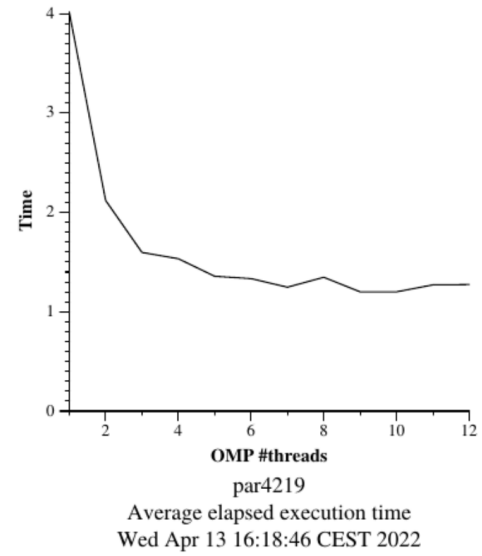

Figure 3.1.4 Plots of time execution and speed-up with point task strategy

Let's now study how parallelism works in our program. To do that, we will use the paravear program that is going to show us the detailed information with visual tools, so we executed the command *sbatch ./submit-extrae.sh* to get the .prv file.
First of all we are going to see the diagrams for worksharing construct and implicit tasks in parallel construct (see figure 3.1.5). The worksharing construct shows us when the single region is activated, meanwhile the implicit tasks in parallel construct shows us the task line number.
In this case we can see how thread number 4 is the one entering in the single region and the creator of the tasks.

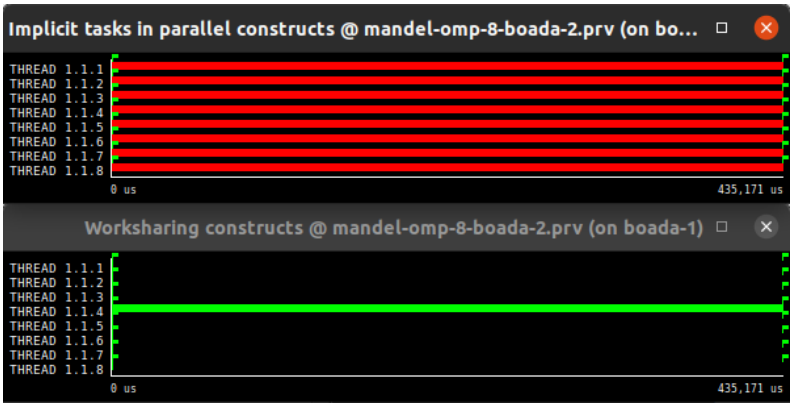

Figure 3.1.5 Implicit tasks in parallel constructs and worksharing for point strategy with task

Next we have created the histogram that shows the creation and the execution of tasks. By analysing, it is shown again that the thread creating the tasks is number 4. The total number of tasks create it's 102.400 and the average per thread is 12.800. All threads except the creator of the tasks are executing about 14.000 tasks, which means that the number of tasks is balanced between threads except for one.

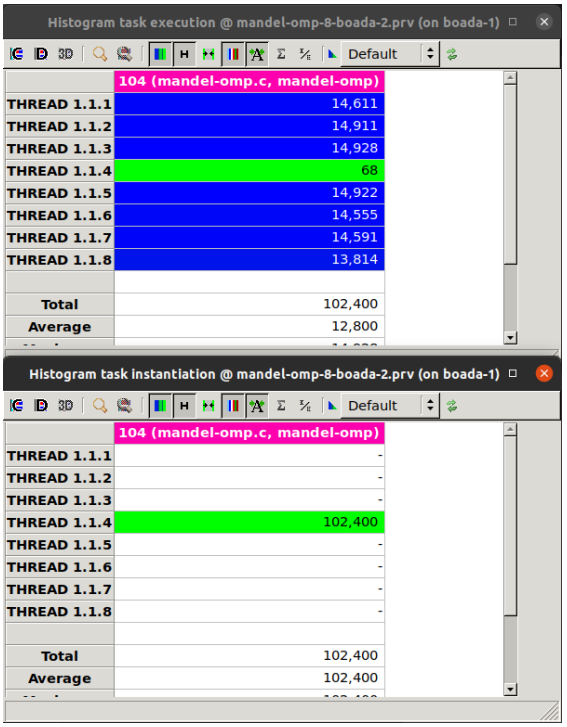

Figure 3.1.6 Profile of explicits tasks created and executed per thread.

One more, we clearly see the same behaviour for explicit tasks histogram with time mesure. In the figure below the thread creating tasks, which is number 4, is the one executing fewer tasks and have less time of work, but the rest of threads are well balanced.

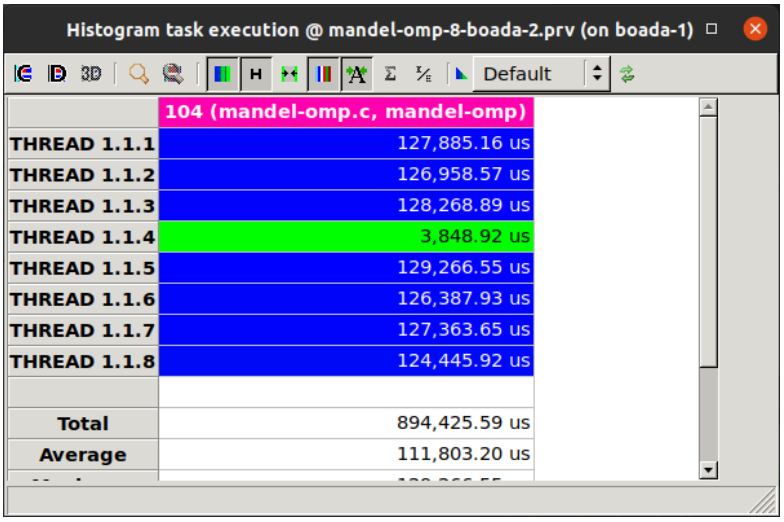

Figure 3.1.7 Explicit tasks time execution per thread.

Finally now, we have created the explicit tasks execution duration histogram in which we can see that thread 4 is creating tasks and then the rest of threads are executing them. We see that the work, as said before, is well balanced but it is observed that there are barriers in which the threads are waiting and not doing useful work.

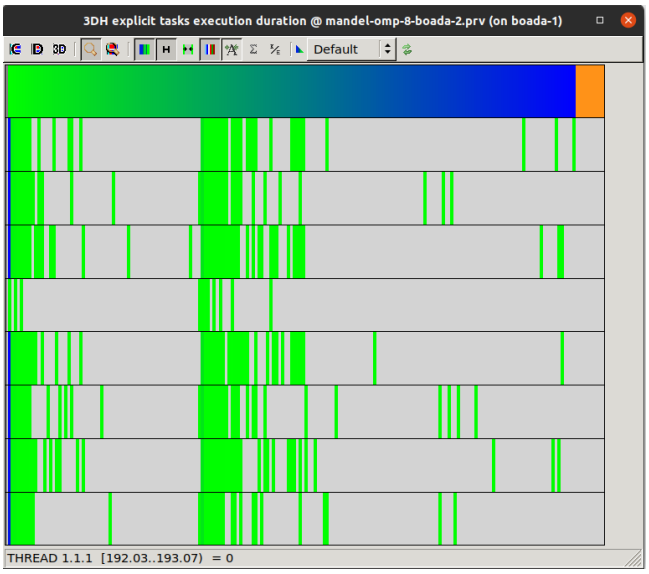

Figure 3.1.8 Every explicit task execution duration per thread.

3.2 Point strategy with granularity control using taskloop implementation

In this second part we are focusing on the implementation of point strategy using taskloop. To generate

this, we have added the line #pragma omp taskloop as seen in the image below. We have maintained the directives of #pragma omp atomic to protect the access to histogram[k-1] and the #pragma omp critical to protect the calls that use the “color” variable as we have explained in the previous section.

```
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
    #pragma omp parallel
    #pragma omp single
    for (int row = 0; row < height; ++row) {
        #pragma omp taskloop
        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) {
                #pragma omp atomic
                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 3.2.1 Code for mandel-omp point strategy with taskloop

After these modifications again, we checked our result with the cmp command as shown below.

```
par4219@boada-1:~/lab3$ cmp output_seq.out output_omp_1.out
par4219@boada-1:~/lab3$ cmp output_seq.out output_omp_8.out
par4219@boada-1:~/lab3$
```

Figure 3.2.2 cmp command for output of point strategy with taskloop

To analyse this new implementation, we have studied the time execution without any option, with 1 thread and with 8 threads to see again the behaviour of the program.

```
par4219@boada-1:~/lab3$ cat submit-omp.sh.o175631
make: 'mandel-seq' is up to date.

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

Total execution time (in seconds): 3.036375

Mandelbrot set: Computed
Histogram for Mandelbrot set: Computed
Writing output file to disk: output_seq.out
par4219@boada-1:~/lab3$ cat submit-omp.sh.o175629
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): 3.048532

Mandelbrot set: Computed
Histogram for Mandelbrot set: Computed
Writing output file to disk: output_omp_1.out
par4219@boada-1:~/lab3$ cat submit-omp.sh.o175630
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): 0.583856

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

Figure 3.2.3 Time executions point strategy with taskloop

Point taskloop	Sequential Mandelbrot	Mandelbrot omp 1 thread	Mandelbrot omp 8 threads
Execution time	3.036375 seconds	3.048532 seconds	0,583856 seconds

Table 3.2.1Time executions of point strategy using taskloop

In this case clearly the overhead of creating a task is much lower than in the previous implementation. Comparing the time of sequential and mandelbrot omp 1 thread, it is shown that the overhead of parallel execution with one thread it's about 0.01 seconds.
Also again, we can see on the parallel code with 8 threads, that the time execution decreases in this case, to almost 0.6 seconds.
Analysing the plots below, it is plain to see that the average elapsed execution time decreased by adding more threads; the ceiling point in this case is not as clear as before, but the marginal decrease slowly becomes lower and stabilises.
At the same time, the speedup of the program seems to increase by adding more threads and the

marginal increase slowly decreases. Comparing this performance with the one before, we can say that in this case the strong scalability of the program is better with point strategy and taskloop than with task at the beginning, but then stabilises.

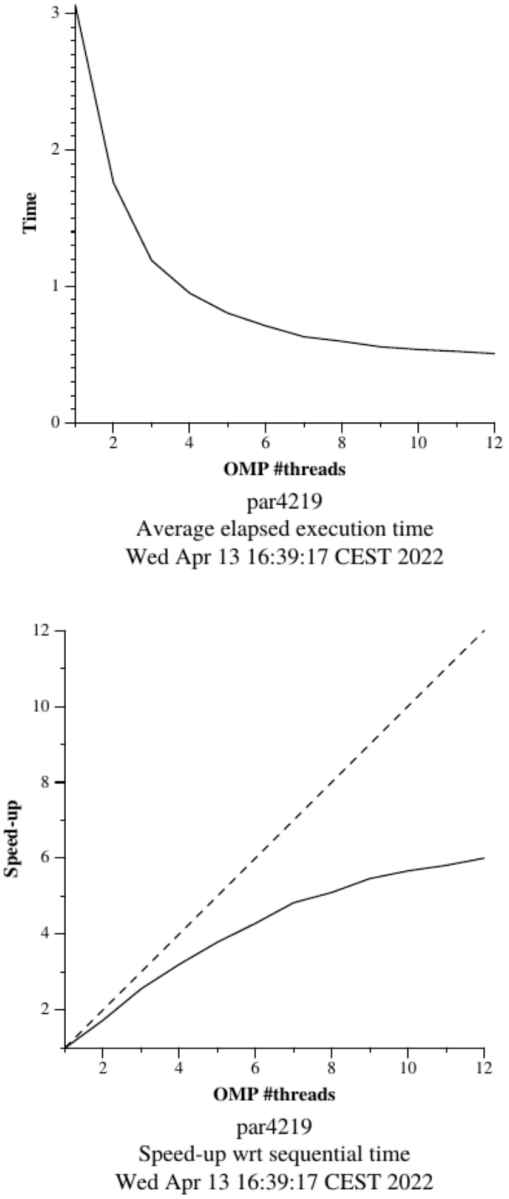

Figure 3.2.4 Plots of time execution and speed-up with point taskloop strategy

After this first analysis, we are going to study this implementation again with the Paraver interface. The first thing we are going to generate is the new window diagram.
In this diagram the yellow colour represents the time spent in the creation of tasks (thread 4 responsible for that), in colour red the time spent in the synchronisation of tasks and finally in colour blue, the time of real useful computing work.
Then we created the diagrams of worksharing constructs and implicit tasks in parallel constructs. As we have said previously, the worksharing constructs diagram is showing the thread that is creating the tasks that we will execute on the program. Below, we see that in this case the thread responsible for this is thread number 4.

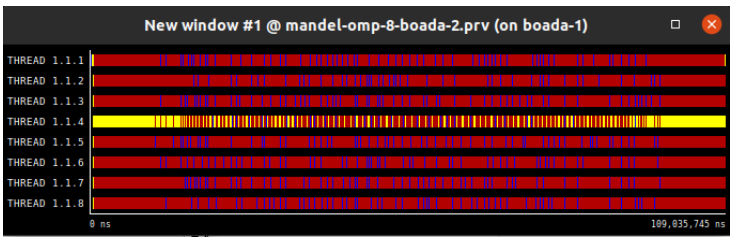

Figure 3.2.5 Parallelism execution on point strategy with taskloop

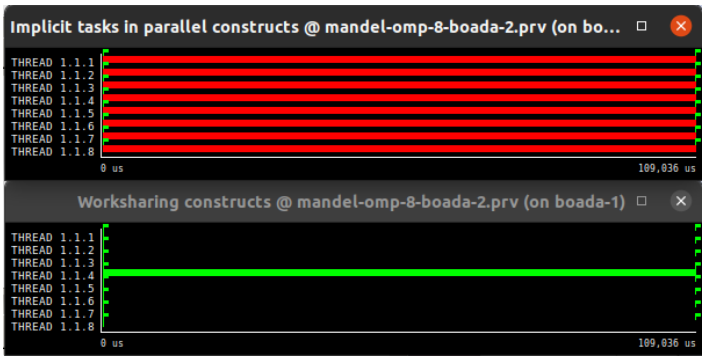

Figure 3.2.6 Implicit tasks in parallel constructs and worksharing for point strategy with taskloop

Now we study the histogram task execution. In this case we see that thread number 4, the responsible of the creation, is also executing a higher number of tasks.
Again it seems that the number of tasks is well balanced between threads except for number 4 that is executing a little more.

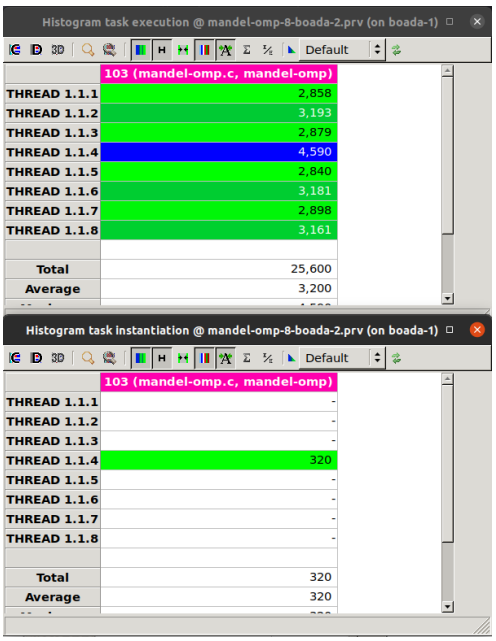

Figure 3.2.7 Profile of number of tasks created and executed per thread of point strategy with taskloop

Apart from the histogram of the number of tasks, we can study the execution of the program by the perspective of time of execution of threads. If we analyse that, we see how thread 4 spent less time, this is because the creation of tasks is a fast process (anyway is better balanced than the previous implementation with task). The rest of the threads have a balanced time of execution because the tasks are well shared and so it is the synchronisation.

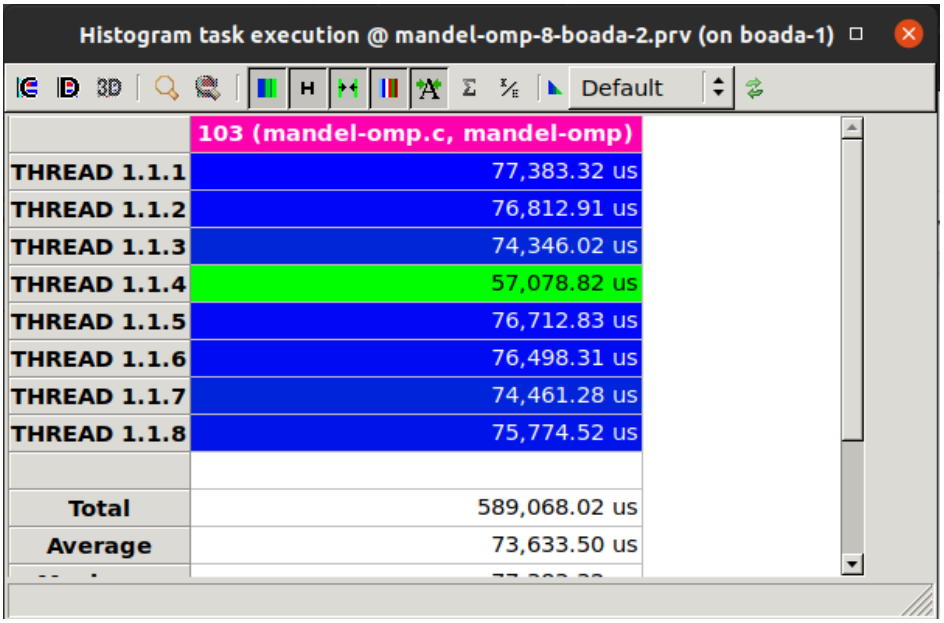

Figure 3.2.8 Explicit tasks time execution per thread of point strategy with taskloop

Finally we looked at the timeline of execution. Again it seems like an implicit barrier is making threads waiting without doing real work. As the tasks are waiting for all of their descendents, we have concluded that the implicit barrier applied to the *taskloop*, in this case is, *taskgroup*. We thought that the barrier was not necessary as there is no dependency to calculate a point of the Mandelbrot set without options -h and -d.

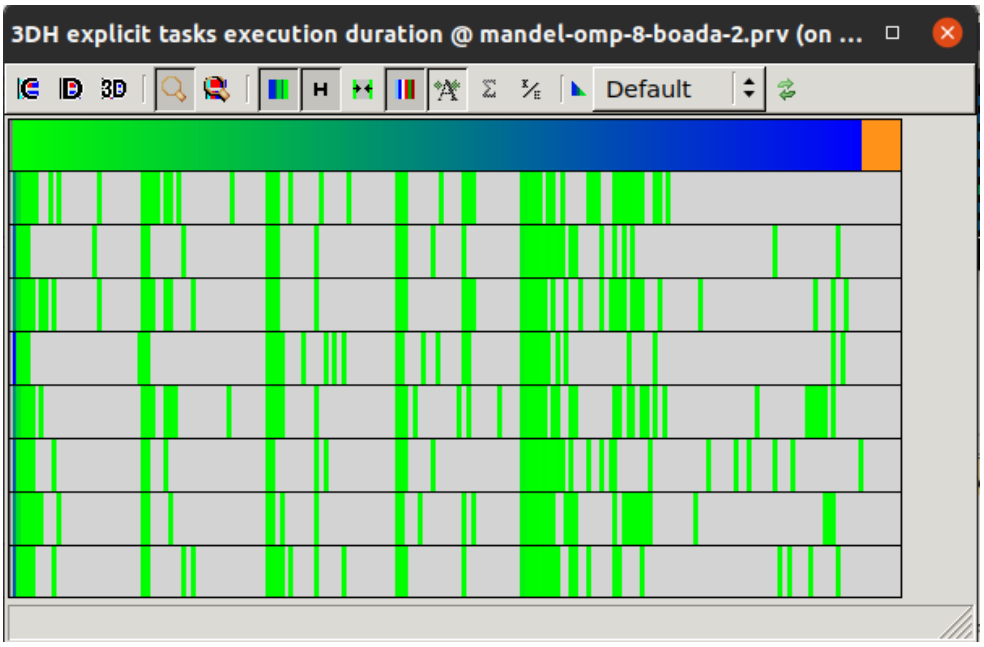

Figure 3.2.9 Every explicit task execution duration per thread of point strategy with taskloop

Let's now apply a new change on our code, as we saw, there were some synchronisation barriers that were not necessary and we tried to remove them. The *nogroup* clause will avoid the creation of implicit *taskgroup* regions and then also the synchronisation barriers (see figure 3.2.10 to see how the code has been modified).

```
#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) {
```

Figure 3.2.10 Implementation of point strategy with taskloop with nogroup clause

We have to protect the variable *row* because now the tasks can execute different rows at the same time, knowing that there's no barrier of synchronisation that forces all the tasks to go together.

That synchronisation barriers should be lowering the time execution so we can see an improvement on the following plots of strong scalability (see figure 3.2.11). In this case, the scalability of the program improves by a little applying this clause.

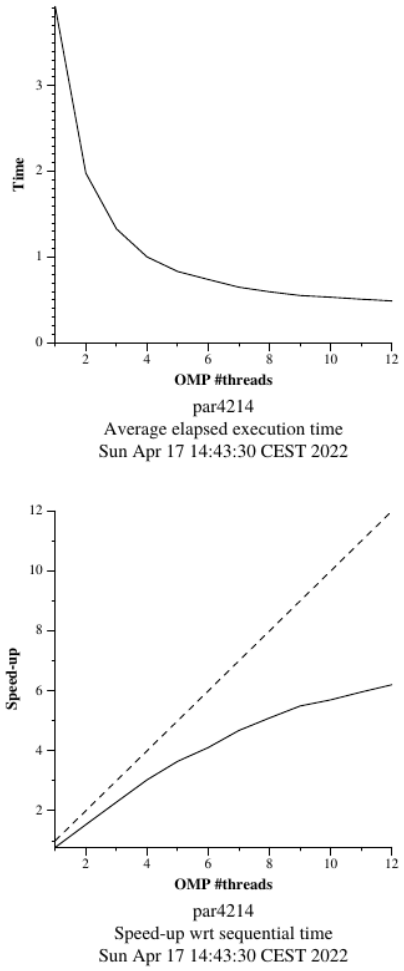

Figure 3.2.11 Plots of time execution and speed-up with point taskloop strategy and nogroup clause

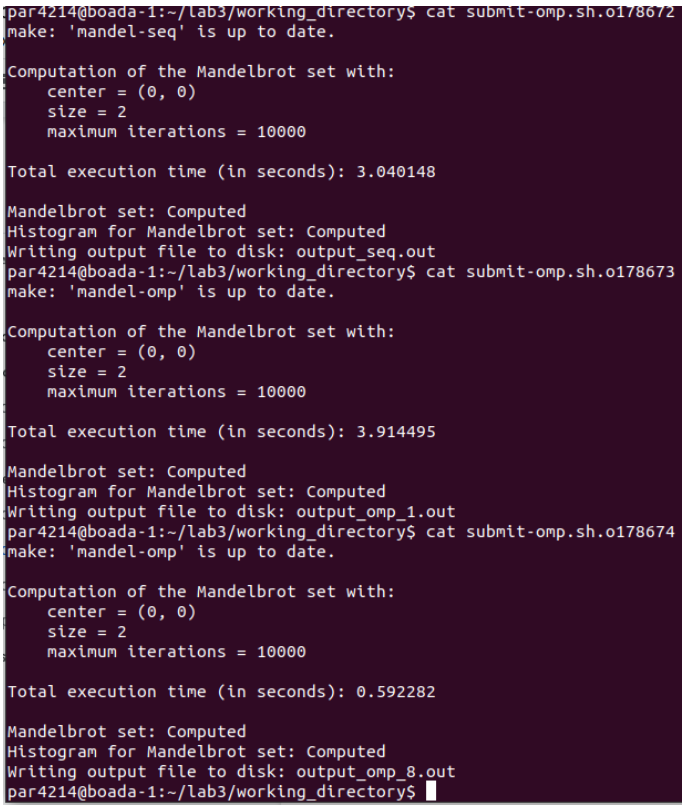

Figure 3.2.12 Time executions of point strategy with taskloop nogroup firstprivate(row) clauses

Point taskloop nogroup firstprivate(row)	Sequential Mandelbrot	Mandelbrot omp 1 thread	Mandelbrot omp 8 threads
Execution time	3.040148 seconds	3.914495 seconds	0,592282 seconds

Table 3.2.2 Time executions of point strategy using taskloop nogroup firstprivate(row)

In this case we can appreciate that the time of the overhead has increased in relation to the previous execution with the implicit barrier. However, as the scalability has improved by little, the time execution reached with 8 threads it's similar to the execution with taskloop and implicit barrier and better than with option task.

We can also see on figure 3.2.13 that the tasks are well balanced among threads.

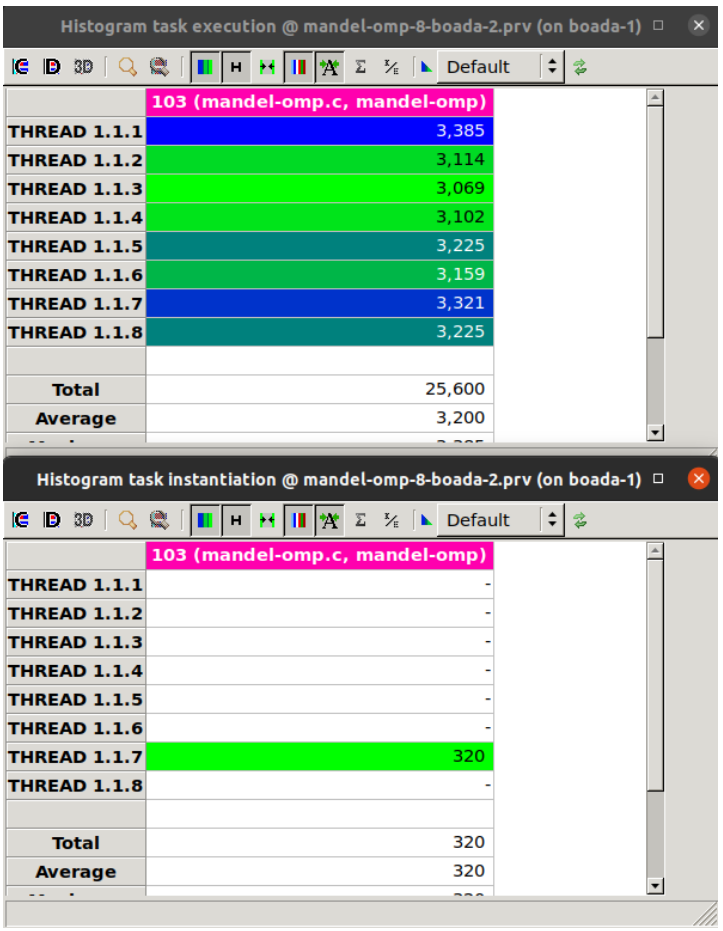

Figure 3.2.13 Every explicit task execution duration and number of tasks created per thread of point strategy with taskloop and nogroup clause

A way to see if there is taskwait or taskgroup barrier synchronisation is opening *Paraver* tool with clause *#pragma omp taskloop* in our code. Then, as we see on figure 3.2.14, there is no event on the trace if we try to show the taskwait construct, otherwise, if we do the same with the taskgroup clause, we can see that the thread that is creating tasks (in that case the thread 8) shows some events of taskgroup in the trace.

Figure 3.2.14 Results of taskwait construct and taskgroup construct on taskloop clause in code

3.3 Row strategy implementation

Next, we have studied the row strategy implementation and its times of execution for each of the options. Firstly, as for the previous implementations, we have modified the code to generate the program. See the figure below.

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

Figure 3.3.1 Code for row strategy implementation with taskloop

```
par4214@boada-1:~/lab3/working_directory$ cat submit-omp.sh.o178678
make: 'mandel-seq' is up to date.

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

Total execution time (in seconds): 3.041635

Mandelbrot set: Computed
Histogram for Mandelbrot set: Computed
Writing output file to disk: output_seq.out
par4214@boada-1:~/lab3/working_directory$ cat submit-omp.sh.o178679
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): 3.914109

Mandelbrot set: Computed
Histogram for Mandelbrot set: Computed
Writing output file to disk: output_omp_1.out
par4214@boada-1:~/lab3/working_directory$ cat submit-omp.sh.o178680
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): 0.555867

Mandelbrot set: Computed
Histogram for Mandelbrot set: Computed
Writing output file to disk: output_omp_8.out
par4214@boada-1:~/lab3/working_directory$
```

Figure 3.3.2 Time executions of row strategy

Row strategy	Sequential Mandelbrot	Mandelbrot omp 1 thread	Mandelbrot omp 8 threads
Execution time	3.041635 seconds	3.914109 seconds	0.555867 seconds

Table 3.3.1 Time executions of row strategy

After this, and as we have done previously, we checked the modifications on the code were correct with the cmp command.

```
par4214@boada-1:~/lab3/working_directory$ cmp output_seq.out output_omp_1.out
par4214@boada-1:~/lab3/working_directory$ cmp output_seq.out output_omp_8.out
par4214@boada-1:~/lab3/working_directory$
```

Figure 3.3.2 Cmp command for row strategy output

Finally, we executed the and studied the strong scalability plots that we got with *./submit-strong.sh* script (see figure 3.3.3).

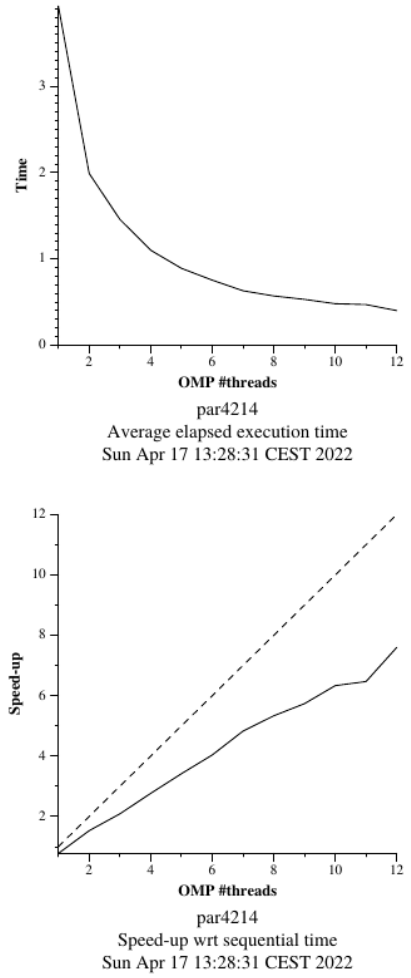

Figure 3.3.3 Plots of time execution and speed-up with row strategy

As clearly seen on figure above, the average elapsed time is around 4 seconds initially, higher than previous implementations with point strategy, but as the number of threads is increased, the time reached is better than previous executions with point implementation and task option, so in fact, it seems to behave similar to the option of taskloop and without implicit barrier. With 6 threads the program reaches the same time as point executions and with 8 threads seems to become better. Until 12 threads as shown on the graph, the time seems to keep improving, so we can't say that a stable point is reached but we can guess that this will happen at a point more or less closer to 12 threads as the marginal decrease on time is lowering.
Finally, if we take a look into the second plot, we can see that in this case the scalability of the program has improved by far. Even though it is not linear (it would mean that it is perfectly parallelised), the speed-up has become much better than previous implementations, and in fact, this is one of the desired characteristics that one would like their program to achieve.

After this first study with plots, we moved on the *Paraver* analysis and found that with row strategy implementation the number of tasks created have decreased but their grainsize is higher (more work for one task). As seen on Figure 3.3.4, there is one thread creating tasks (number 5 in this case), and the others were executing them. We can see that the number of tasks between threads are well balanced unless the one that is creating them (this is a behaviour also shared with the previous implementations).

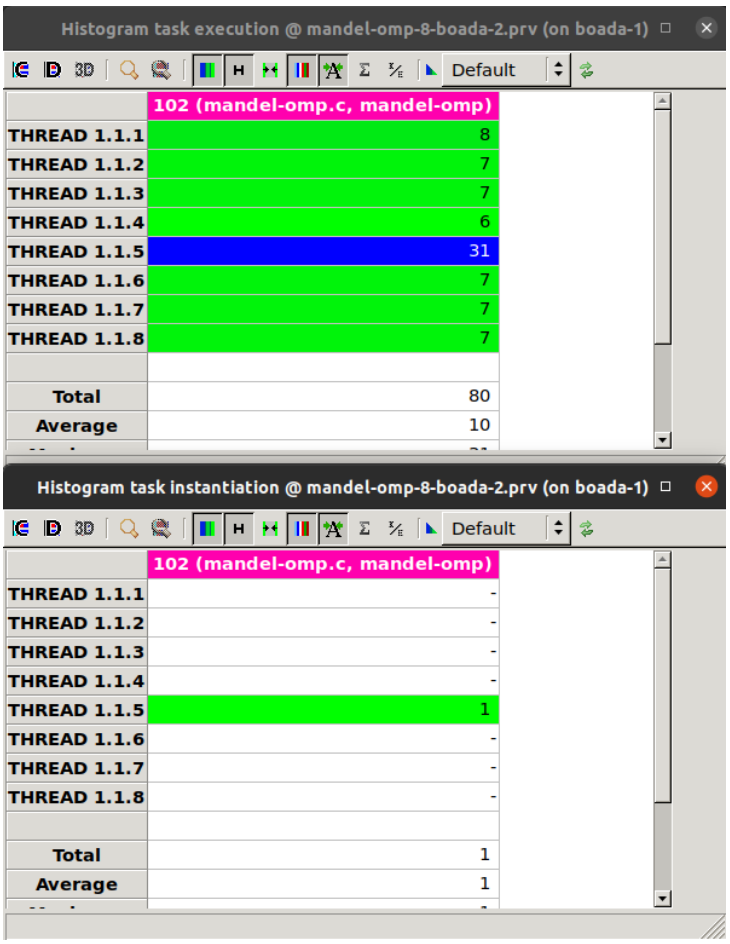

Figure 3.3.4 Histograms of tasks created and time execution per thread in row strategy

We decided to do the same analysis but taking into account time instead of tasks. In this case the time seems to be well balanced among all threads (also with the one creating tasks). In the previous histogram thread 5 seems to do a higher number of tasks but looking into the time we can see that these tasks required less time to be executed (in fact, it is the one with less task execution in time).

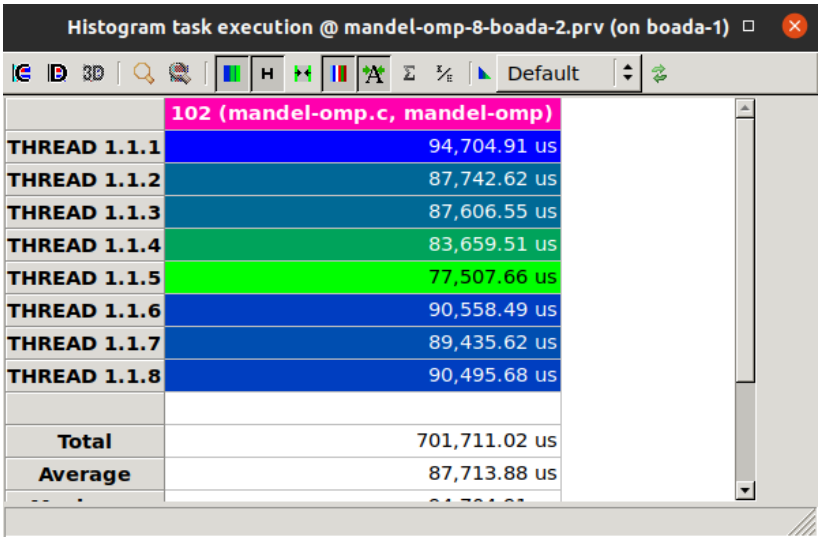

Figure 3.3.5 Histograms of tasks created and time execution per thread in row strategy

The next step we thought was necessary to understand the execution of this implementation was to

generate the new window diagram. As seen on Figure 3.3.6, the majority of time that threads spent time is doing useful work (blue colour) and comparing this diagram to the one seen on point strategy implementation, the synchronisation proportion has decreased considerably (red colour). This is also a good improvement as we don't want to increase the number of threads if that means a higher cost in terms of overhead.

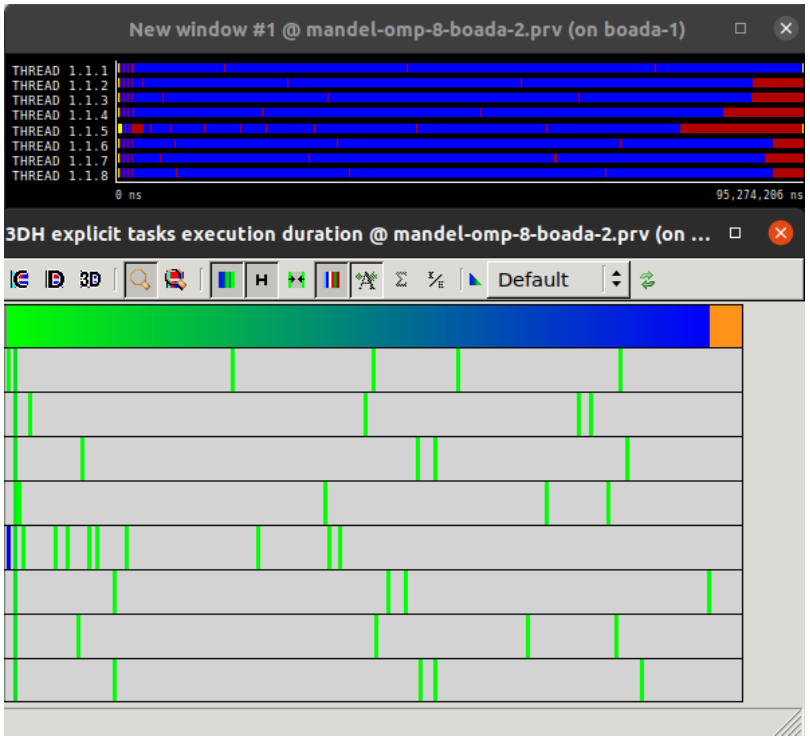

Figure 3.3.6 Histogram and trace of thread tasks executions and execution time of threads

Finally looking into the second histogram of Figure 3.3.6, we can see that apparently there is any implicit barrier applied between tasks, as they have been executed along the timeline indistinctly.

3.4 Optional

In this optional part, we are going to explore the behaviour of both implementations point and row strategy when changing the granularities of tasks.
For this, we are going to use the script provided named submit-numtasks-omp.sh and the implementations chosen will be the ones using taskloop directive as this clause is the one letting the user to change the parameters of granularities.
After exploring the code we find out the variable user_param that allows us to change the number of tasks created by the program. This variable was initialised with the value of 1, but could be changed at the main part. See Figure 3.4.1.

```
else if (strcmp(argv[i], "-u")==0) {
                    user_param = atof(argv[++i]);
}
```

Figure 3.4.1 Part of the code to change user_param

On Figure 3.4.2 we can see the changes applied on the code in order to modify the number of tasks to be created with the value of user_param.

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

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

Figure 3.4.2 Code modified with the value of user_param as num_tasks.
On the left the row strategy and on the right the point strategy one

After these modifications, we submitted to execute the command line: *#sbatch ./submit-numtasks-omp.sh mandel-omp,* we saw that the script change automatically the number of tasks for every execution setting the user_param variable with the desired values.

```
foreach ntasks ( $num_tasks )
        echo $ntasks >> $out
        ./$PROG -h -i $numits -u $ntasks >> $out
        set result = `cat $out | tail -n 4  | grep "Total execution time"| cut -d':' -f 2`
        echo $i >> ./elapsed.txt
        echo $result >> ./elapsed.txt
        set i = `echo $i + 1 | bc -l`
end
```

$num_task = { 1, 2, 4, 8, ... , 512, 800 }

execution of the program

Figure 3.4.2 Part of the script submit-omp-numtasks.sh where is executed the program mandel-omp with different num_tasks

Finally we can see the plots generated here below. In the first one, for row strategy, we can see how time decreased by increasing the number of tasks and reached a ceiling when arriving at the number of 128 tasks. We can see this decrease is higher when changing from 1 to 2 tasks and becomes lower when keeping increasing the number. Between 4 and 32 tasks, the decrease seems to be linear but at the point of 32 tasks the marginal decrease is lowering until reaching a stable point. We reach a number of tasks where the grain size of the tasks doesn't really matter at all, or just that the grain size of the task reaches at 1 instruction. As we know, the mandelbrot set that is being computed is 320x320 pixels, so the number of rows of the image is 320. If the number of tasks are 128, then the grain size of the task are 2 rows, so the difference between executing with a gran size of 2, or 1 (256 number of tasks or higher) it's minimum, and inclusively worse for the overhead of creating a task (but also that is a minimum change).

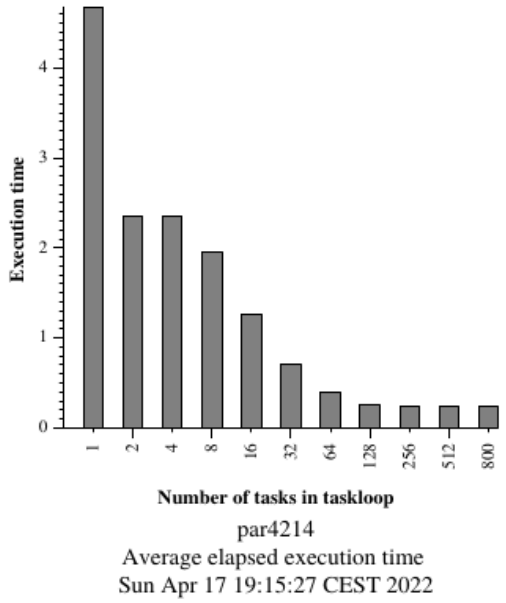

Figure 3.4.2 Plot row strategy average elapsed execution time

Finally, when we execute the script with the point strategy we clearly see the same behaviour as the row strategy, unless that now the time execution ceiling is on 64 number of tasks and the initial decrease is more gradual (between 1, 2 and 4 number of tasks). Now that ceiling is a little bit less stable as the row one, we can see that sometimes the overhead can worse the time execution by a little.

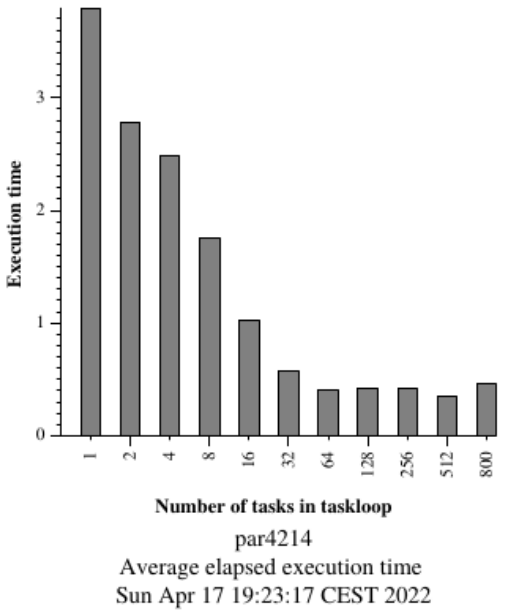

Figure 3.4.3 Plot point strategy average elapsed execution time

4

Conclusions

In this final section we are going to explain the main conclusions we have obtained after doing these two laboratory sessions. Firstly we will talk about the task decomposition strategies, which one is the best, the difference between them and finally about the conclusions of the impact that granularity has on the program.

Initially, we have been studying the different strategies implementations to parallelise the Mandelbrot set code. With the help of *Tareador* tool we have seen how the program was perfectly parallelizable, this is the reason why we have analysed both strategies: point and row implementations. In addition, we saw that the program was impossible to parallelise with -d and -h options (computing the output diagram and the histogram) because of the creation of dependencies.

After this first study, we move on the analysis of two different ways to implement the point strategy. Both implementations allow us to parallelise the code creating a task for every point of the set, which means a high number of tasks with a little granularity. The first one implemented with task OpenMP clause seemed to be worse in time execution and overheads than the second one with the taskloop directive. Moreover, both of them didn't show a good performance on strong scalability plots but again the second had better results.

Then in the same section, finally we studied the row strategy implementation. This implementation has a lower number of tasks but with higher granularity. Although the time was worse with 1 thread, the increase in threads made the program perform better. Also, the scalability in this case looked better than point strategy with taskloop.

To conclude that study, we optionally saw how the best strategies behaved by modifying the number of tasks per thread, in other words, which was the time execution with different granularities. The results of the script *submit-omp-numtasks.sh* of point strategy with taskloop and row strategy were similar. Both strategies reached a ceiling when the granularity was little (high number of tasks) but the point strategy one reaches it faster (with less number of tasks). Also, looking at the time executions, in average row strategy implementation stabilises at a lower time execution.

We can say that in these two sessions we have seen the importance of the implementation of parallelisation, how to treat the implicit barriers, how to analyse in different terms the programs (tasks and time), how to see the importance of the overheads and finally how to analyse the potential scalability of the program, which is a desired characteristic for a programmer.