PAR Laboratory Assignment 3

Index

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

In this laboratory session, we learnt about the tasking model in OpenMP in order to express iterative task decompositions. All the programs that we work in this assignment compute the *Mandelbrot* set, which is the representation of a set of points, in the complex domain, whose boundary generates the following two dimensional fractal shape that is very recognizable:

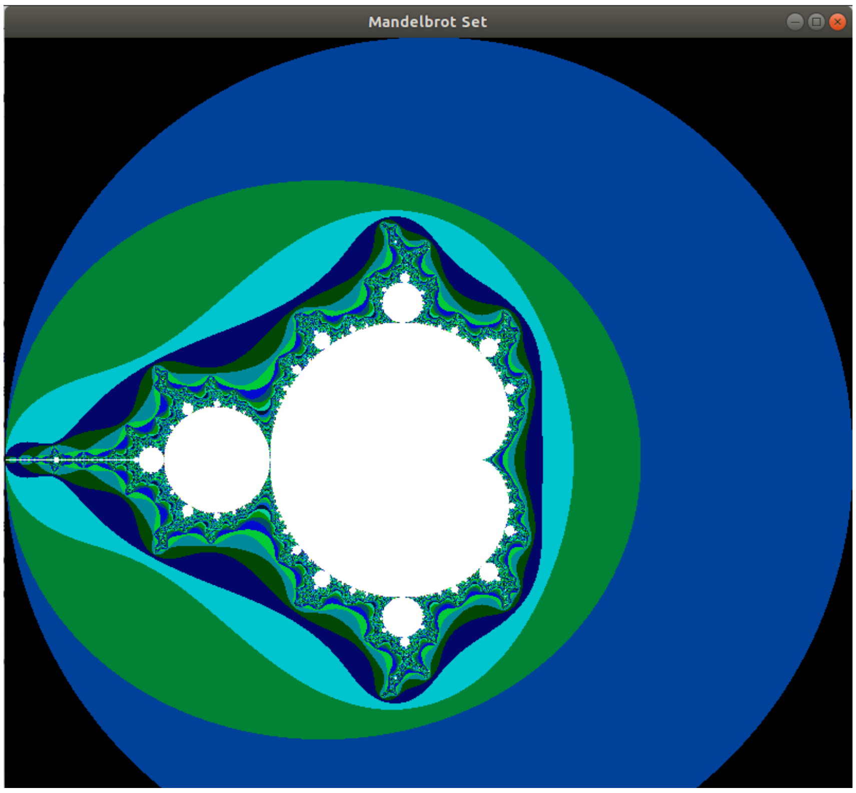


fig. 1: *Mandelbrot* set displayed executing *mandel-seq.c* using the **-d** option

Before we dived into the exercises we tested the functionality of the *mandel-seq.c* file and the different output options that the program can be executed with: **-h**, where the histogram for the values in the *Mandelbrot* set is computed along with the program; **-d**, where the *Mandelbrot* set is displayed (see fig.1 for an example); **-o**, where the values of the set and/or histogram are written to the disk, in order to compare the values obtained with the reference output (this option is usually implicit inside the execution of most of the files that are given).

Task decomposition analysis and granularity analysis with Tareador

In this section we used *Tareador* to see the two proposed task decomposition strategies *Row* and *Point*, and we studied their characteristics by looking at the task dependence graphs computed. For this section we used the *mandel-tar.c* file for the code and the *run-tareador.sh* sript to execute it with *Tareador*.

Row strategy

We started by analyzing the potential parallelism for the *Row* strategy. We completed the code by creating tasks inside the *row* loop writing `tareador_task_start(“ROW”)` and `tareador_task_end(“ROW”)`, just as the following code:

```
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_task_start("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_task_end("ROW");
    }
```

When compiling and executing the file (without using options) with *Tareador* we obtained the following graph:

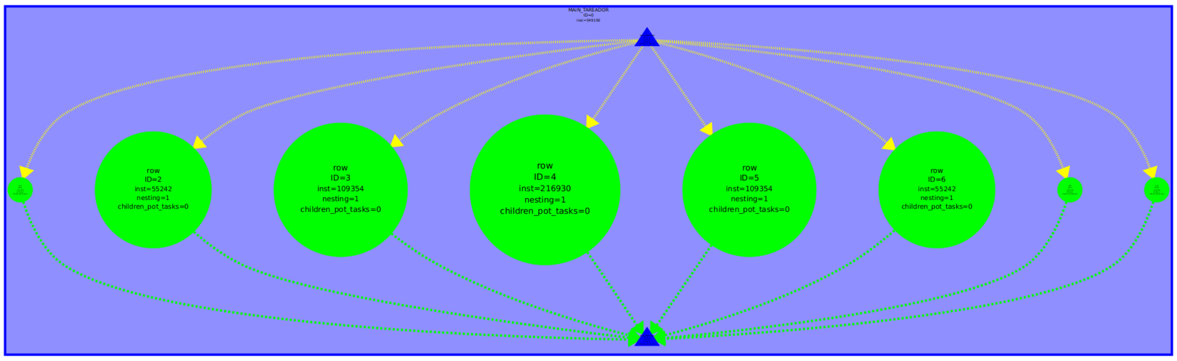


fig. 2: Row task graph (no options used)

One characteristic that we can see after generating the graph, is that there are tasks (green nodes) that are way bigger than others. Obviously, for parallelism purposes this is not great. The other characteristic that we found is that between tasks there are no dependencies, they all have a dependency from the main task (shown in a blue triangle on fig. 2).

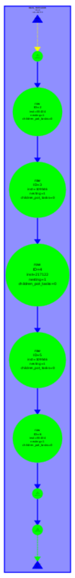


fig. 3: Row task graph (option **-d** used)

Next, we executed *mandel-tar* again but, this time, using the **-d** option, where the *Mandelbrot* set is displayed. When executing the file a small window is opened displaying a small *Mandelbrot* set, and on *Tareador* we see the graph that we have on the right (see fig. 3).
Again, we see that there are tasks a lot bigger than others, but this time, each task has a dependency with a previous one, forming a serialization of tasks.
The part of the code that is making a big difference in this execution, we find it inside the *if(output2display)* clause, because we are using the **-d** option. Inside this *if* condition we find the functions *XSetForeground* and *XDrawPoint*, which set a color of a pixel and display it on the screen.
This area of the code needs to be protected in order to prevent different threads accessing the same variable that stores the color, which is accessed multiple times. If not, that would create a data race, so a good way to avoid it would be using *#pragma omp critical* right before the functions *XSetForeground* and *XDrawPoint*. This is something that we are going to apply to our code in the next sections.

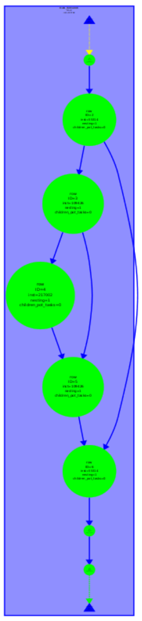


fig. 4: Row task graph (option **-h** used)

Finally, we execute the code using the **-h** option. The resulting graph that we have on *Tareador* is the one that we have on the left (see fig. 4). Like the previous one, we have a serialization of tasks and some of them are way bigger than others. Although there are tasks that have more than one dependency.
The part of the code that makes a difference in this execution is the *if(output2histogram)* condition, which is accessed when executing with **-h** option, where the histogram is computed along with the execution of the set.
In this part of the code, threads access to the *k-1* position of *histogram*, the code turns to sequential because they can't access elements that don't exist in that position. On top of that, just like in the previous situation, a datarace occurs too because the threads will access a position in the memory all at once. In order to protect it, we create a region that only one thread can access it, so we will use *#pragma omp atomic*. We use it over *critical* because, apart from being much faster, it ensures the serialization of a particular operation, in this case *histogram[k-1]++*. Again, this is a strategy that we are going to use for the next sections.

Point strategy

Now that we have studied the *Row* strategy, we analyse the *Point* strategy. In order to see the the potential parallelism for this strategy we completed the initial *mandel-tar.c* code the following way:

```
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_task_start("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_task_end("COL");
        }
    }
```

Since we created the tasks in the innermost loop, when executing *mandel-tar* with *Tareador*, we see a lot more tasks than the *Row* strategy:

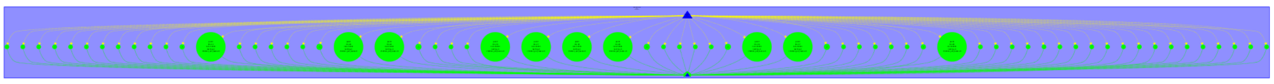


fig. 5: *Point* task graph (no option used)

We have the same characteristics as the previous strategy: no dependencies between tasks and some tasks are bigger than others.

Then, we execute the file using the **-d** option, and the resulting graph is the following:

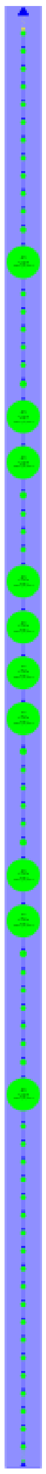


fig. 6: *Point* task graph (option **-d** used)

This long task dependence graph that we see on the left (see fig. 6), has the same characteristics as the one we did in the previous strategy, but obviously has a lot more tasks. We still see a size imbalance between tasks and just one dependency with the previous task (serialization).

Finally, if we execute with the **-h** option, the graph that we have on the right (see fig. 7) is generated on *Tareador*. This graph is quite different from the ones we have seen before. Although it seems that the tasks are executing in a parallel formation, it is still considered to be sequential because the bigger tasks still are being executed in a serialization.

In conclusion, having seen an analysis of the *Row* and *Point* strategies, we see that in both cases the part of the code that makes a difference in the execution is the same, either if we execute with the **-d** or the **-h** option. If we study the graphs we can draw the same conclusions in terms of the main characteristics. However, in the *Point* strategy there are a lot more tasks because the creation is in the innermost loop.
Out of both strategies, the most appropriate one in this case is the *Row* strategy. The reason is because there are tasks way bigger than others, and in every execution they are executed sequentially so, since the *Row* strategy has a lot less tasks to compute, this method would be the most appropriate in this case. Furthermore, for the same reason, the *Point* strategy has a problem with overheads when creating the tasks every iteration in both loops.

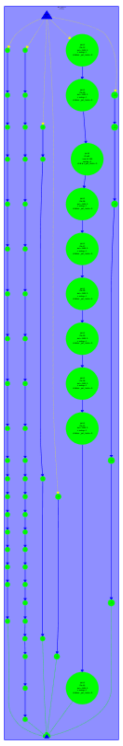
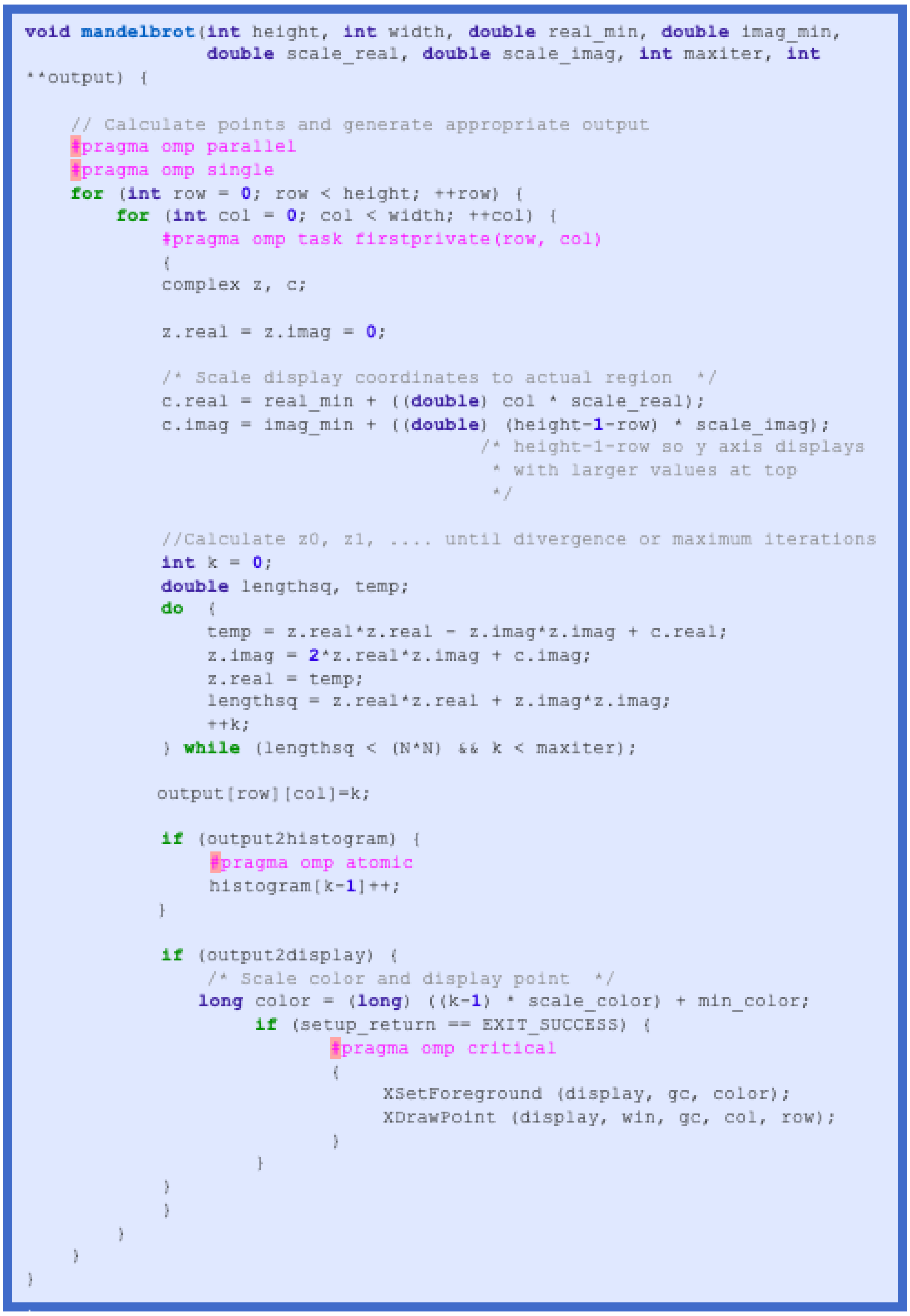


fig. 7: *Point* task graph (option **-h** used)

In these next sections we will see the different ways to express the iterative task decomposition strategies for the Mandelbrot computation program. We will study the scalability and other characteristics of our code.

Point strategy implementation using task

In this section we implement the *Point* strategy in the given code *mandel-omp.c* using `task`. First of all, we open the code and we make sure that it is in fact using the *Point* parallelization. Then, we insert the OpenMP directives *atomic* and *critical* in order to protect the dependencies we detected in the previous section using *Tareador*:



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

In the previous code we show the function `mandelbrot` where we make all the changes we need. Just as we justified in section 2.2, we add *#pragma omp atomic* inside the histogram clause and *#pragma omp critical* for the display clause.
Next, we compile and interactively execute our *mandel-omp* file, in order to check if the parallelization is correct. We execute it using the **-d** option and with 1 thread and 10000 iterations (`OMP NUM THREADS=1 ./mandel-omp -d -h -i 10000`) and the following set is displayed:

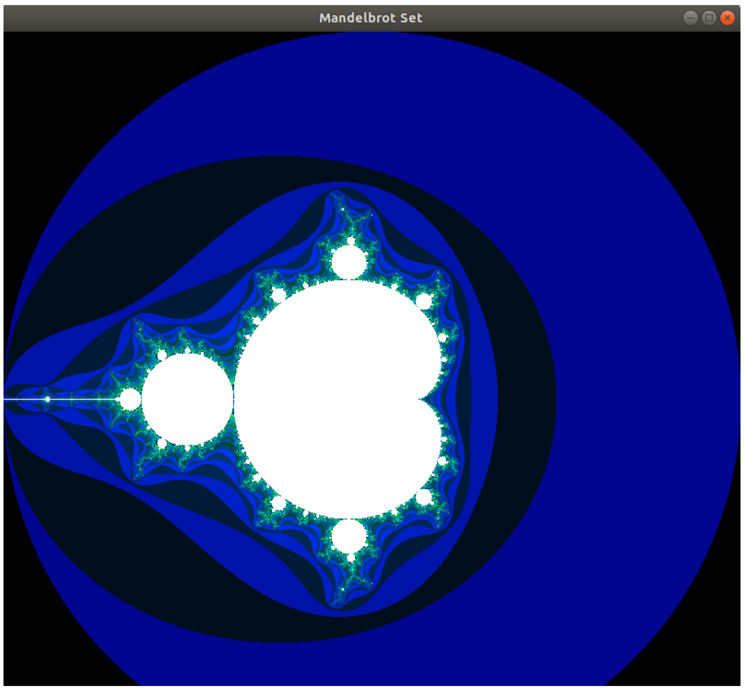


fig. 8: Mandelbrot set generated with *mandel-omp*

As we can see the image is generated correctly (see fig. 8). We see differences with the sequential version if we look at the colors, but the generated set is the same.
Then, we interactively execute the binary again but with 2 threads and 10000 iterations (`OMP NUM THREADS=1 ./mandel-omp -d -h -i 10000`) and we clearly see that the set is exactly the same as the one executed with 1 thread (see fig. 8), so it is generated correctly too.

The next thing we want to measure is the time reduction we had when we executed in parallel, in comparison to the sequential execution. In order to do that, we execute *mandel-seq* and *mandel-omp* using the *submit-omp.sh* script with 1 and 8 threads:

- *mandel-seq*:
  ```
  3.04user  0.00system  0:03.20elapsed  94%CPU  (0avgtext+0avgdata
  5364maxresident)k

  120inputs+5088outputs (1major+770minor)pagefaults 0swaps
  ```

  Total execution time: 3.042768 seconds

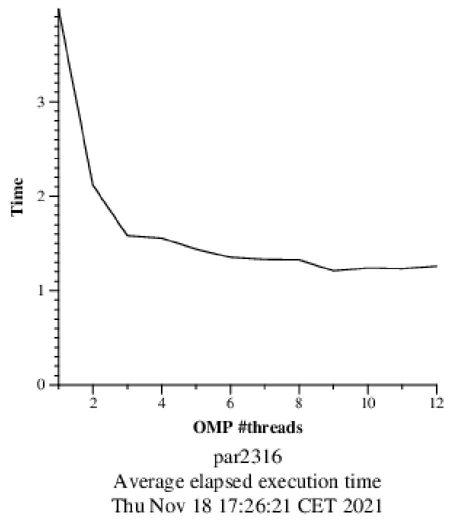
- *mandel-omp* 1 thread:
  ```
  3.97user  0.00system  0:04.09elapsed  97%CPU  (0avgtext+0avgdata
  7116maxresident)k

  144inputs+5088outputs (1major+1024minor)pagefaults 0swaps
  ```

Total execution time: 3.971450 seconds

- *mandel-omp* 8 threads:
  ```
  10.99user 0.22system 0:01.51elapsed 743%CPU (0avgtext+0avgdata 8380maxresident)k

  0inputs+5088outputs (0major+2795minor)pagefaults 0swaps
  ```

  Total execution time: 1.390128 seconds

We can clearly see, especially in the *mandel-omp* execution with 8 threads, that there is an improvement in the total execution time (it takes less time to execute than the sequential code). To better show the results of the speed-up and execution time, we execute it using *submit-omp-strong.sh*, so we can obtain the plots for the 1-12 processor range:



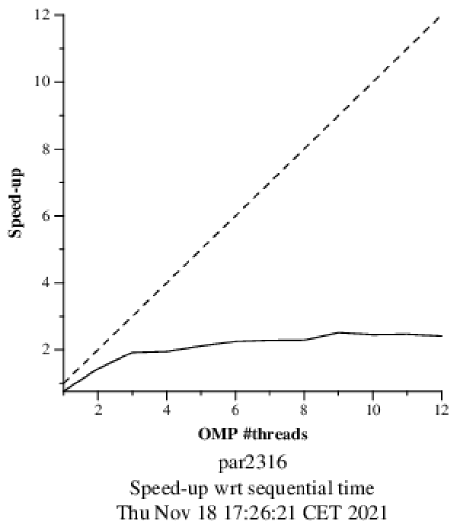
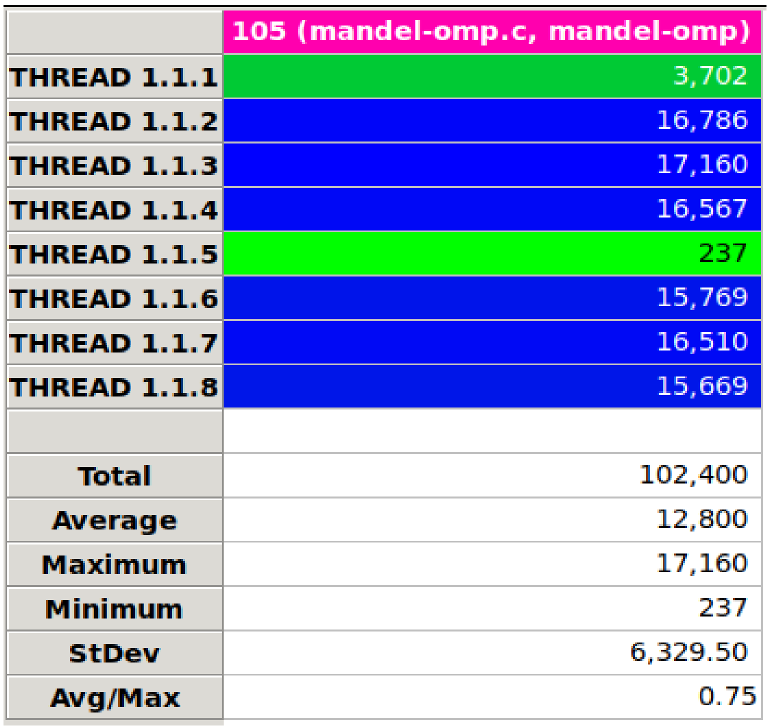


fig. 9: Time vs Number of threads and scalability plot

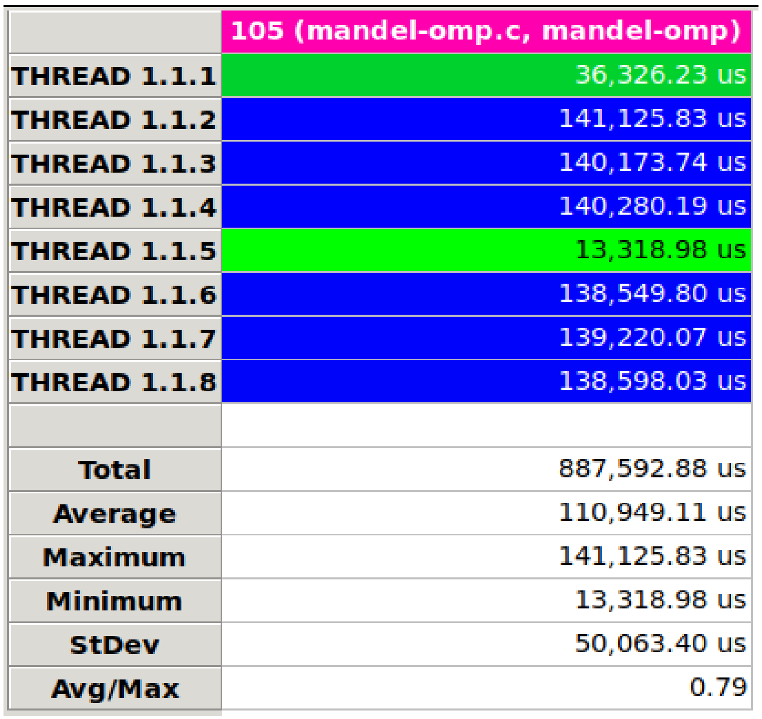
On the Time vs Number of threads plot, we see that time decreases when the number of threads is increased. But when it gets to 5-6 threads, that time decreases very slowly to the point where we can consider that it stays constant. On the other hand, in the scalability plot we can see that the scalability is far from being considered optimal. The speedup does not increase linearly along with the increase of threads nor even close to that, so the scalability is not considered appropriate in this case.

The next step is to find out how the execution is working. In order to do that we execute with 8 threads the *mandel-omp* file with the *submit-extrae.sh* script. What this execution does is to generate many files, one of them, the *.prv* file, is the one we open with *Paraver*. Once we have it, we can analyze the different traces that the *Paraver* tool offers us. We can visualise the *implicit* and *explicit tasks* that are created unfolding the *Hints* menu and selecting the specific option, respectfully. This will show a profile of the tasks generated and executed. The next tables are a configuration file showing a profile of the explicit tasks:



	105 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	3,702
THREAD 1.1.2	16,786
THREAD 1.1.3	17,160
THREAD 1.1.4	16,567
THREAD 1.1.5	237
THREAD 1.1.6	15,769
THREAD 1.1.7	16,510
THREAD 1.1.8	15,669
Total	102,400
Average	12,800
Maximum	17,160
Minimum	237
StDev	6,329.50
Avg/Max	0.75

fig. 10: Profile of the tasks generated and executed. Shows number of tasks per thread



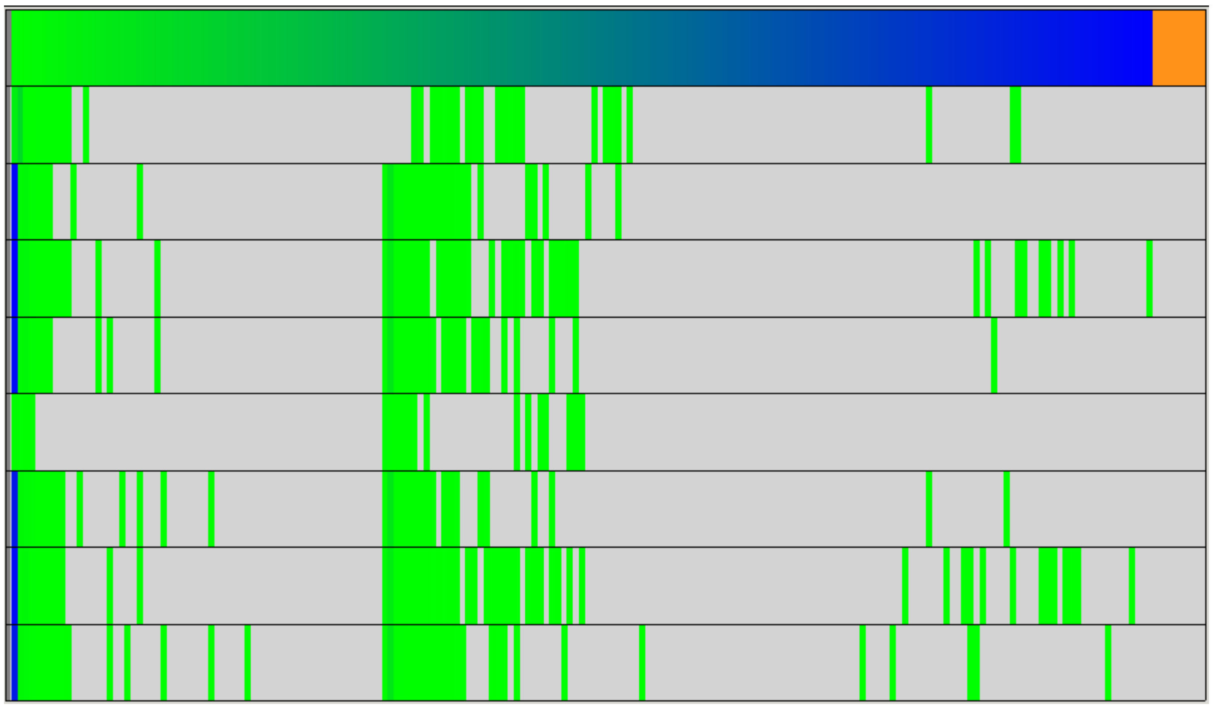
	105 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	36,326.23 us
THREAD 1.1.2	141,125.83 us
THREAD 1.1.3	140,173.74 us
THREAD 1.1.4	140,280.19 us
THREAD 1.1.5	13,318.98 us
THREAD 1.1.6	138,549.80 us
THREAD 1.1.7	139,220.07 us
THREAD 1.1.8	138,598.03 us
Total	887,592.88 us
Average	110,949.11 us
Maximum	141,125.83 us
Minimum	13,318.98 us
StDev	50,063.40 us
Avg/Max	0.79

fig. 11: Profile of the tasks generated and executed. Shows execution time per thread

The previous tables show the number of tasks per thread (see fig. 10) and the execution time per thread (see fig. 11). What we can conclude from these two tables is that excluding the first and the fifth thread, we see that the rest of the threads are balanced. In case of thread number 5 we observe that it executes fewer tasks because it is the one that creates them, and in the case of thread number 1 we can see that although it creates a lot more tasks, its total execution time is less. Basically, these two threads are the ones creating the tasks.
The total number of tasks created is 102.400 and we can see that they are not well balanced between the threads. The first and the fifth thread execute way less tasks and they have less execution time than the rest. But, out of the two terms the number of tasks is the one that is relevant because that is the factor that determines what thread/s is/are the task creators.

The following histogram shows the duration of the explicit tasks:

fig. 12: Histogram - Duration explicit tasks

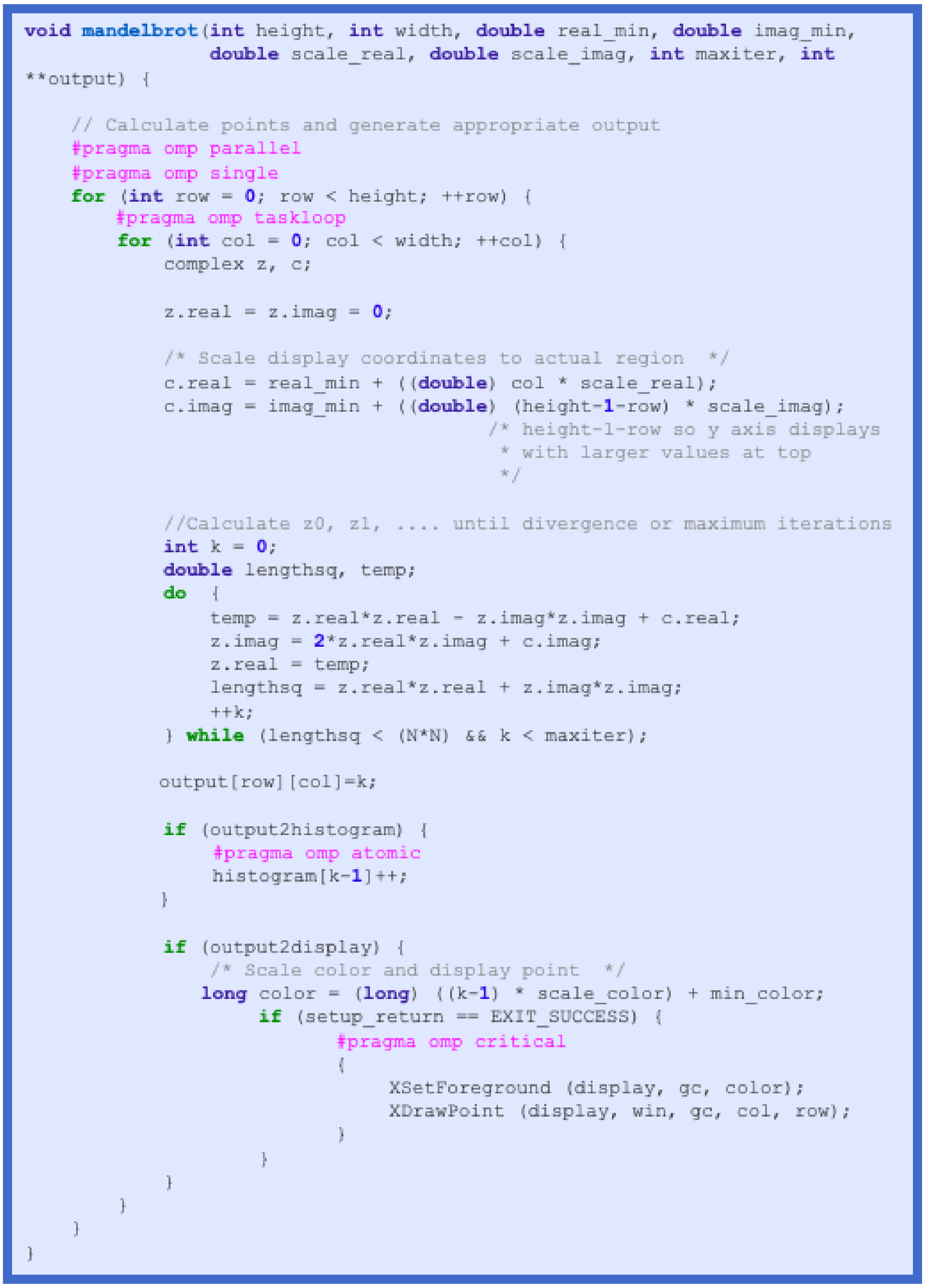


The previous image (see fig. 12) shows the duration of each task executed per thread. We can see that these tasks are created by what seems to be cycles. The reason for that is because a taskloop is created in each iteration of the *row* loop and in that iteration, as many tasks as *col* iterations are created at the same time (that creates these “green barriers” that are shown in fig. 12).

Finally, we can look at the effect of the overheads per thread just by looking at the trace diagram that is provided when generating the file in *Paraver*. We can see that the granularity of the tasks is not appropriate because there is a big imbalance between tasks. That can be changed using other OpenMP directives, just like we did in section 2.2.

Point strategy with granularity control using taskloop

Next we take the same code as the previous section and we use the `taskloop` construct instead. That clause generates tasks out of iterations of the loop so it allows better control of the tasks created and their granularity.
To implement it, we add *#pragma omp taskloop* before the *col* loop and inside the *row* loop:



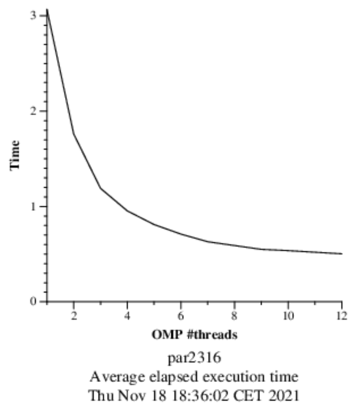
```
void mandelbrot(int height, int width, double real_min, double imag_min,
                double scale_real, double scale_imag, int maxiter, int
**output) {

    // 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);
                           }
                    }
            }
        }
    }
}
```

Once the mandel-omp.c code has been modified, we execute it with a different number of threads:

Threads	Time (s)
1	13.230470
2	3.595168
8	1.675000

fig. 12: mandel-omp with diferent number of threads

To study the scalability of the program, we execute the *submit-strong-omp.sh* with *sbatch* and we pass the *mandel-omp* as a parameter. Once the execution is finished, using gs we obtain the strong scalability graphs:



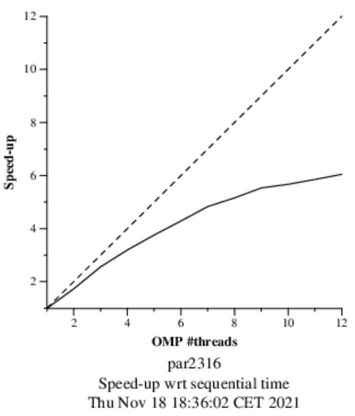


fig. 13: mandel-omp strong-scalability

When executed with a single thread, the overheads that are generated by the massive creation of tasks, make the execution time very long.

By running it with 2 threads, the tasks are spread out and the execution time is reduced considerably, despite the overheads.

The speedup compared to the sequential version improves significantly, up to 8 threads, and then it stabilizes.

The speedup is not what you would expect due to the great influence that overheads have on runtime.

We can see in the graphs, that with respect to the task, the taskloop manages to considerably increase the speed.

Next we trace the execution with 8 processors to better understand the scalability. With *paraver* we generate the graph of the 8 thread execution. In the graph, we observe how much of the time is spent in synchronization (red color). We also notice that thread 5 is responsible for generating the tasks, this is represented by the color yellow.

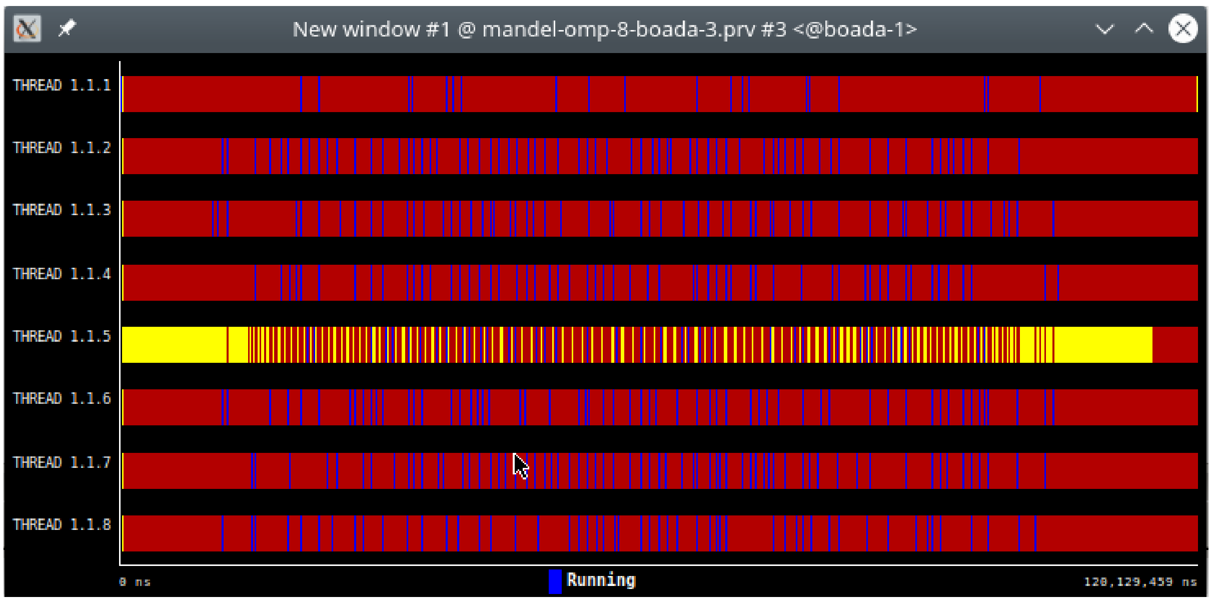
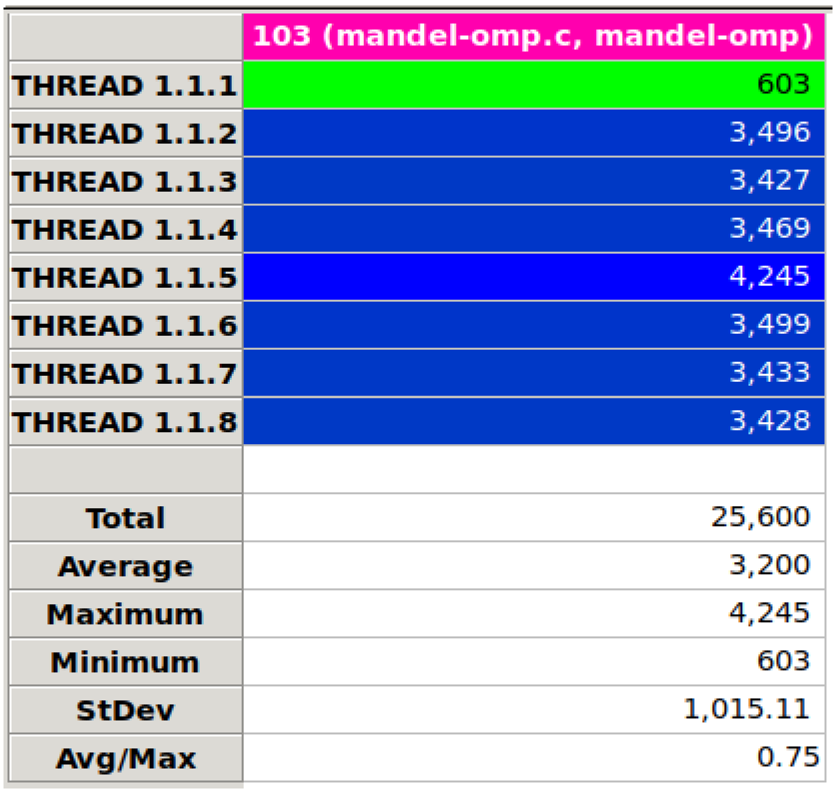


fig. 14: mandel-omp execution graph

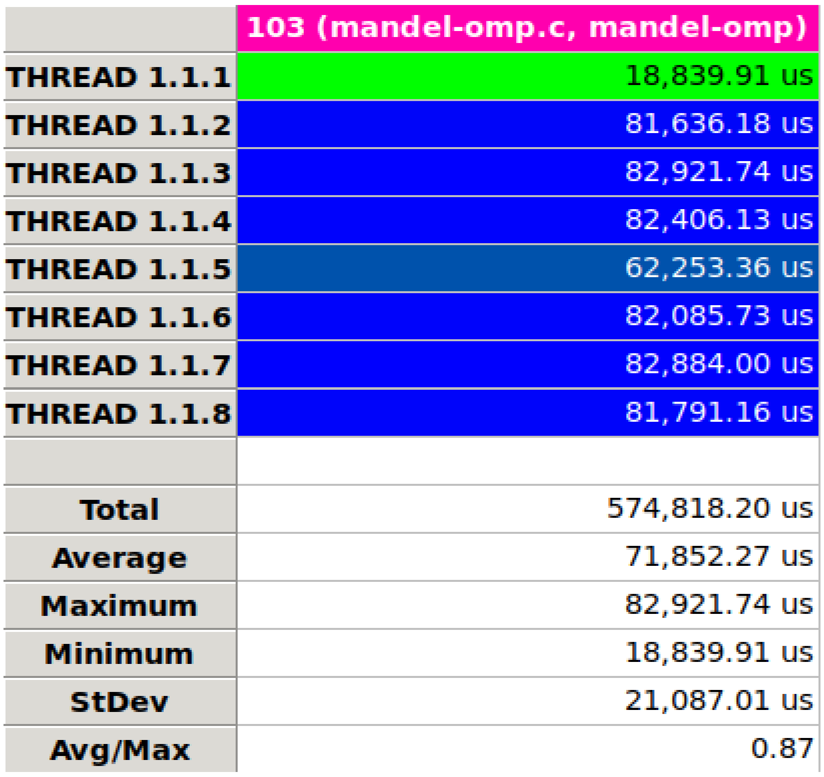


	103 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	603
THREAD 1.1.2	3,496
THREAD 1.1.3	3,427
THREAD 1.1.4	3,469
THREAD 1.1.5	4,245
THREAD 1.1.6	3,499
THREAD 1.1.7	3,433
THREAD 1.1.8	3,428
Total	25,600
Average	3,200
Maximum	4,245
Minimum	603
StDev	1,015.11
Avg/Max	0.75

fig. 15: Profile of the tasks generated and executed. Shows number of tasks per thread

The table in figure 15 shows the number of tasks per thread.

Observing the graphs we can deduce that thread 1 executes fewer tasks, because it spends most of the time in synchronization and therefore its execution time is less than the rest of the threads. Thread 5, as we have already mentioned before, is in charge of creating the tasks, and because creating a task is a fast process compared to its execution, it has more tasks assigned than any other thread. Like thread 1, its execution time is less than the rest.



	103 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	18,839.91 us
THREAD 1.1.2	81,636.18 us
THREAD 1.1.3	82,921.74 us
THREAD 1.1.4	82,406.13 us
THREAD 1.1.5	62,253.36 us
THREAD 1.1.6	82,085.73 us
THREAD 1.1.7	82,884.00 us
THREAD 1.1.8	81,791.16 us
Total	574,818.20 us
Average	71,852.27 us
Maximum	82,921.74 us
Minimum	18,839.91 us
StDev	21,087.01 us
Avg/Max	0.87

fig. 16: Profile of the tasks generated and executed. Shows execution time per thread

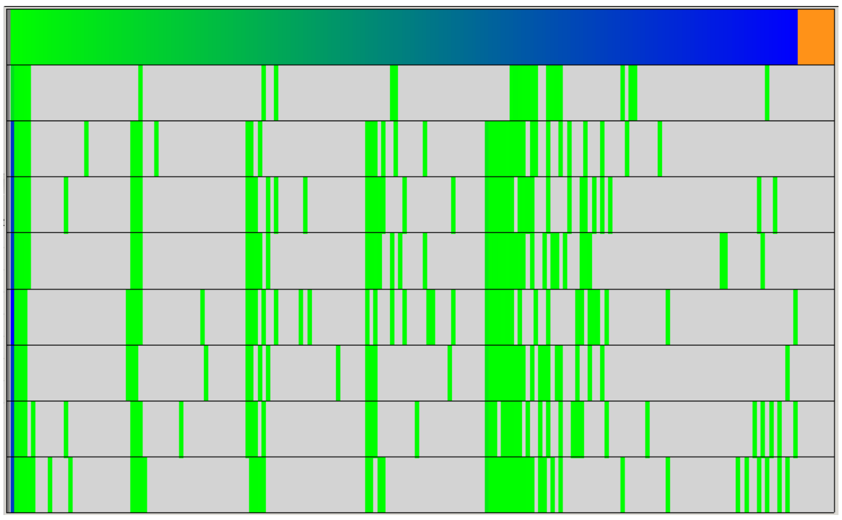


fig. 17: Histogram - Duration explicit tasks

The histogram shows how each execution of the row loop generates burst tasks that are executed in parallel using the available threads (see fig. 17).

On every taskloop Width (number of pixels of the window) tasks are created, the number of threads has nothing to do with the width, since it is defined in NPIXELS.

When trying to use the hints in taskgroup construct and in taskwait construct, the program only allows us to select the hint with the option used by the taskloop. In this case, the *Paraver*'s chosen option has been the taskgroup construct.

The barriers that are introduced so that all the tasks finish before generating the following ones, are not necessary, as long as the -h (output2histogram) option is not activated. In the case of using this option, the program is no longer embarrassingly parallelizable and needs the implicit barriers.

So we modified *the mandel-omp* code and implemented a version by adding the `nogroup` parameter to the omp directive of the taskloop.
In the strong scalability graphs we observe a behavior similar to the previous version of the code. The more the number of threads increases, the faster it moves away from the ideal (see fig. 19).

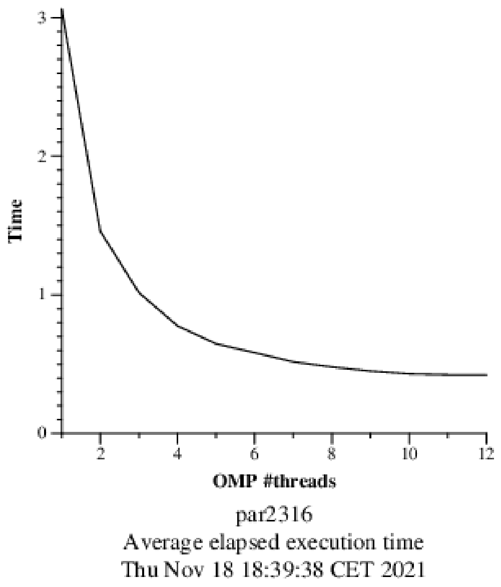


fig. 18: Time vs Number of threads (nogroup clause added)

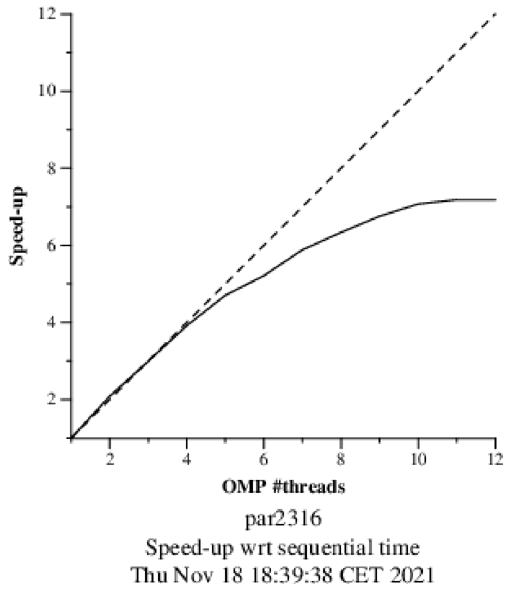
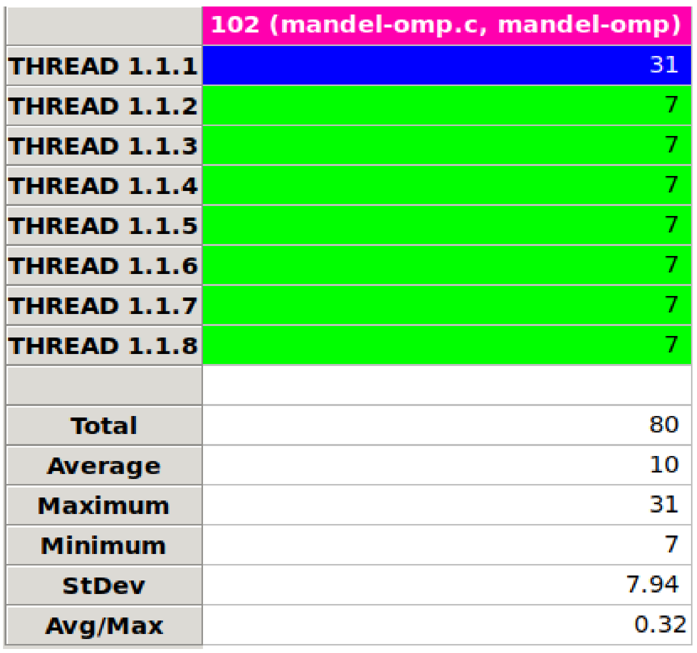


fig. 19: Strong-scalability plot (nogroup clause added)



	102 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	31
THREAD 1.1.2	7
THREAD 1.1.3	7
THREAD 1.1.4	7
THREAD 1.1.5	7
THREAD 1.1.6	7
THREAD 1.1.7	7
THREAD 1.1.8	7
Total	80
Average	10
Maximum	31
Minimum	7
StDev	7.94
Avg/Max	0.32

fig. 20: Profile of the tasks generated and executed. Shows number of tasks per thread

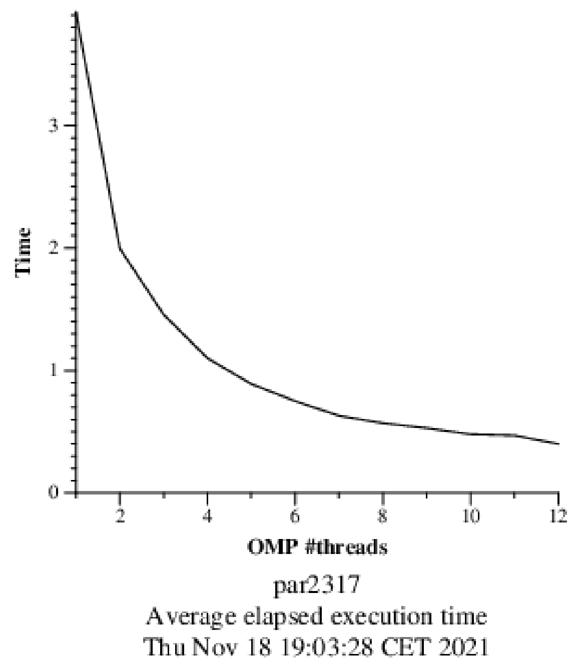
As we can see in the previous table (see fig. 20), using the *Row* strategy has a lot less number of tasks per thread. Obviously, that is because the taskloop construct is outside the *row* loop so the total number of tasks is way less than in the *Point* strategy. In this case, thread number 1 is the main creator of tasks and the rest of the threads all have 7 tasks each, so we can still say that there is an unbalance of tasks.

Row strategy implementation

To carry out this strategy, we move *#pragma omp taskloop* before the loop row.
Once the *mandel-omp.c* code was modified, we verified that it was running correctly.
Then we go on to study its scalability.

This time there is a significant improvement in speedup and runtime in the strong scalability charts. Unlike the Point version, with row strategy we manage to bring the speedup and the reduction of execution time closer to the ideal one.

Finally, we trace the execution and we see in fig. 23 that, while each of the rest of threads execute the same amount of tasks, thread 1 executes most of the work, so there's a work imbalance.



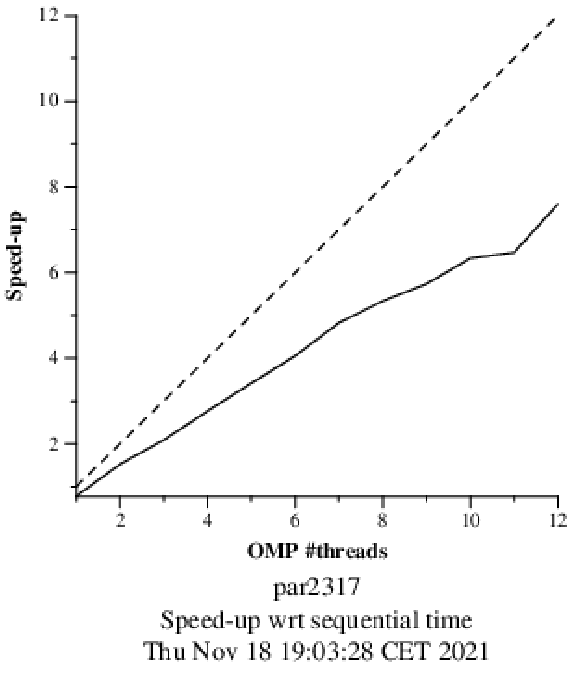


fig. 22: Strong-scalability plots (*Row* strategy)

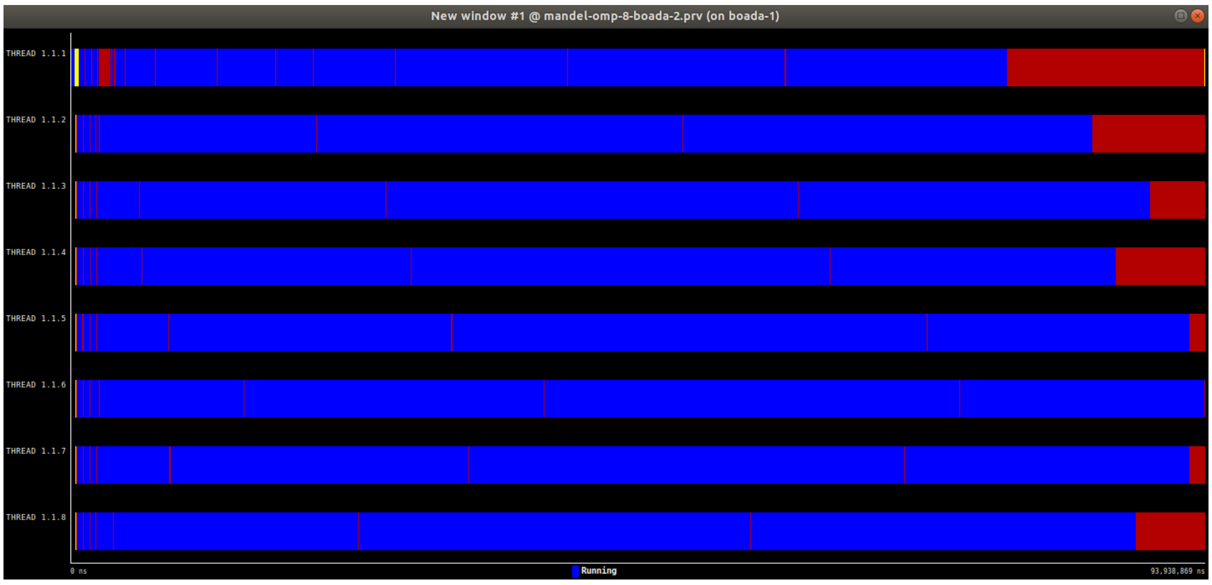


fig. 23: execution graph

Although there is part of the execution time, which is used in the synchronization of the threads, this time does not slow down the execution since even if it did not need synchronization, the workload could not be reorganized to reduce the execution time.

Optional

In the last optional section of this assignment, we explore how the *Point* and *Row* strategy codes behave with different task granularities. To do that, we submit the execution of both files using a new script called *submit-numtasks-omp.sh*, that explores the different granularities by changing the number of tasks executed in every taskloop. The number of threads is specified by the user as an argument of the script. What we need to find out is how to send this value (number of tasks) to the *mandel-omp.c* file.

What we do in both strategies using taskloop, is to add the *num_tasks* clause next to the taskloop construct. Since we found out that the option **-u** in the script is the one that in the code adds the value of the argument to a variable called *user_param* (declared as a global variable in *mandel_omp.c*), we decided to put it inside the brackets *num_tasks* clause (`#pragma omp taskloop num_tasks(user_param)`. This line of code is placed according to the strategy implemented). This variable will have the number of tasks determined by a list in the script (800, 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1 tasks). When this is executed with *sbatch*, a file called *mandel-omp-ntasks.ps* is generated. If we open it using *gs* we see the following plot (one for each strategy):

Row strategy:

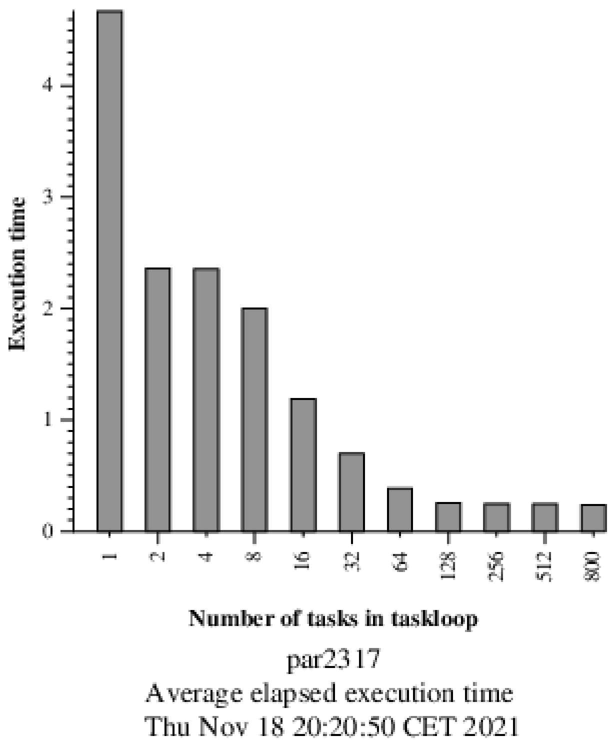


fig. 23: granularity performance comparison

Point strategy:

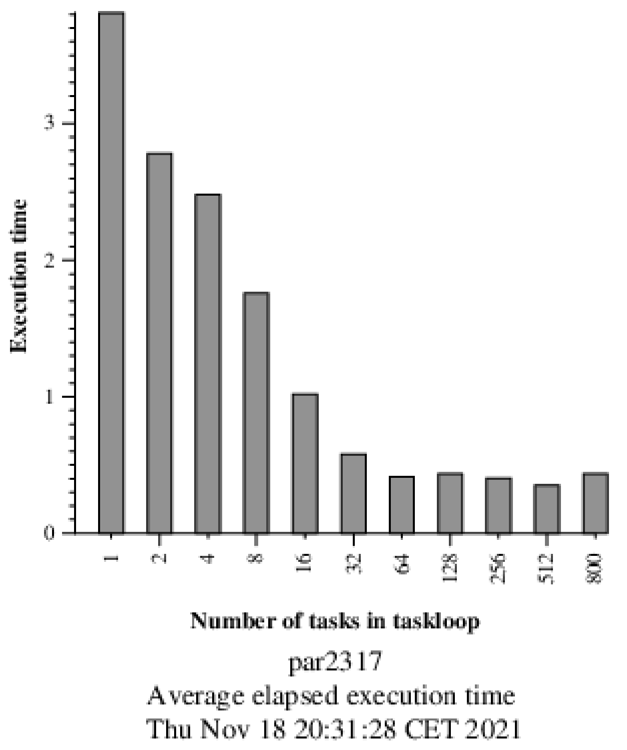


fig. 24: granularity performance comparison

As we can see in the previous plots, in both strategies the execution time decreases, somewhat logarithmically, when the number of tasks in the taskloop increases. Although, we see a little difference with the *Row* strategy, where thread number one takes a little bit more execution time than expected, in comparison to the *Point* strategy.

Conclusions

In the first part, we became familiar with the scripts that we would work with during the rest of the sessions of this lab. We learned to print the graphical representation of the Mandelbrot set on the screen and to select specific areas of the image with the cursor. Once we were familiar with the code to be dealt with, we began studying the decomposition of the tasks.
Using the *Tareador* tool, we were able to generate graphs of the different executions using specific scripts with different strategies: the row and the point.

In the next section we explored the different options that *OpenMP* allowed us to control the execution of the script. Once we had modified the code, we analyzed its scalability.

Later, we learned about the taskloop directive that generates tasks out of the loop iterations. With this directive we tried different combinations in the code to alter the granularity and number of tasks.

Finally, in the optional section, we explored how the versions of row and point behave with different granularities.

In this assignment we have verified the importance of correctly parallelizing the tasks and how to add or remove implicit barriers and speed up execution.