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

Teacher:

Jordi Tubella

Students:

Eric Hurtado (par2114)

Jordi Pitarch (par2118)

Index:

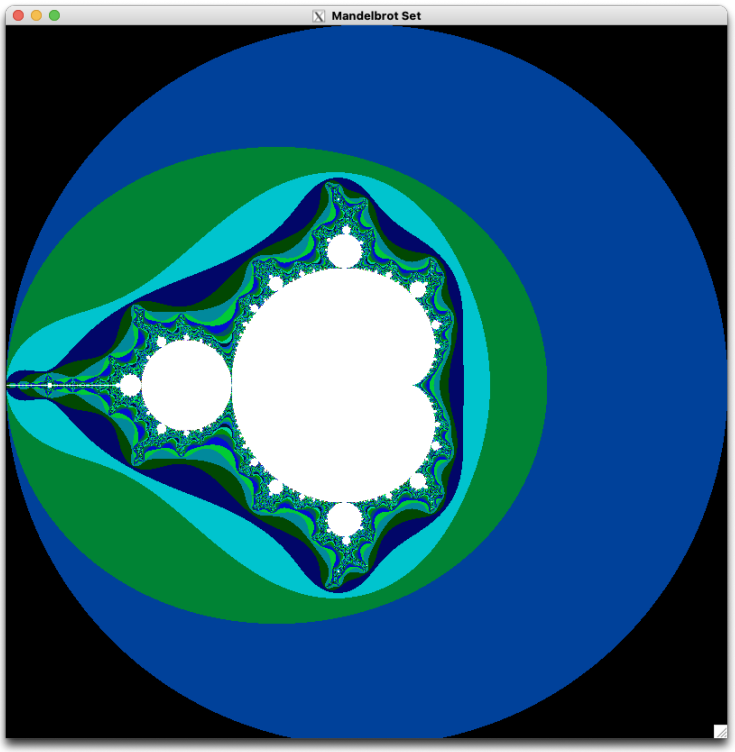
1. Introduction

In this laboratory we are going to learn the Mandelbrot set and how this is made by the program given by the teachers, and how to parallelize this with different clauses and with different granularity.

2. Task decomposition analysis for the Mandelbrot set computation

2.1 The Mandelbrot set

In this laboratory assignment you are going to explore the tasking model in OpenMP to express iterative task decompositions. But before that you will start by exploring the most appropriate ones by using Tareador. The program that will be used is 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.



We have several options like:

-o to write computed image and histogram to disk (default no file generated)

-h to produce histogram of values in computed image (default no histogram)

-d to display computed image (default no display)

-i to specify maximum number of iterations at each point (default 1000)

-w to specify the size of the image to compute (default 800x800 elements)

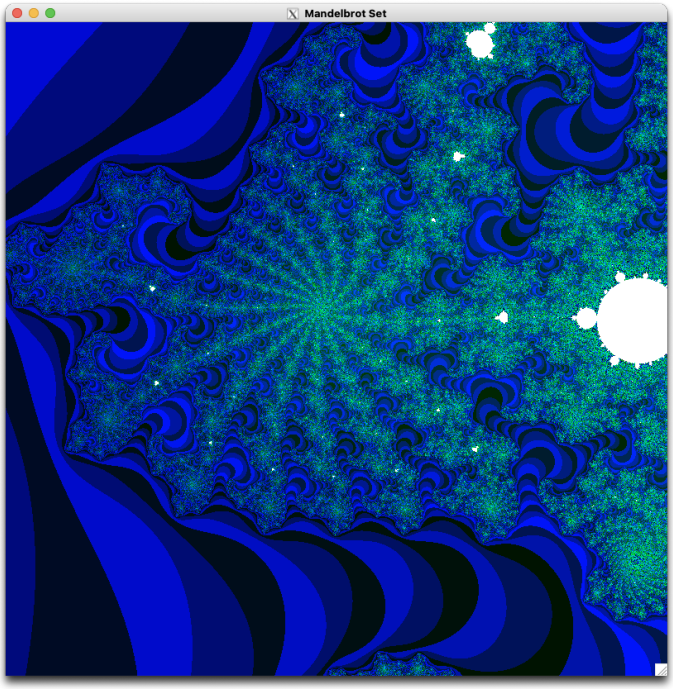
-c to specify the center x0+iy0 of the square to compute (default origin)

-s to specify the size of the square to compute (default 2, i.e. size 4 by 4)

In order to use these options we have executed several times the program with a wide variety of options.

First we have tried:

./mandel-seq -d -c -0.737 0.207 -s 0.01 -i 100000



./mandel-seq -h -i 10000 -o

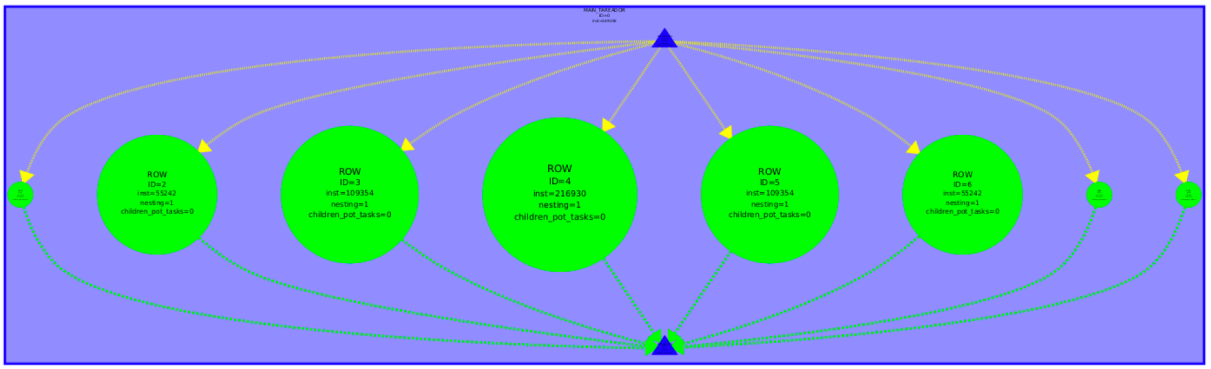
We got a file which will be used to compare with the different versions that we will be obtaining parallelizing the different strategies that are proposed to do later.

2.2 Task decomposition analysis with Tareador

In order to analyze the potential parallelism for the row strategy, with granularity of one iteration of the row loop per task, we have changed the code adding a task each time we generate a new row. This decision has been taken because we want to parallelize the program. We have done the changes that you can see in the following picture.

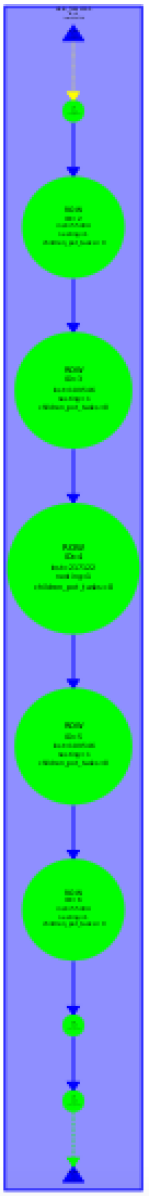
```
// Calculate points and generate appropriate output
for (int row = 0; row < height; ++row) {
    tareador_start_task("ROW");
    for (int col = 0; col < width; ++col) {
        complex z, c;

        z.real = z.imag = 0;

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

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

        output[row][col]=k;

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

        if (output2display) {
            /* Scale color and display point  */
            long color = (long) ((k-1) * scale_color) + min_color;
            if (setup_return == EXIT_SUCCESS) {
                XSetForeground (display, gc, color);
                XDrawPoint (display, win, gc, col, row);
            }
        }
    }
    tareador_end_task("ROW");
}
```

After the row decomposition we have used tareador.sh in order to generate a reasonable task graph.



We can see that the tasks in green are way bigger than the other tasks generated. If we want a parallel program, we should solve it. On the other hand, we can see that there are no dependencies between the green tasks, but they all have dependencies from the main task.

Now we are going to execute the same program with the option “-d”.



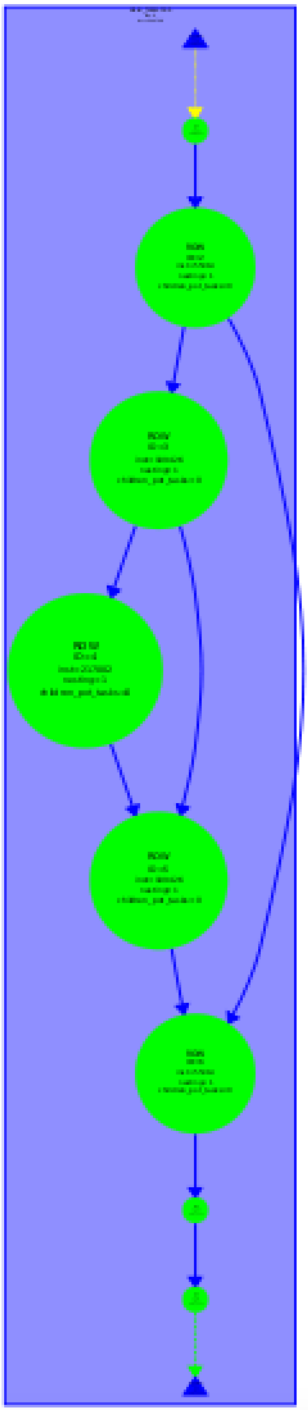
Which are the two most important characteristics for the task graph that is generated? Which part of the code is making a big difference with the previous case? How will you protect this section of code in the parallel OpenMP code that you will program in the next sessions? Save the new TDG generated for later inclusion in the deliverable.

You can see that we have executed using the -d option, which means that to display a computed image. We can see the result on the right side.

As we have seen before, we have big green tasks, but now they do not depend on the initial task. They are executed after the previous task, like a sequential program.

The part of code that is making a big difference with the previous case is the condition “if(output2display)”, inside we have functions called “XSetForeground '' and “XDrawPoint ''.

To protect this section of code in the parallel OpenMP code, we can do a #pragma omp critical right before the functions called before. Doing this, we can avoid data races done by several threads trying to access the same variable that stores the color.



Finally we have tested the option -h.

What does each chain of tasks in the task graph represent? Which part of the code is making the big difference with the two previous cases? How will you protect this section of code in the parallel OpenMP code that you will program in the next sessions? Don't forget to save the new TDG generated for later inclusion in the deliverable.

As we can see on the left, we have a similar serialization as we had in the previous case. Although there are the initial green tasks and the last two tasks which have no more than one dependency.

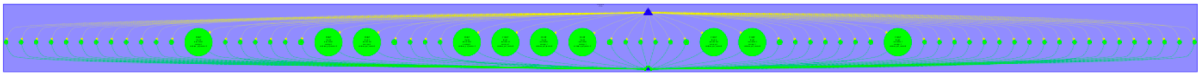
The part of the code which is making the difference in this section is "if(output2histogram)" which is executed when we use the option "-h", which means that we produce a histogram of values in a computed image. The problem is that we try to access the "k-1" position of histogram. As a result of the access to a position that does not exist, we result in a sequential program.

In order to protect this code we should create a region that should be accessible to one thread each time. This can be done using "#pragma omp atomic". We decided this rather than critical because this is faster and ensures us the serialization of the operation "histogram[k-1]++".

On the other hand, we will implement the point strategy. In order to implement this strategy, we have modified the code, as you can see in the following image:

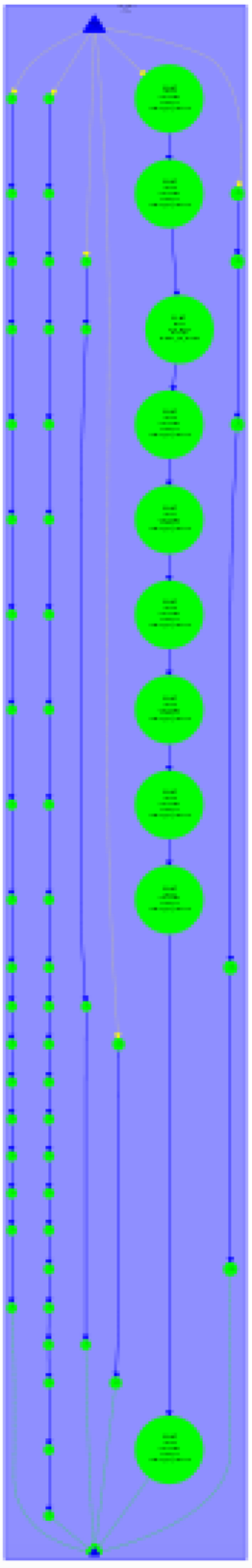
```
void mandelbrot(int height, int width, double real_min, double imag_min,
                double scale_real, double scale_imag, int maxiter, int **output) {

    // Calculate points and generate appropriate output
    for (int row = 0; row < height; ++row) {
        for (int col = 0; col < width; ++col) {
            tareador_start_task("POINT");
            complex z, c;

            z.real = z.imag = 0;

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

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

            output[row][col]=k;

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

            if (output2display) {
                /* Scale color and display point  */
                long color = (long) ((k-1) * scale_color) + min_color;
                if (setup_return == EXIT_SUCCESS) {
                    XSetForeground (display, gc, color);
                    XDrawPoint (display, win, gc, col, row);
                }
            }
            tareador_end_task("POINT");
        }
    }
}
```

We have moved the “tareador_start_task” and the “tareador_end_task” to the innermost for in order to create a task every time we make a point. We compile this code in the Makefile, and thanks to the tareador, we obtained the following graph:



As we have said before, we can see that if we generate a task each time we are in the most innermost loop, we will create a lot of simultaneous tasks as we can see it. We have something similar to the row strategy because we do not have dependencies between tasks, and we have tasks bigger than others.

Next, we have executed this point strategy with the option “-d” due to the fact that we want to see the differences between this execution and the last execution with row strategy.



When we execute the code like we said before, we obtained the graph which can be seen on the right side. Like in the row strategy, we have the same type of execution, but now we have more tasks due to the point strategy. We have the same dependencies as in the previous strategy, we only have the dependence from the previous task, serialization, and we also see an imbalance between the task due to the fact that it is not parallelized.

Finally, we executed the code with the option “-h”, which means that we produce a histogram of values in the computed image. We obtained the following graph, which can be seen in the left hand.

This graph is quite different from the other obtained in the row strategy, because it seems to be parallelized. Despite the fact said before, this should be considered sequential due to the fact that the bigger task are executed in sequential order, as a result the execution time will not be as different as if the bigger task were done in different threads.

In conclusion, after seeing both implementations and the execution without flags and with the flags “-d” and “-h”, we can draw some conclusions in terms of the main characteristics of the graphs. The main difference is that in the point strategy we have way more tasks than in the row strategy, this is caused by the creation task in the innermost loop.

If we should implement only one strategy, we would choose the row strategy because the bigger task will always be implemented sequentially, as a result if we implement the row strategy we will reduce the task creation, which is one of the biggest problem when we have a lot of small tasks because we have a problem with the overheads.

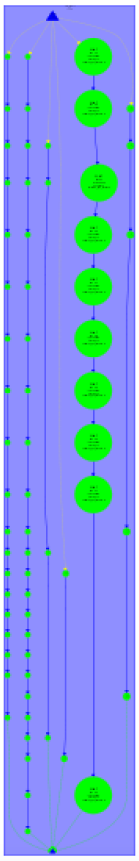
3. Implementation and analysis of task decompositions in OpenMP

3.1 Point decomposition strategy

Now we are going to implement the Mandelbrot using task, firstly we executed the images obtained by the code done by the teachers in order to see the result using the clause “firstprivate(row, col)” in the innermost loop and adding the atomic in histogram[k-1]++ and the critical in if setup_return == EXIT_SUCCESS.

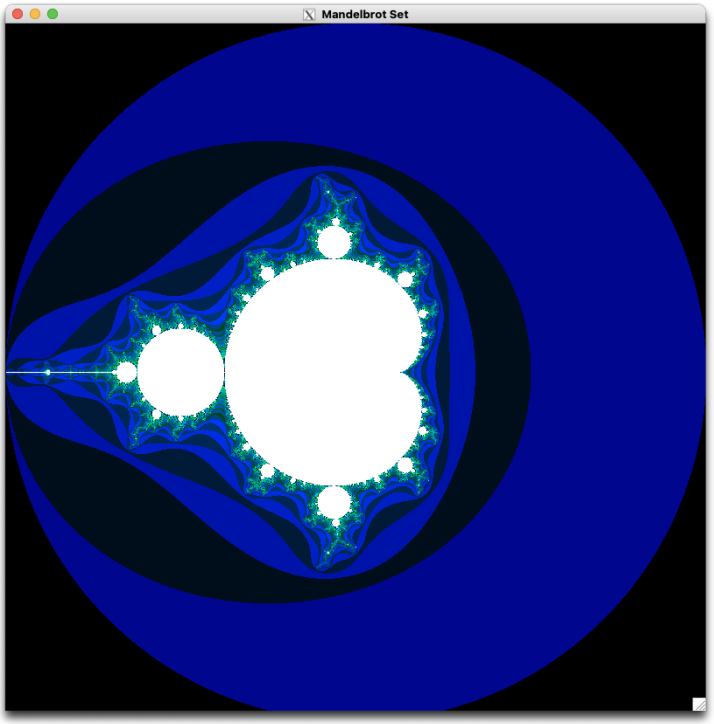
```
void mandelbrot(int height, int width, double real_min, double imag_min,
                double scale_real, double scale_imag, int maxiter, int **output) {

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

If we execute in tareador we get the following result, which is similar to the result obtained in the previous point:

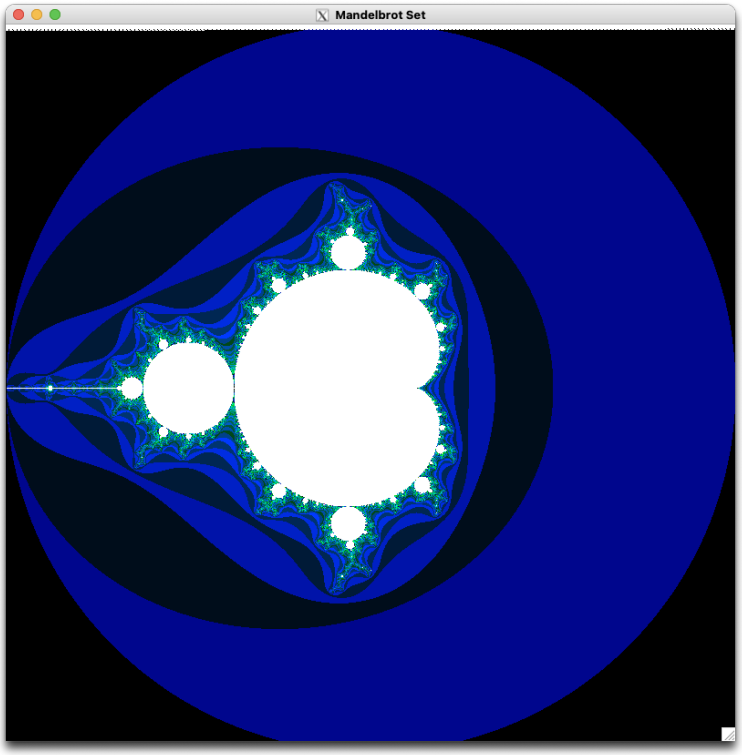


If we execute this code with one or two threads like:

OMP_NUM_THREADS=1 ./mandel-omp -d -h -i 10000



or: OMP_NUM_THREADS=2 ./mandel-omp -d -h -i 10000



If we compare both images with the command “cmp output_omp_1.out output_omp_2.out” we can see that both are the same. If we compare these images with the obtained at the beginning of this laboratory, we can appreciate that the colors in which the image is displayed are different now.

If we execute this with the submit-omp.sh with 1 thread we get a time of:

Total execution time (in seconds): 2.966194

But if we execute with 8 threads we get a time of:

Total execution time (in seconds): 1.471757

Seeing the result, we can appreciate that if we use more threads to execute the program, the time required to execute the program is less. In order to see this we have tried the submit-strong-omp.sh.

If we interpret the results obtained, we can see that the time that is necessary for the execution is not reduced if we use more threads. As a result of these facts, we see that the strong scalability is not as good as we would have speculated.

```
Resultat de l'experiment (tambe es troben a  ./elapsed-boada-11.txt  i  ./speedup-boada-11.txt  )
#threads        Elapsed min
1   2.967353
2   1.619433
3   1.363778
4   1.459814
5   1.363463
6   1.520275
7   1.391518
8   1.468439
9   1.484940
10   1.501415
11   1.474631
12   1.510927
13   1.490232
14   1.606413
15   1.645242
16   1.727078
17   1.697972
18   1.795059
19   1.768237
20   1.833830

#threads        Speedup
1   .87926781882708258842
2   1.61111821236198101434
3   1.91313982187716769151
4   1.78728111937548208196
5   1.91358181336787283556
6   1.71620134515137064018
7   1.87500125761937682444
8   1.77678337336450475641
9   1.75703934165690196236
10   1.73775938031790011422
11   1.76932263054282732425
12   1.72681936321212077089
13   1.75079987545563375367
14   1.62417634817447318964
15   1.58584451405932987365
16   1.51070073268260032262
17   1.53659659876605739081
18   1.45348871541269674144
19   1.47553636757968530236
20   1.42275892530932529187
```

Now we have decided to see the results with Modelfactors to see the results obtained, we have used:

sbatch submit-strong-extrae.sh mandel-omp

And if we open the modelfactor.py

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.58	0.36	0.31	0.33	0.34
Speedup	1.00	1.62	1.85	1.73	1.72
Efficiency	1.00	0.40	0.23	0.14	0.11

Table 1: Analysis done on Tue Oct 18 11:13:44 AM CEST 2022, par2118

Overview of the Efficiency metrics in parallel fraction, ϕ=99.92%					
Number of processors	1	4	8	12	16
Global efficiency	95.43%	38.60%	22.08%	13.77%	10.23%
Parallelization strategy efficiency	95.43%	46.83%	29.86%	19.62%	15.27%
Load balancing	100.00%	91.25%	54.99%	36.34%	25.14%
In execution efficiency	95.43%	51.33%	54.30%	53.98%	60.75%
Scalability for computation tasks	100.00%	82.41%	73.94%	70.18%	67.02%
IPC scalability	100.00%	79.83%	74.20%	72.29%	69.14%
Instruction scalability	100.00%	103.70%	104.22%	104.29%	104.14%
Frequency scalability	100.00%	99.56%	95.61%	93.09%	93.07%

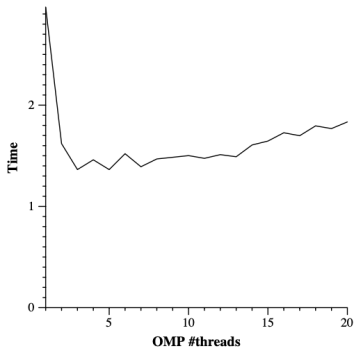
Table 2: Analysis done on Tue Oct 18 11:13:44 AM CEST 2022, par2118

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	102400.0	102400.0	102400.0	102400.0	102400.0
LB (number of explicit tasks executed)	1.0	0.8	0.83	0.87	0.85
LB (time executing explicit tasks)	1.0	0.89	0.89	0.91	0.9
Time per explicit task (average us)	4.9	5.69	5.93	5.99	6.03
Overhead per explicit task (synch %)	0.0	100.37	264.74	499.37	716.98
Overhead per explicit task (sched %)	5.25	29.79	23.47	25.02	21.39
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

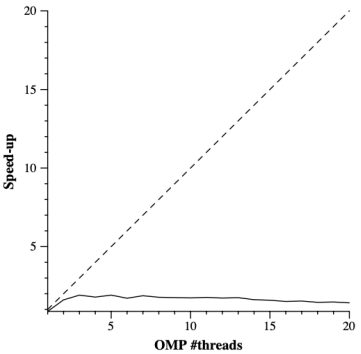
Table 3: Analysis done on Tue Oct 18 11:13:44 AM CEST 2022, par2118

We can see that the speedup increases with the number of threads, but this number is so low if we compare these two numbers. As a result, this speedup is not appropriate.

On the other hand, if we see the scalability, these decrease so fast as the number of threads increase. As a consequence, the scalability is not appropriate due to the fact that we don't use the 100% or something similar of the threads.

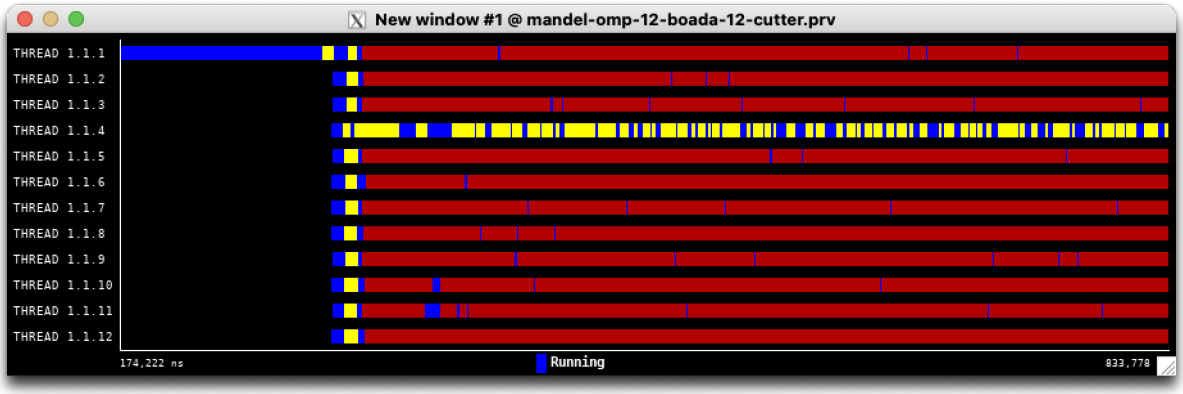


par2118
Min elapsed execution time
Generated by par2118 on Tue Oct 18 11:01:57 AM CEST 2022

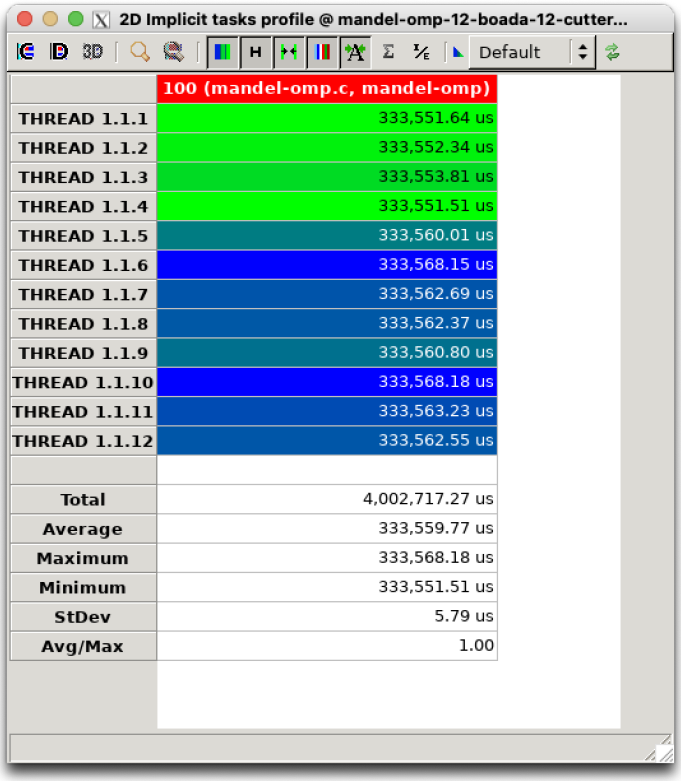


par2118
Speed-up wrt sequential time
Generated by par2118 on Tue Oct 18 11:01:57 AM CEST 2022

Now, we have decided to see the results using paraver.



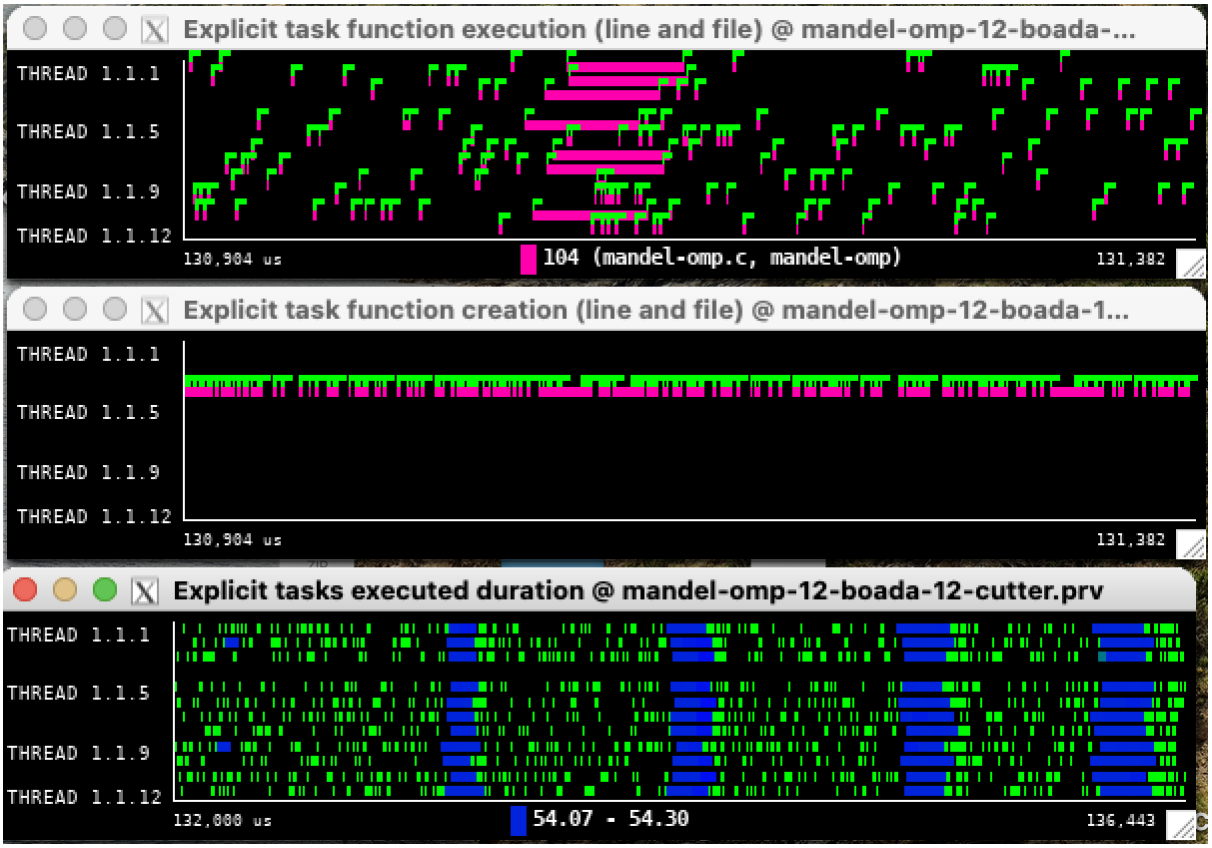
First of all we can see the execution time graphic, we have ampliate the start of the execution in order to see that in the beginning there is only one thread executing the code while the others do not do anything. Later the start running all threads but after a little, most of the threads are most time synchronizing while only one pass most of the time executing the code, this is caused by the among of overheads.



	100 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	333,551.64 us
THREAD 1.1.2	333,552.34 us
THREAD 1.1.3	333,553.81 us
THREAD 1.1.4	333,551.51 us
THREAD 1.1.5	333,560.01 us
THREAD 1.1.6	333,568.15 us
THREAD 1.1.7	333,562.69 us
THREAD 1.1.8	333,562.37 us
THREAD 1.1.9	333,560.80 us
THREAD 1.1.10	333,568.18 us
THREAD 1.1.11	333,563.23 us
THREAD 1.1.12	333,562.55 us
Total	4,002,717.27 us
Average	333,559.77 us
Maximum	333,568.18 us
Minimum	333,551.51 us
StDev	5.79 us
Avg/Max	1.00

Now we have decided to see the implicit tasks, and see which threads are executing then, if we use the previous image, we can see that all threads execute the same among time implicit tasks.

By the other hand if we see the explicit task functions, we can see that they are generated only by one thread, but they are executed by all the threads. This causes a big overhead because the others have to wait until they create the tasks, black areas.



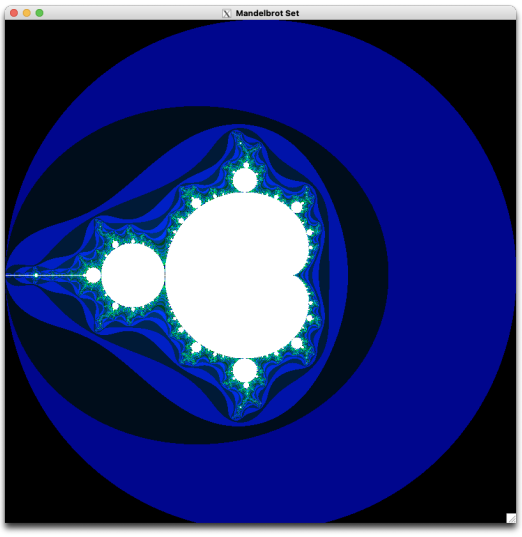
Finally, after seeing all the results obtained by the modelfactor and the paraver, we can consider that the strategy made is not appropriate to this problem because several factors as the scalability, which is so bad in relation to the number of threads and the dependence on the explicit tasks because they depend on one thread who is the creator of all explicit tasks.

Now we have decided to implement another version of the code, now using taskloop. We have changed the code like the following image:

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

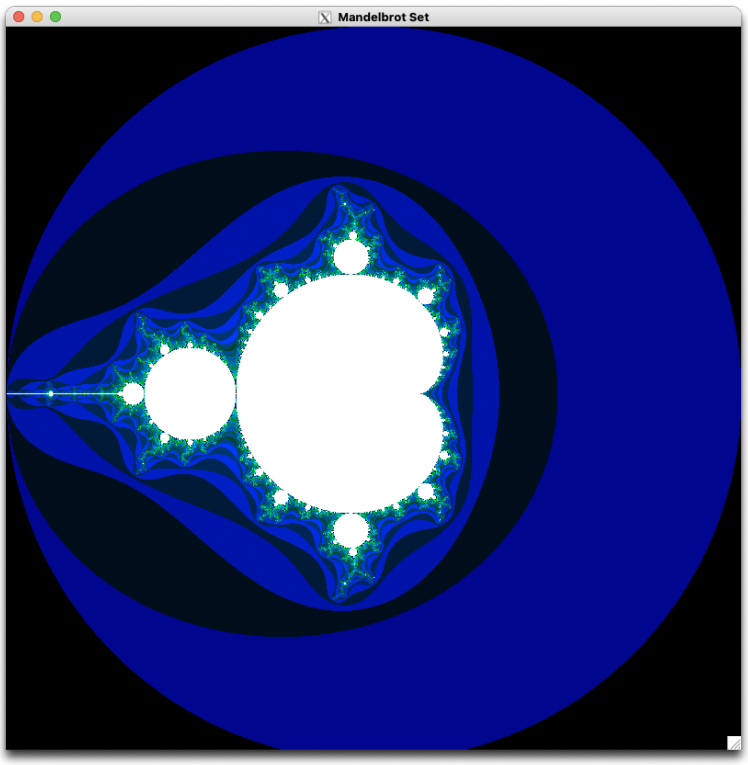
If we execute interactively with one thread like:

OMP_NUM_THREADS=1 ./mandel-omp -d -h -i 10000



If we execute interactively with two thread like:

OMP_NUM_THREADS=2 ./mandel-omp -d -h -i 10000



We can see that we have the same images as before, with the same colors. The difference between these images and the first are the colors which are different.

If we execute this program with the submit-omp.sh, we can see that the execution time with 1 thread is: Total execution time (in seconds): 2.560722, while with 8 threads is: Total execution time (in seconds): 0.551776. If we compare these values with the obtained in the past we can see that the execution time with 1 thread is almost the same, but with 8 threads this time is a third part of the time obtained. This is caused due to the fact that now we don't have as overheads as before.

Now we have executed submit-strong-extrae.sh.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.41	0.14	0.13	0.16	0.20
Speedup	1.00	2.94	3.19	2.62	2.11
Efficiency	1.00	0.73	0.40	0.22	0.13

Table 1: Analysis done on Tue Oct 18 01:02:50 PM CEST 2022, par2118

Overview of the Efficiency metrics in parallel fraction, ϕ=99.93%					
Number of processors	1	4	8	12	16
Global efficiency	99.57%	73.17%	39.77%	21.79%	13.12%
Parallelization strategy efficiency	99.57%	76.81%	44.93%	25.69%	15.76%
Load balancing	100.00%	95.16%	96.48%	95.37%	93.21%
In execution efficiency	99.57%	80.71%	46.56%	26.94%	16.91%
Scalability for computation tasks	100.00%	95.27%	88.52%	84.82%	83.27%
IPC scalability	100.00%	97.49%	97.20%	96.76%	96.20%
Instruction scalability	100.00%	99.43%	98.67%	97.93%	97.19%
Frequency scalability	100.00%	98.28%	92.30%	89.51%	89.06%

Table 2: Analysis done on Tue Oct 18 01:02:50 PM CEST 2022, par2118

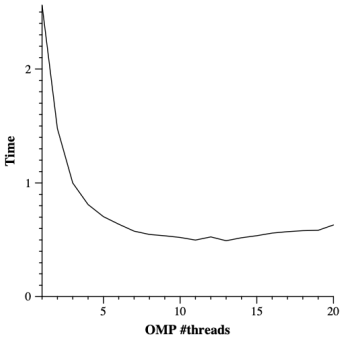
Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	3200.0	12800.0	25600.0	38400.0	51200.0
LB (number of explicit tasks executed)	1.0	0.94	0.83	0.53	0.48
LB (time executing explicit tasks)	1.0	0.95	0.96	0.95	0.93
Time per explicit task (average us)	128.92	33.83	18.2	12.66	9.67
Overhead per explicit task (synch %)	0.06	26.55	108.96	265.68	501.26
Overhead per explicit task (sched %)	0.37	3.65	13.68	23.72	33.61
Number of taskwait/taskgroup (total)	320.0	320.0	320.0	320.0	320.0

Table 3: Analysis done on Tue Oct 18 01:02:50 PM CEST 2022, par2118

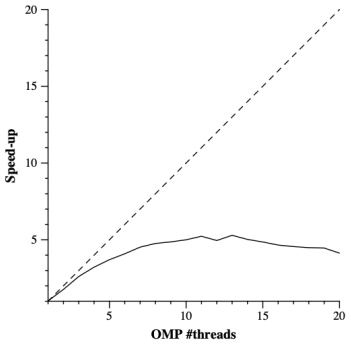
We have opened the model-factor-table.pdf in order to see the differences. If we see the load balancing, we can appreciate that now this is much bigger than in the other case, which was around 15%, but the in execution efficiency is lower now than before.In the case of the tasks executed we can see that now this number is not always the same, it depends on the number of threads, and this is reduced until it starts to grow, also being higher than the beginning. Despite the number of threads the number of taskloop remains constant, always being 320. And the granularity is so little that the overhead produced between task is higher than the cost of doing it. If we compare the elapsed time executing the program, we can see that now is way faster than in the “task” program.

As a result of the among of taskloop we can see that we always have the same amount of taskwait, this is because there are always the same rows, as a result if this number does not change, this will remain constant. They occur as a consequence of waiting until the end of the execution of each row in order to get always the same result, avoiding dataraces.

If we compare the plots obtained with the ghost script, we can see that now the speedup is higher than before and the time required to execute the program is higher when we have more than twelve threads, this is due to the overheads of the among threads.

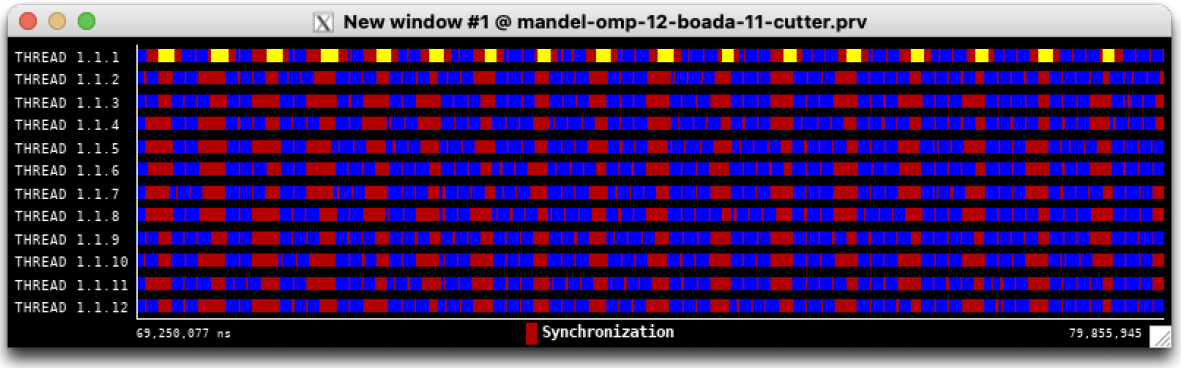


par2118
Min elapsed execution time
Generated by par2118 on Tue Oct 18 01:14:05 PM CEST 2022



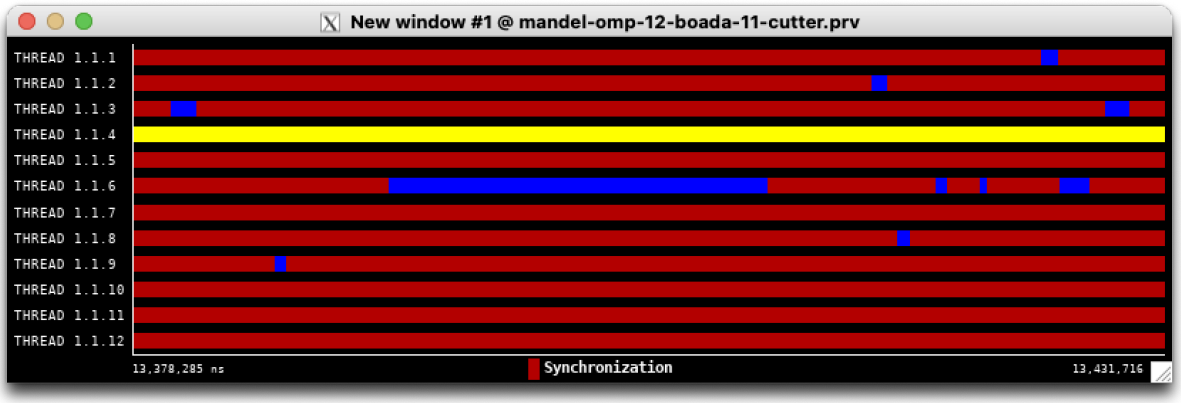
par2118
Speed-up wrt sequential time
Generated by par2118 on Tue Oct 18 01:14:05 PM CEST 2022

If we open paraver, we can see the following image:



In the image we can see that there is one thread who makes the forks while the other threads do the task and remain in synchronization until they can start the following task. In order to solve this problem, we have tried to add the nogroup clause to the taskloop in order to eliminate the implicit taskgroup.

Now we have the following image, which shows us that now we spend more time in synchronization than before. But the task are not as synchronized as before.



In the other hand if see the table of modelfactors, we can see that the speedup now is a little bit higher, this is due to the fact that now we do not have number of taskwait/taskgroup. As a result we get a better performance. The scalability have improved as a consequence of a better speedup.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.46	0.13	0.11	0.13	0.18
Speedup	1.00	3.65	4.21	3.47	2.54
Efficiency	1.00	0.91	0.53	0.29	0.16

Table 1: Analysis done on Thu Oct 20 08:02:12 PM CEST 2022, par2118

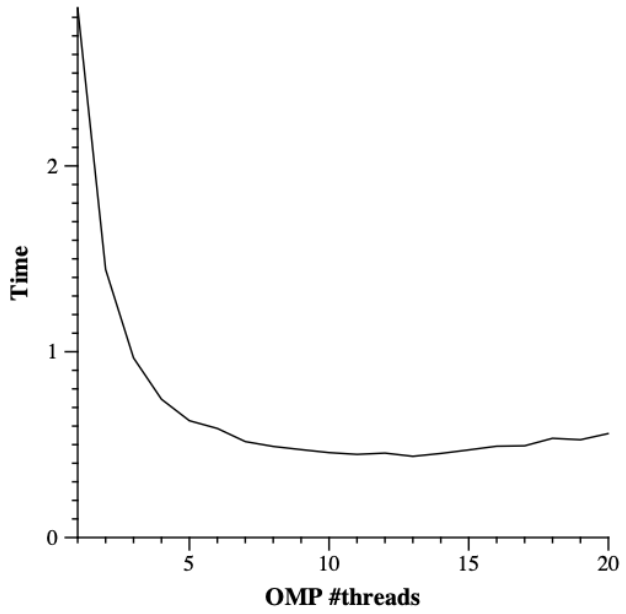
Overview of the Efficiency metrics in parallel fraction, ϕ=99.88%					
Number of processors	1	4	8	12	16
Global efficiency	99.71%	91.01%	52.54%	28.91%	15.86%
Parallelization strategy efficiency	99.71%	95.32%	59.49%	34.01%	18.98%
Load balancing	100.00%	98.99%	98.33%	97.75%	96.86%
In execution efficiency	99.71%	96.29%	60.50%	34.79%	19.60%
Scalability for computation tasks	100.00%	95.49%	88.32%	85.01%	83.56%
IPC scalability	100.00%	98.24%	97.22%	96.93%	96.36%
Instruction scalability	100.00%	99.47%	98.76%	98.06%	97.40%
Frequency scalability	100.00%	97.71%	91.98%	89.44%	89.04%

Table 2: Analysis done on Thu Oct 20 08:02:12 PM CEST 2022, par2118

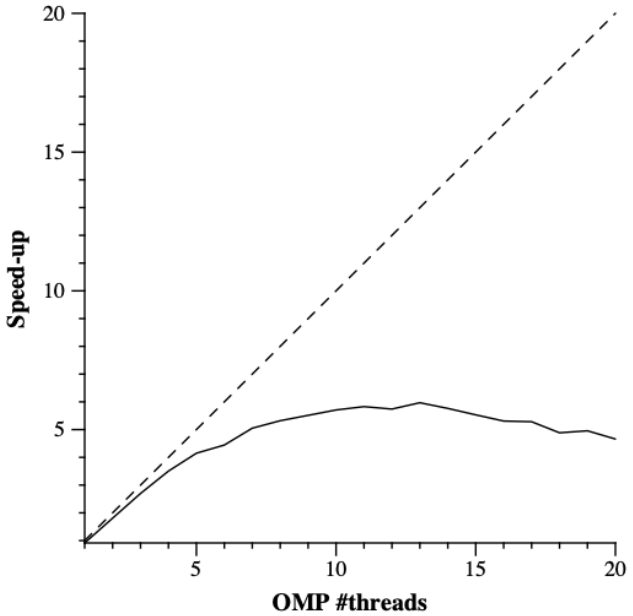
Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	3200.0	12800.0	25600.0	38400.0	51200.0
LB (number of explicit tasks executed)	1.0	0.48	0.83	0.74	0.94
LB (time executing explicit tasks)	1.0	0.99	0.98	0.98	0.97
Time per explicit task (average us)	143.26	37.5	20.27	14.04	10.71
Overhead per explicit task (synch %)	0.0	3.08	58.58	176.64	398.92
Overhead per explicit task (sched %)	0.29	1.83	9.55	17.54	28.28
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Thu Oct 20 08:02:12 PM CEST 2022, par2118

We also have executed the “strong_mandel_omp” to see the graphics of the execution. As you can see below it arrives a number of threads when the execution time is higher with more threads as we had before.



par2118
Min elapsed execution time
Generated by par2118 on Thu Oct 20 08:07:03 PM CEST 2022



par2118
Speed-up wrt sequential time
Generated by par2118 on Thu Oct 20 08:07:03 PM CEST 2022

In relation to the code, we have added the clausule “firstprivate(row)” as a result of the necessity of privatizing it in each task.

```
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 nogroup firstprivate(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){
                #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);
                        }
                }
            }
        }
    }
}
```

3.2 Row decomposition strategy

We have implemented a new strategy, which consists of making the taskgroup out of the row loop, like the image below.

```
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
    #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;

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

If we make the “submit-strong-extrae” of the previous code, we can see the scalability and more analysis of the execution of this new implementation.

As we can see in the tables, we can see that now we have less elapsed time if we increment the number of threads of the execution. This is due to the fact that the number of tasks

created is much smaller than before. Also, we can see that the number of taskwait/taskloop is always 1 because to only be executed one time the taskloop.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.46	0.13	0.07	0.05	0.04
Speedup	1.00	3.55	6.64	9.51	12.31
Efficiency	1.00	0.89	0.83	0.79	0.77

Table 1: Analysis done on Fri Oct 21 01:50:43 PM CEST 2022, par2118

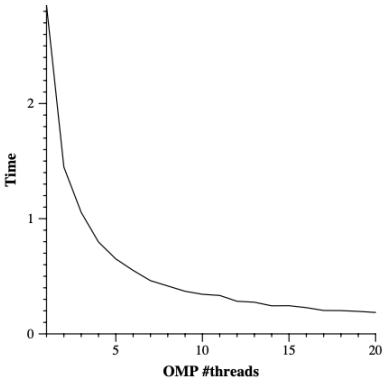
Overview of the Efficiency metrics in parallel fraction, ϕ=99.88%					
Number of processors	1	4	8	12	16
Global efficiency	99.98%	88.71%	83.25%	79.69%	77.45%
Parallelization strategy efficiency	99.98%	90.69%	92.41%	90.61%	89.15%
Load balancing	100.00%	90.76%	92.58%	90.82%	89.47%
In execution efficiency	99.98%	99.92%	99.82%	99.77%	99.64%
Scalability for computation tasks	100.00%	97.81%	90.09%	87.95%	86.87%
IPC scalability	100.00%	98.64%	97.36%	97.00%	95.81%
Instruction scalability	100.00%	100.00%	99.99%	99.99%	99.99%
Frequency scalability	100.00%	99.17%	92.54%	90.67%	90.68%

Table 2: Analysis done on Fri Oct 21 01:50:43 PM CEST 2022, par2118

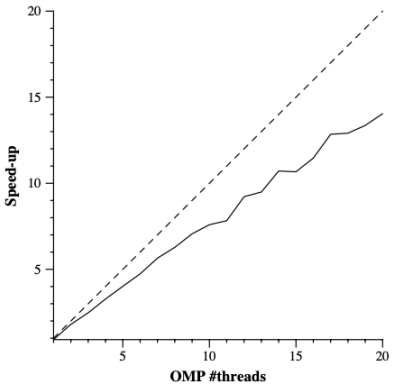
Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	10.0	40.0	80.0	120.0	160.0
LB (number of explicit tasks executed)	1.0	0.59	0.32	0.19	0.17
LB (time executing explicit tasks)	1.0	0.91	0.93	0.91	0.89
Time per explicit task (average us)	45773.13	11698.05	6350.17	4336.26	3292.06
Overhead per explicit task (synch %)	0.0	10.24	8.16	10.28	12.08
Overhead per explicit task (sched %)	0.01	0.03	0.04	0.06	0.07
Number of taskwait/taskgroup (total)	1.0	1.0	1.0	1.0	1.0

Table 3: Analysis done on Fri Oct 21 01:50:43 PM CEST 2022, par2118

If we make the “submit-strong-omp” in order to see the scalability in a graph, we can see that now the execution time of the program is always less if we increment the threads. As a consequence the speedup of the program is always higher as we increment the number of threads.

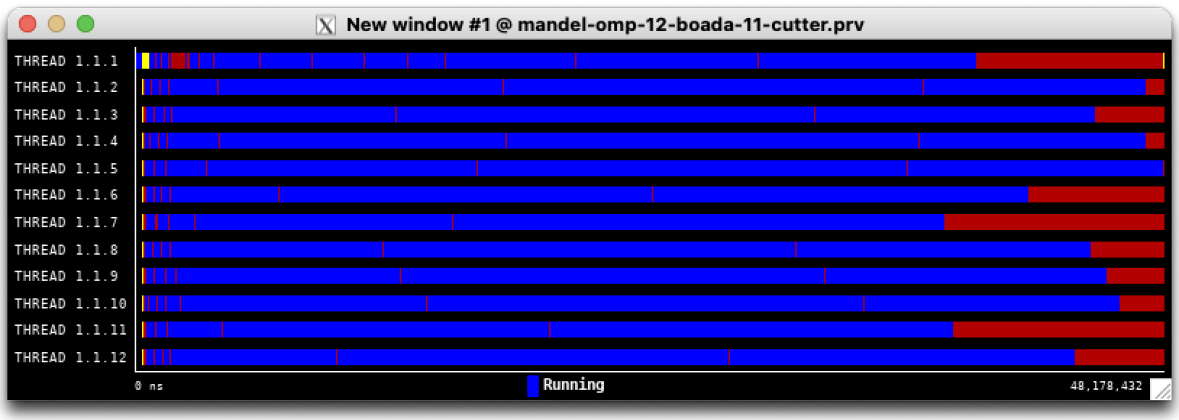


par2118
Min elapsed execution time
Generated by par2118 on Fri Oct 21 02:01:11 PM CEST 2022



par2118
Speed-up wrt sequential time
Generated by par2118 on Fri Oct 21 02:01:11 PM CEST 2022

On the other hand, if we see the “wxparaver”, we can appreciate that now most of the time is spent running the program, something that was not happening before. Now the only time when they are synchronizing is at the end as a result of waiting until the end of all threads. This makes that the balance within threads is not as good as before, but they spend less time executing the overall program. Now we do not have synchronization between task , the only synchronization is at the end of the program. Also each thread creates tasks at the beginning of the execution time.



3.3 Optional: task granularity tune

We have explored the point and row strategy codes that behave differently with the task granularity, we have done this thanks to the “submit-numtasks-omp.sh”, which explores the different granularities by changing the number of tasks executed in every taskloop.

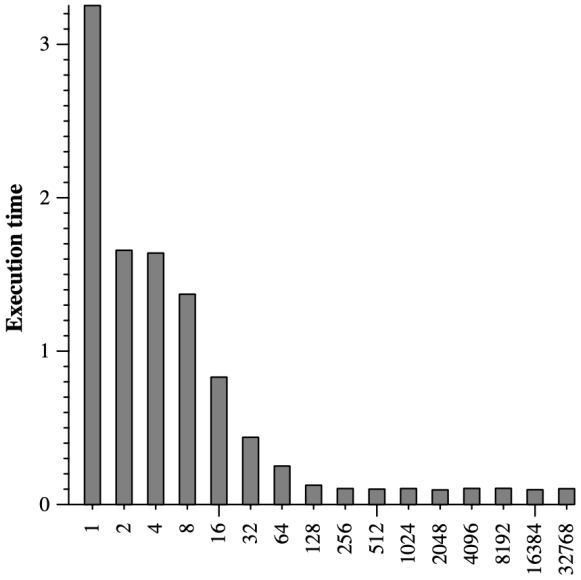
First of all we have implemented the row strategy as we can see in the following image:

```
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
    #pragma omp taskloop num_tasks(user_param)
    for (int row = 0; row < height; ++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){
                #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);
                        }
                }
            }
        }
    }
}
```

We have seen that the option “-u” in the script is which allows us to add the value of the argument to a variable called num_tasks clause. This is due to the following line of the script:

```
./$PROG -h -i $numits -u $ntasks >> $out
```

When we execute this script, we get the following plot:

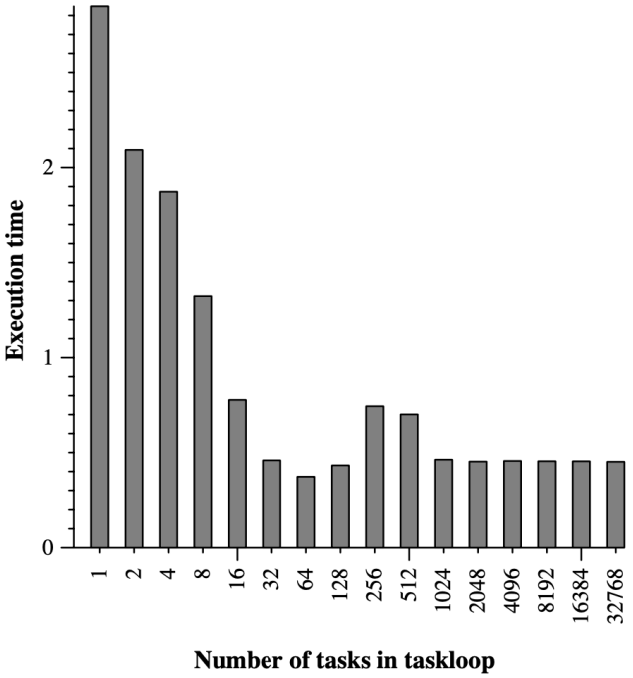


par2118
Average elapsed execution time
Fri Oct 21 02:20:47 PM CEST 2022

In the other hand, we have implemented the same strategy as we have said before in the point strategy, like we can see in the following image:

```
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 num_tasks(user_param)
        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);
                        }
                }
            }
        }
    }
}
```

If we now execute the script, we get the following plot:



As we can see in the previous plots, in both the execution time decreases when the number of tasks in the taskloop increases. Despite this, there is a big difference between the two implementations. If we see the row graphic, we can see that the time after 128 number of tasks in taskloop is always the same number, while in the point strategy this execution rises up in 256 number of tasks, and later in 1024 number of task remain constant but much higher than the row strategy. These differences are produced as a result of the amount of overheads generated in the point strategy because of the among of tasks generated each time.

4. Conclusion

Firstly, we have learned about the scripts we are going to work with during this laboratory and when using each one. We also have learned to print the Mandelbrot set and the different options available.

We have analyzed the scalability of the different versions done during this laboratory and the reasons why we should use some clauses in some cases and when not use them.

Furthermore, we have learned about the taskloop, which generates tasks out of the loop iterations. We have done several combinations, altering the granularity and the number of tasks.

Finally, in the optional part, we have explored how the row and point strategies are with different granularity.