

PAR Laboratory Assignment

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

Marc Duran Lopez: par1311
Jesús Pérez Bermejo: par1317
Spring 2022-23
April 14, 2023

Index

1. Introduction	3
2. Task decomposition and granularity analysis for the Mandelbrot set computation	4
2.1 Task decomposition analysis with Tareador	4
2.1.1 Analysis of the potential parallelism for the Row strategy of Mandel-tar.c with no additional options	4
2.1.2 With -d option	5
2.1.3 With -h option	7
2.2.1 Analysis of the potential parallelism for the Point strategy of Mandel-tar.c with no additional options	9
2.2.2 With -d option	10
2.2.3 With -h option	12
3. Point decomposition in OMP	14
3.1 Explicit tasks without taskloops	14
3.2 Taskloop without using neither num task nor grainsize	18
3.3 Taskloop without nogroup	22
4. Row decomposition strategy	25
5. Conclusions	29

1. INTRODUCTION

In this laboratory assignment, we will be exploring various methods of breaking down tasks in OpenMP. Our focus will be on utilizing these methods to compute the Mandelbrot set, which will allow us to thoroughly examine the outcomes and draw appropriate conclusions.

2. TASK DECOMPOSITION AND GRANULARITY ANALYSIS FOR THE MANDELBROT SET COMPUTATION

2.1 TASK DECOMPOSITION ANALYSIS WITH TAREADOR

Next we are going to show the analysis made with Tareador of the different task breakdown strategies and their different options respectively.

-h option: The histogram for the values in the Mandelbrot set is also computed.

-d option: The Mandelbrot set is displayed, for visual inspection.

2.1.1 ANALYSIS OF THE POTENTIAL PARALLELISM FOR THE ROW STRATEGY OF MANDEL-TAR.C WITH NO ADDITIONAL OPTIONS

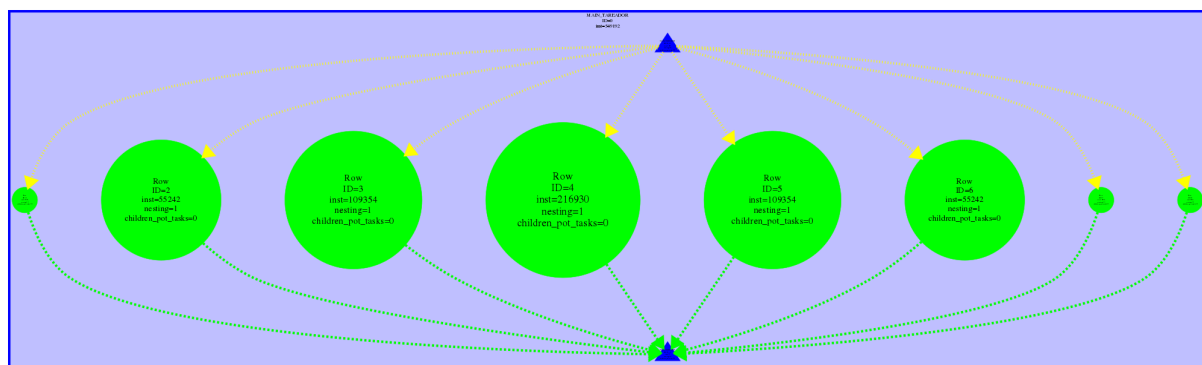


Fig.1: TDG of the mandel-tar.c with no additional options [Row strategy]

Which are the two most important characteristics for the task graph that is generated?

1. The tasks have varying sizes, with some requiring more time to complete than others. This is because certain tasks execute a greater number of iterations compared to others. Specifically, the tasks located in the middle perform more iterations than those located at the edges, as they tend to diverge more rapidly.
2. The tasks that have been created are not reliant on one another and can be considered independent. We can also verify the independence of the tasks by examining Fig. 6, which displays the arrangement of the tasks in a horizontal manner, indicating that there are no dependencies.

2.1.2 WITH -D OPTION



Fig. 2: TDG of the mandel-tar.c with -d option [Row strategy]

Which are the two most important characteristics for the task graph that is generated?

1. The tasks have varying sizes, with some requiring more time to complete than others. This is because certain tasks execute a greater number of iterations compared to others. Specifically, the tasks located in the middle perform more iterations than those located at the edges, as they tend to diverge more rapidly.
2. Upon examining the dependency graph presented in Fig. 2, we observe that the tasks are organized vertically, indicating that they are dependent. The reason for this is due to the code being added (option -d), which involves the execution of `XSetForeground` (which sets the color) and `XDrawPoint` (which uses the color previously set by `XSetForeground`). So a task starts when the previous one finishes.

Which part of the code is making the big difference with the previous case?

```
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);
    }
}
```

Fig. 3: Piece of code of the mandel-tar.c

The instructions `XSetForeground` and `XDrawPoint` are the ones that create the dependencies.

How will you protect this section of code in the parallel OpenMP code?

It could be done in different ways, one option would be to do the calculation first and then paint the pixels, but it might be too complicated. Another option that would work around this dependency would be to try to prevent the data race from occurring by using `#pragma omp critical` constructor.

2.1.3 WITH -H OPTION

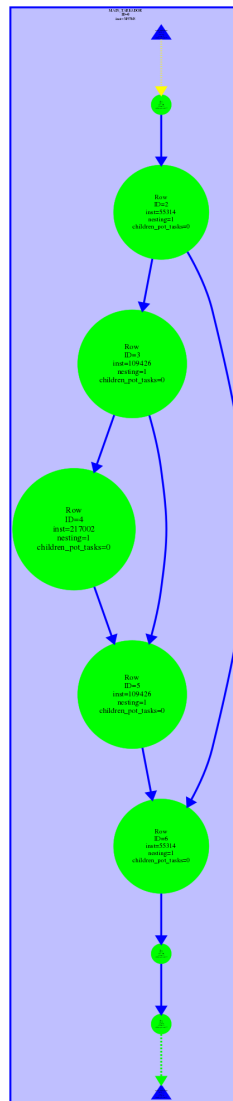


Fig. 4: TDG of the mandel-tar.c with -h option [Row strategy]

What does each chain of tasks in the task graph represents?

An iteration of the outer for, that is, one iteration of the row.

Which part of the code is making the big difference with the two previous cases?

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

Fig. 5: Piece of code of the mandel-tar.c

The part of the code is producing the dependencies between the tasks, as we can see above in fig. 4.

How will you protect this section of code in the parallel OpenMP code?

We could do it using one of the following omp statements:

- `#pragma omp critical.`
- `#pragma omp atomic.`
- `#pragma omp reduction constructors.`

2.2.1 ANALYSIS OF THE POTENTIAL PARALLELISM FOR THE POINT STRATEGY OF MANDEL-TAR.C WITH NO ADDITIONAL OPTIONS

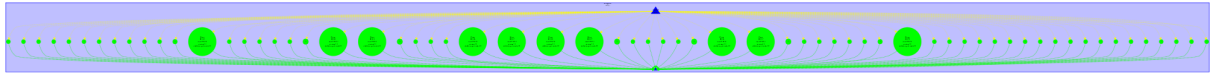


Fig. 6: TDG of the mandel-tar.c with no additional options [Point strategy]

Which are the two most important characteristics for the task graph that is generated?

The point strategy shares similar characteristics with the row strategy, with the main distinction being that it generates a greater number of tasks (more granularity).

1. The tasks have varying sizes, with some requiring more time to complete than others. This is because certain tasks execute a greater number of iterations compared to others. Specifically, the tasks located in the middle perform more iterations than those located at the edges, as they tend to diverge more rapidly.
2. The tasks that have been created are not reliant on one another and can be considered independent. We can also verify the independence of the tasks by examining Fig. 6, which displays the arrangement of the tasks in a horizontal manner, indicating that there are no dependencies.

2.2.2 WITH -D OPTION



Fig. 7: TDG of the mandel-tar.c with -d option [Point strategy]

Which are the two most important characteristics for the task graph that is generated?

The point strategy shares similar characteristics with the row strategy, with the main distinction being that it generates a greater number of tasks (more granularity).

1. The tasks have varying sizes, with some requiring more time to complete than others. This is because certain tasks execute a greater number of iterations compared to others. Specifically, the tasks located in the middle perform more iterations than those located at the edges, as they tend to diverge more rapidly.
2. Upon examining the dependency graph presented in Fig. 2, we observe that the tasks are organized vertically, indicating that they are dependent. The reason for this is due to the code being added (option -d), which involves the execution of `XSetForeground` (which sets the color) and `XDrawPoint` (which uses the color previously set by `XSetForeground`). So a task starts when the previous one finishes.

Which part of the code is making the big difference with the previous case?

```
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);
    }
}
```

Fig. 8: Piece of code of the mandel-tar.c

The instructions `XSetForeground` and `XDrawPoint` are the ones that create the dependencies.

How will you protect this section of code in the parallel OpenMP code that you will program in the next sessions?

It could be done in different ways, one option would be to do the calculation first and then paint the pixels, but it might be too complicated. Another option that would work around this dependency would be to try to prevent the data race from occurring by using `#pragma omp critical` constructor.

2.2.3 WITH -H OPTION

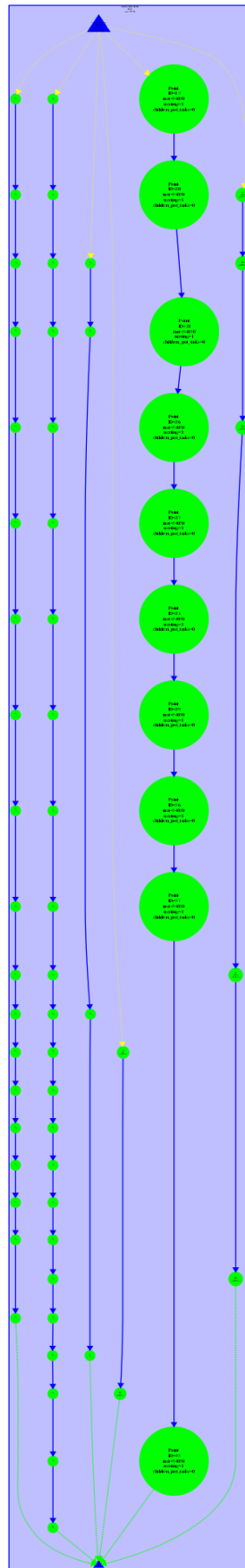


Fig. 9: TDG of the mandel-tar.c with -h option [Point strategy]

What does each chain of tasks in the task graph represents?

An iteration of the inner for, that is, one iteration of the column.

Which part of the code is making the big difference with the two previous cases?

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

Fig. 10: Piece of code of the mandel-tar.c

The part of the code is producing the dependencies between the tasks, as we can see above in fig. 9.

How will you protect this section of code in the parallel OpenMP code?

We could do it using one of the following omp statements:

- #pragma omp critical.
- #pragma omp atomic.
- #pragma omp reduction constructors.

Which is the main change that you observe with respect to the Row strategy?

The primary contrast between the Point strategy and the Row strategy lies in their respective granularities. The Point strategy utilizes a larger granularity, which results in the creation of more tasks (one for each inner iteration of the *for* loop). On the other hand, the Row strategy generates fewer tasks, as it creates only one task for each outer iteration of the *for* loop.

3. POINT DECOMPOSITION IN OMP

3.1 Explicit tasks without taskloops

In this section we implement the point strategy in the given code “mandel_omp.c” with granularity control using task. First of all, we make sure that we are using “#pragma omp task firstprivate(row, col)”, then we add the OMP directives atomic and critical.

```
// 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);

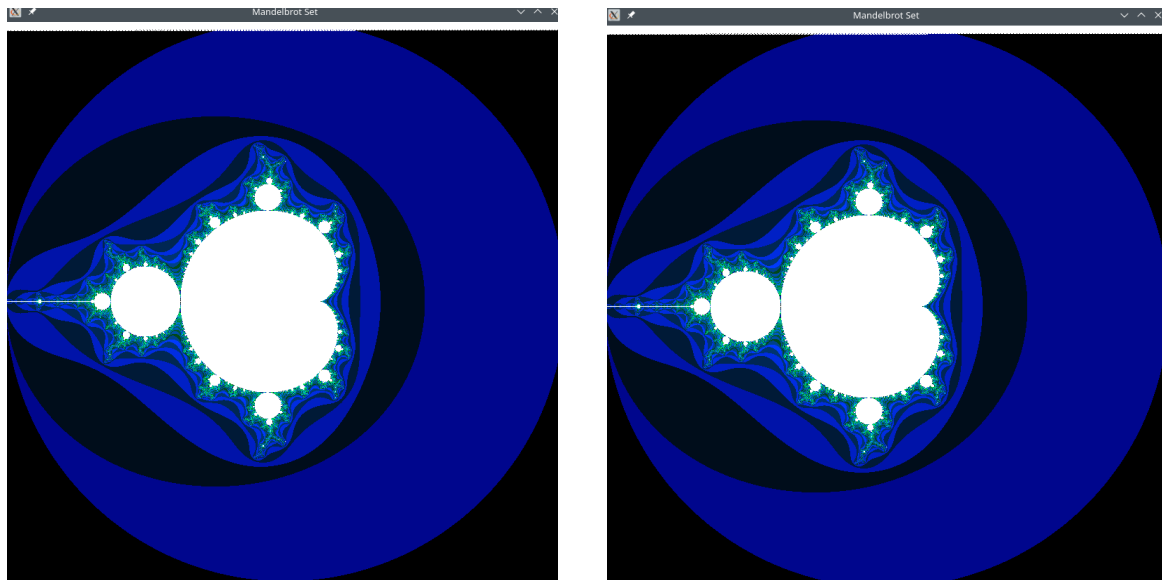
            output[row][col]=k;

            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);
                    }
                }
            }
        }
    }
}
```

Fig. 11 Piece of code of mandel_omp.c

To verify the correctness of the code, we made a comparison between 1 and 2 threads in the mandelbrot tool.



1 thread

2 threads

Fig. 12 Comparison between the display images with 1 and 2 threads

Once we got these results, we submitted the binary to the strong script to compare our results. We executed the following command: “sbatch ./submit-strong-omp.sh mandel-omp”.

We got the following results:

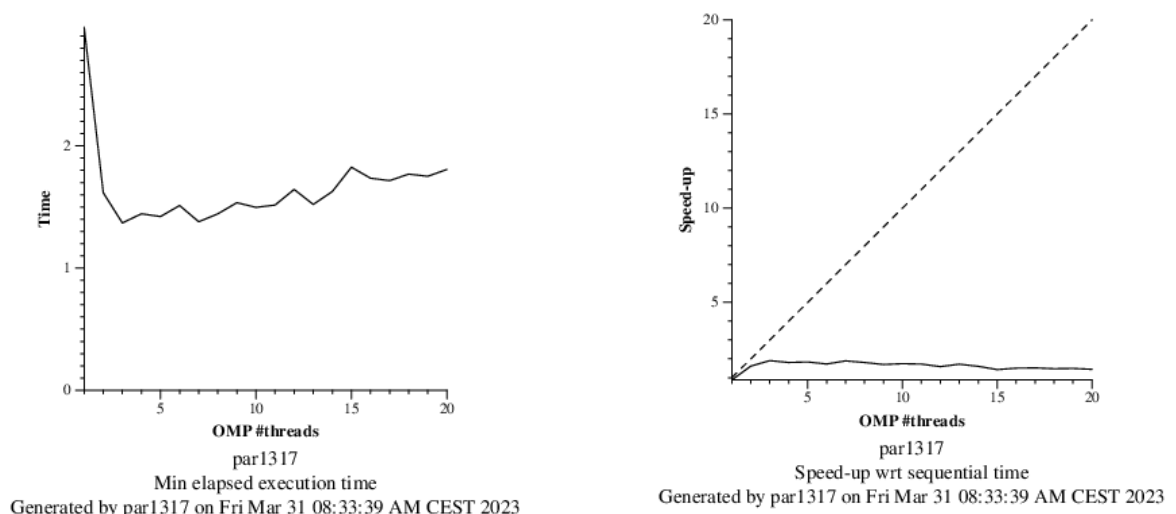


Fig. 13 Time and speed-up depending on threads

We can see that the speed-up is practically constant, but it's not linear. It can only reach 2. Scalability is not appropriate. Time execution is growing when we use more than 2 threads. We can watch this properly in the following modelfactor tables:

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.58	0.36	0.30	0.34	0.35
Speedup	1.00	1.60	1.90	1.71	1.67
Efficiency	1.00	0.40	0.24	0.14	0.10

Table 1: Analysis done on Fri Mar 31 09:07:13 AM CEST 2023, par1317

Overview of the Efficiency metrics in parallel fraction, $\phi=99.91\%$					
Number of processors	1	4	8	12	16
Global efficiency	95.35%	38.20%	22.71%	13.58%	9.98%
Parallelization strategy efficiency	95.35%	46.38%	30.27%	20.05%	15.00%
Load balancing	100.00%	92.12%	56.71%	34.04%	24.84%
In execution efficiency	95.35%	50.34%	53.39%	58.91%	60.37%
Scalability for computation tasks	100.00%	82.36%	75.01%	67.75%	66.52%
IPC scalability	100.00%	79.86%	74.98%	69.88%	68.75%
Instruction scalability	100.00%	104.05%	104.35%	104.19%	104.15%
Frequency scalability	100.00%	99.12%	95.87%	93.04%	92.90%

Table 2: Analysis done on Fri Mar 31 09:07:13 AM CEST 2023, par1317

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	0.82	0.83	0.83	0.87
LB (time executing explicit tasks)	1.0	0.89	0.89	0.88	0.89
Time per explicit task (average us)	4.9	5.7	5.9	6.07	6.05
Overhead per explicit task (synch %)	0.0	101.76	256.44	499.05	735.32
Overhead per explicit task (sched %)	5.35	30.66	23.38	22.34	22.08
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Fri Mar 31 09:07:13 AM CEST 2023, par1317

Fig. 14 Modelfactor tables

As we can see, the efficiency as we use more threads decreases, that's because the speed-up maintains constant

Next, using the paraver tool we opened the task creation option using "mandel-omp-strong-extrae"

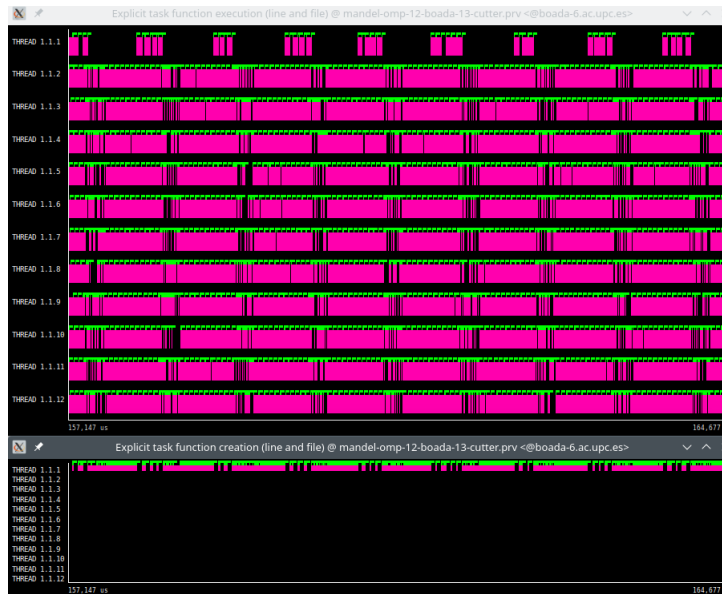


Fig. 15 Explicit task function execution and explicit task function creation graphs

As we can see, only one thread between twelve threads creates the tasks (thread 1.1.1), and all twelve threads execute them.

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	56.25 %	0.00 %	43.75 %
THREAD 1.1.2	17.92 %	82.08 %	0.00 %
THREAD 1.1.3	16.68 %	83.32 %	0.00 %
THREAD 1.1.4	17.91 %	82.09 %	0.00 %
THREAD 1.1.5	16.57 %	83.42 %	0.00 %
THREAD 1.1.6	17.92 %	82.08 %	0.00 %
THREAD 1.1.7	16.66 %	83.34 %	0.00 %
THREAD 1.1.8	17.69 %	82.31 %	0.00 %
THREAD 1.1.9	16.46 %	83.54 %	0.00 %
THREAD 1.1.10	17.74 %	82.26 %	0.00 %
THREAD 1.1.11	16.64 %	83.36 %	0.00 %
THREAD 1.1.12	17.51 %	82.49 %	0.00 %
Total	245.95 %	910.28 %	43.77 %
Average	20.50 %	75.86 %	3.65 %
Maximum	56.25 %	83.54 %	43.75 %
Minimum	16.46 %	0.00 %	0.00 %
StDev	10.79 %	22.88 %	12.09 %
Avg/Max	0.36	0.91	0.08

Fig. 16 Thread state profile

Threads between 2 and 12 are balanced, their running and synchronization percentage is almost the same, and thread number 1 has more running percentage because of the tasks creation.

The following histogram shows the duration of the explicit tasks:



Fig. 17 Histogram

3.2 Taskloop without using neither num_tasks nor grainsize

We modified the previous code and added the line “#pragma omp taskloop firstprivate(row)” between the two loops

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

            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);
        }
    }
}
}
}
}

```

Fig. 18 Piece of code of mandel_omp.c

Next, we look again for the time and the speed-up:

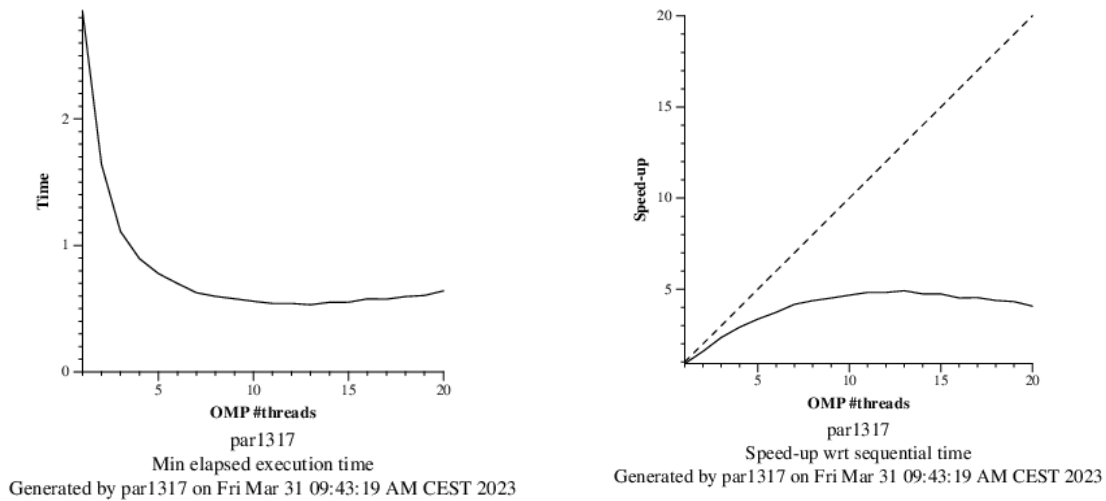


Fig. 19 Time and speed-up depending on threads

As we can see, the time when we use more than 10 threads it's practically the same, it stays constant. In the case of the speed-up, we see the same, when we arrive to 10 threads, the speed-up maintains constant.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.46	0.16	0.14	0.16	0.22
Speedup	1.00	2.94	3.33	2.83	2.13
Efficiency	1.00	0.73	0.42	0.24	0.13

Table 1: Analysis done on Fri Mar 31 09:38:45 AM CEST 2023, par1317

Overview of the Efficiency metrics in parallel fraction, $\phi=99.89\%$					
Number of processors	1	4	8	12	16
Global efficiency	99.58%	73.13%	41.54%	23.54%	13.28%
Parallelization strategy efficiency	99.58%	76.69%	46.68%	27.69%	15.85%
Load balancing	100.00%	96.04%	96.79%	95.61%	95.41%
In execution efficiency	99.58%	79.85%	48.23%	28.96%	16.61%
Scalability for computation tasks	100.00%	95.36%	88.99%	85.03%	83.81%
IPC scalability	100.00%	98.02%	97.63%	97.09%	96.73%
Instruction scalability	100.00%	99.46%	98.73%	98.02%	97.32%
Frequency scalability	100.00%	97.82%	92.32%	89.34%	89.03%

Table 2: Analysis done on Fri Mar 31 09:38:45 AM CEST 2023, par1317

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	0.93	0.84	0.53	0.47
LB (time executing explicit tasks)	1.0	0.96	0.97	0.96	0.95
Time per explicit task (average us)	143.14	37.53	20.11	14.03	10.67
Overhead per explicit task (synch %)	0.06	26.66	101.73	240.05	497.92
Overhead per explicit task (sched %)	0.36	3.74	12.56	21.25	33.32
Number of taskwait/taskgroup (total)	320.0	320.0	320.0	320.0	320.0

Table 3: Analysis done on Fri Mar 31 09:38:45 AM CEST 2023, par1317

Fig. 20 Modelfactor tables

Here we can see that the efficiency stills being quite bad as we use more threads, but it's a little bit better than previously. That's because we are using taskloops, and we can see in Table 3 the number of taskwait/taskgroup.

Following, we watch the explicit task created and executed for this case:

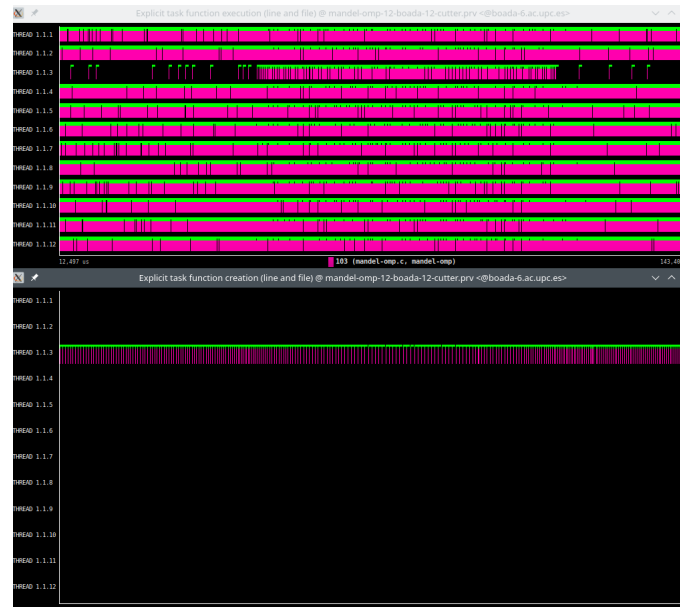
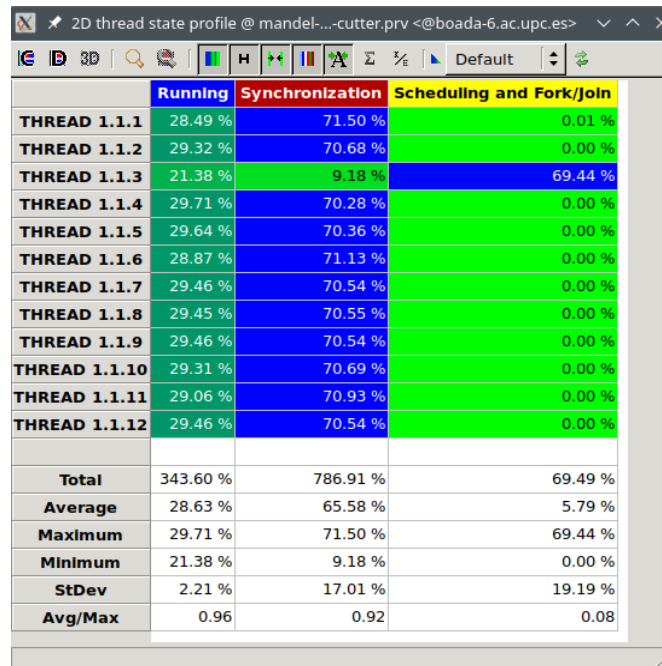


Fig. 21 Explicit task function execution and explicit task function creation graphs

We can see that thread 3 is the one who is creating all the explicit tasks, which are executed by all the threads.

Looking at the state profile we can see the same as the previous case, but in this case the thread that is creating tasks also has a percentage in synchronization, and the running percentage is quite similar to the others. That's because this thread also executes those tasks.



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	28.49 %	71.50 %	0.01 %
THREAD 1.1.2	29.32 %	70.68 %	0.00 %
THREAD 1.1.3	21.38 %	9.18 %	69.44 %
THREAD 1.1.4	29.71 %	70.28 %	0.00 %
THREAD 1.1.5	29.64 %	70.36 %	0.00 %
THREAD 1.1.6	28.87 %	71.13 %	0.00 %
THREAD 1.1.7	29.46 %	70.54 %	0.00 %
THREAD 1.1.8	29.45 %	70.55 %	0.00 %
THREAD 1.1.9	29.46 %	70.54 %	0.00 %
THREAD 1.1.10	29.31 %	70.69 %	0.00 %
THREAD 1.1.11	29.06 %	70.93 %	0.00 %
THREAD 1.1.12	29.46 %	70.54 %	0.00 %
Total	343.60 %	786.91 %	69.49 %
Average	28.63 %	65.58 %	5.79 %
Maximum	29.71 %	71.50 %	69.44 %
Minimum	21.38 %	9.18 %	0.00 %
StDev	2.21 %	17.01 %	19.19 %
Avg/Max	0.96	0.92	0.08

Fig. 22 Thread state profile

Then we see the histogram:

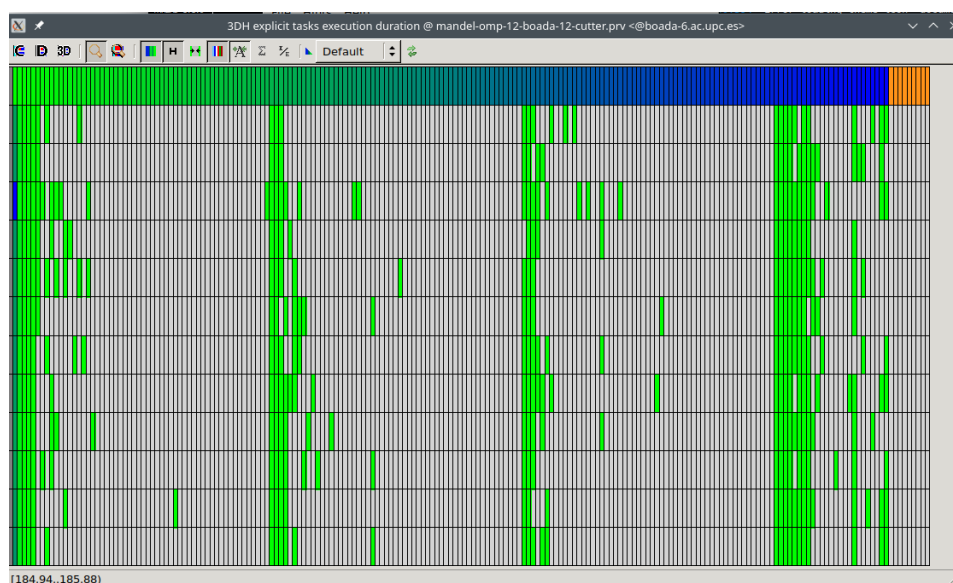


Fig. 23 Histogram

We can observe that the execution is done in loops, as we added the taskloop line.

3.3 Taskloops without nogroup

In this case, we added “nogroup” in the taskloop line between the loops.

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

Fig. 24 Piece of code of mandel_omp.c

Next, as in the previous case, we look at the scalability of this code:

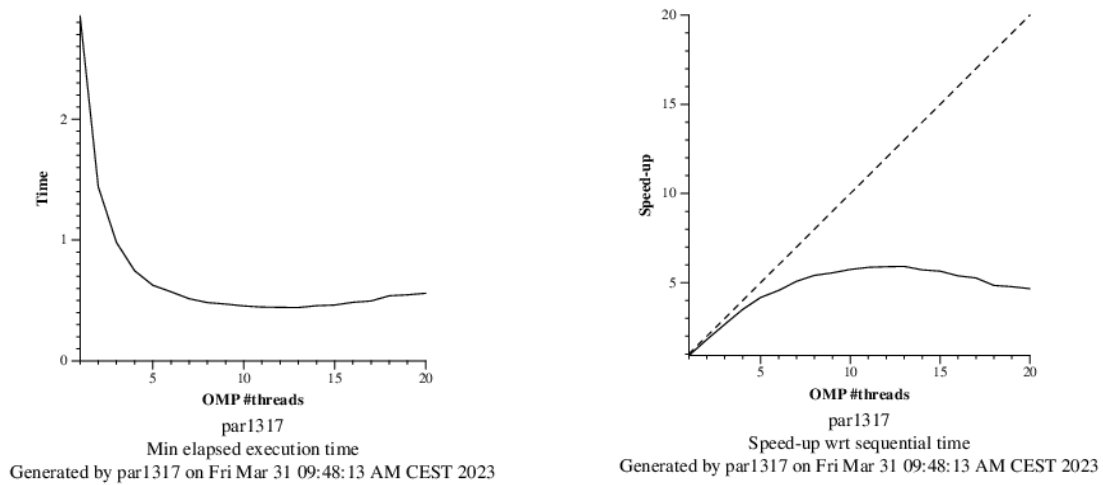


Fig. 25 Time and speed-up depending on threads

We can see that the time is constant when we use more than 10 threads. Talking about the speed-up, it's practically the same as in the previous case. When we use more than 10 threads, it keeps constant and goes a little bit down.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.46	0.13	0.11	0.14	0.18
Speedup	1.00	3.58	4.16	3.34	2.55
Efficiency	1.00	0.90	0.52	0.28	0.16

Table 1: Analysis done on Fri Mar 31 09:48:28 AM CEST 2023, par1317

Overview of the Efficiency metrics in parallel fraction, $\phi=99.89\%$					
Number of processors	1	4	8	12	16
Global efficiency	99.71%	89.43%	51.96%	27.79%	15.89%
Parallelization strategy efficiency	99.71%	93.25%	58.46%	32.57%	18.98%
Load balancing	100.00%	98.56%	98.49%	97.12%	95.95%
In execution efficiency	99.71%	94.61%	59.36%	33.53%	19.78%
Scalability for computation tasks	100.00%	95.91%	88.88%	85.32%	83.70%
IPC scalability	100.00%	98.51%	97.38%	97.11%	96.34%
Instruction scalability	100.00%	99.46%	98.74%	98.02%	97.34%
Frequency scalability	100.00%	97.90%	92.43%	89.63%	89.26%

Table 2: Analysis done on Fri Mar 31 09:48:28 AM CEST 2023, par1317

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	0.53	0.81	0.66	0.91
LB (time executing explicit tasks)	1.0	0.98	0.98	0.97	0.96
Time per explicit task (average us)	143.51	37.39	20.18	14.01	10.71
Overhead per explicit task (synch %)	0.0	4.77	61.13	188.49	398.98
Overhead per explicit task (sched %)	0.29	2.47	9.97	18.73	28.28
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Fri Mar 31 09:48:28 AM CEST 2023, par1317

Fig. 26 Modelfactor tables

We can see that the efficiency stills being not that good using taskloop with nogroup.

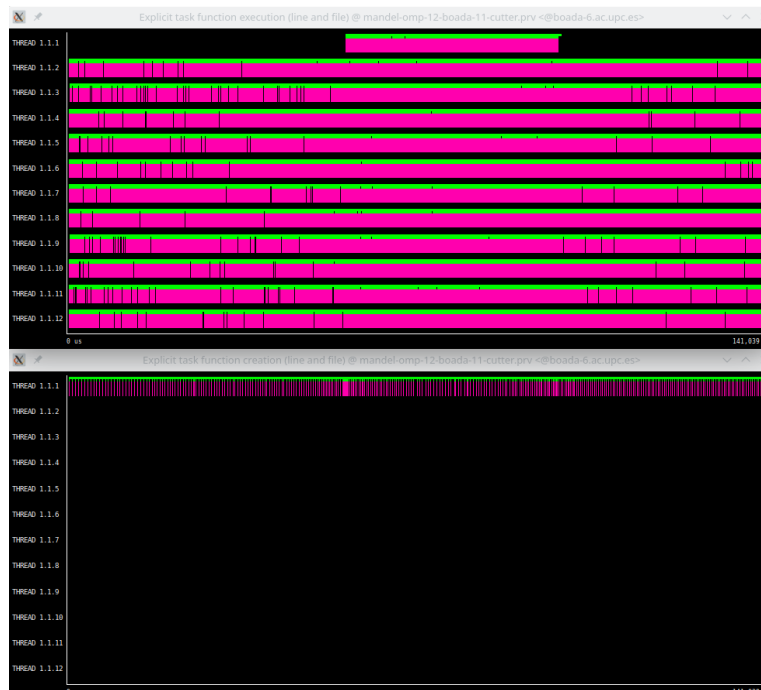


Fig. 27 Explicit task function execution and explicit task function creation graphs

Now, looking at the explicit task created and executed in the previous image, we can see that is the thread 1 the one that is creating the explicit tasks.

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	26.62 %	0.01 %	73.38 %
THREAD 1.1.2	33.20 %	66.80 %	0.00 %
THREAD 1.1.3	32.47 %	67.52 %	0.00 %
THREAD 1.1.4	32.34 %	67.66 %	0.00 %
THREAD 1.1.5	32.18 %	67.82 %	0.00 %
THREAD 1.1.6	32.45 %	67.55 %	0.00 %
THREAD 1.1.7	32.32 %	67.67 %	0.00 %
THREAD 1.1.8	32.65 %	67.35 %	0.00 %
THREAD 1.1.9	31.75 %	68.25 %	0.00 %
THREAD 1.1.10	32.36 %	67.64 %	0.00 %
THREAD 1.1.11	32.12 %	67.88 %	0.00 %
THREAD 1.1.12	32.32 %	67.67 %	0.00 %
Total	382.77 %	743.82 %	73.41 %
Average	31.90 %	61.98 %	6.12 %
Maximum	33.20 %	68.25 %	73.38 %
Minimum	26.62 %	0.01 %	0.00 %
StDev	1.62 %	18.69 %	20.28 %
Avg/Max	0.96	0.91	0.08

Fig. 28 Thread state profile

In the state profile, we can observe an improvement of the running percentage, as it's a bit bigger than in the previous case (382,77%, the other one was 343,60%).

Finally, we watch the histogram, and we observe the same sequence as the other case. The tasks are executed in loops.

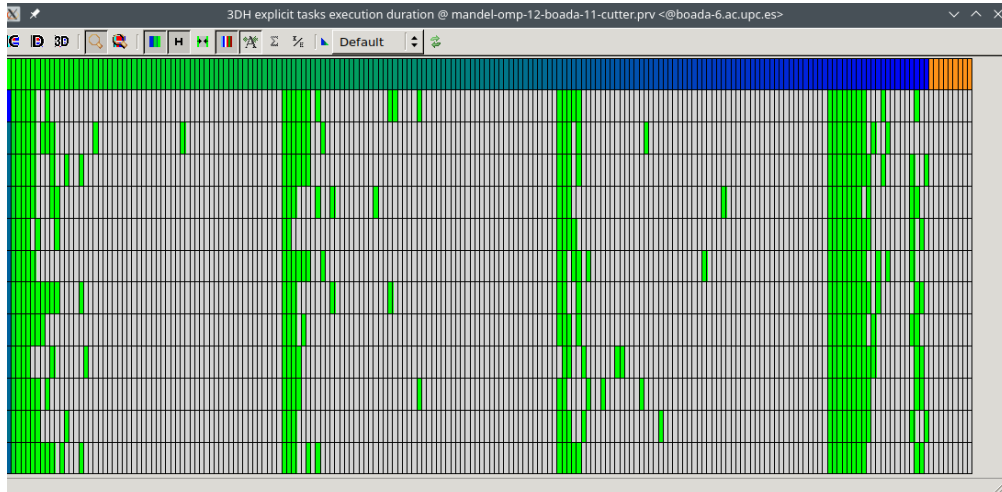


Fig. 29 Histogram

4. Row decomposition strategy

Finally, in the row decomposition strategy, we added the “#pragma omp taskloop” out of the outer loop

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

```

Fig. 30 Piece of code of mandel_omp.c

Now, we look at the scalability of the row decomposition strategy:

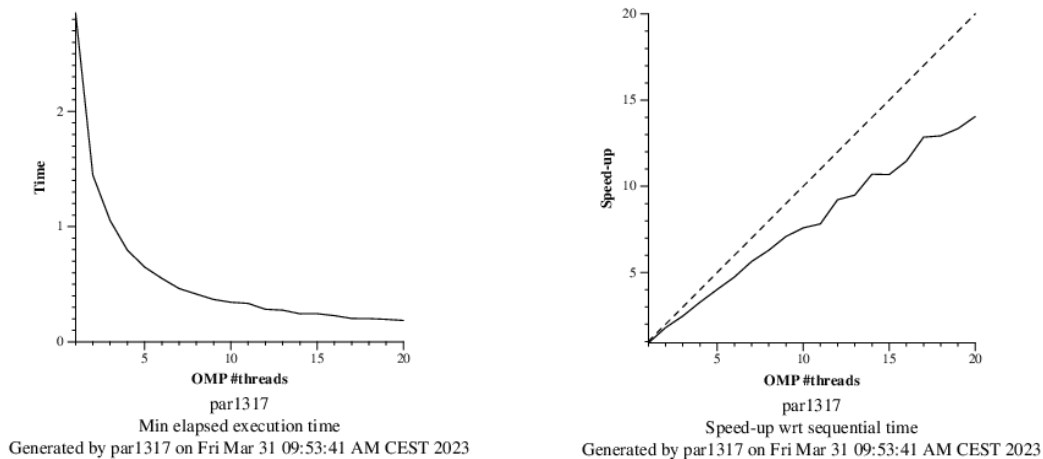


Fig. 31 Time and speed-up depending on threads

We can see a big improvement in the scalability. The speed-up is almost the ideal, it stays linear and grows almost constant when we add more threads. This is a big improvement respect the point decomposition.

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	3.50	6.65	9.51	12.29
Efficiency	1.00	0.88	0.83	0.79	0.77

Table 1: Analysis done on Fri Mar 31 09:53:43 AM CEST 2023, par1317

Overview of the Efficiency metrics in parallel fraction, $\phi=99.89\%$					
Number of processors	1	4	8	12	16
Global efficiency	99.98%	87.71%	83.30%	79.57%	77.33%
Parallelization strategy efficiency	99.98%	90.13%	92.65%	90.56%	89.00%
Load balancing	100.00%	90.20%	92.78%	90.77%	89.30%
In execution efficiency	99.98%	99.92%	99.85%	99.77%	99.66%
Scalability for computation tasks	100.00%	97.31%	89.91%	87.86%	86.89%
IPC scalability	100.00%	98.46%	97.58%	96.95%	95.90%
Instruction scalability	100.00%	100.00%	99.99%	99.99%	99.99%
Frequency scalability	100.00%	98.84%	92.14%	90.63%	90.61%

Table 2: Analysis done on Fri Mar 31 09:53:43 AM CEST 2023, par1317

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	0.9	0.93	0.91	0.89
Time per explicit task (average us)	45741.51	11751.5	6359.18	4337.68	3289.1
Overhead per explicit task (synch %)	0.0	10.91	7.89	10.36	12.25
Overhead per explicit task (sched %)	0.02	0.02	0.03	0.04	0.08
Number of taskwait/taskgroup (total)	1.0	1.0	1.0	1.0	1.0

Table 3: Analysis done on Fri Mar 31 09:53:43 AM CEST 2023, par1317

Fig. 32 Modelfactor tables

In this case, we can observe that using row decomposition the efficiency is a lot better, bigger than 0,77 using 16 threads or less. We can also see that the number of taskwait/taskgroup it's only 1.

Next, we see again the explicit task created and executed:

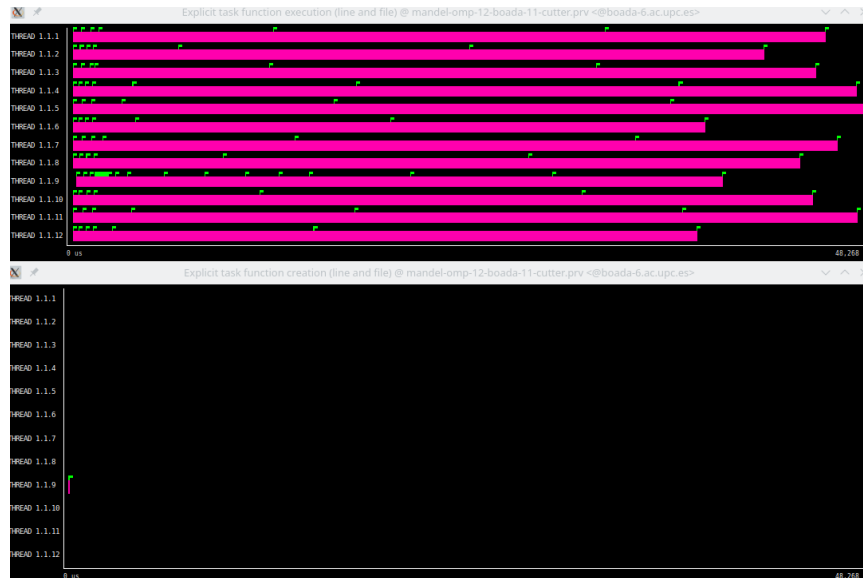


Fig. 33 Explicit task function execution and explicit task function creation graphs

In this case, we can see that the cost of completing each row iteration is not always the same, the load balance can't be controlled and there is a chance where some threads get more workload than others. The black region at the end represents a region where a thread is waiting for others to finish their task.

Finally, we watch the state profile:

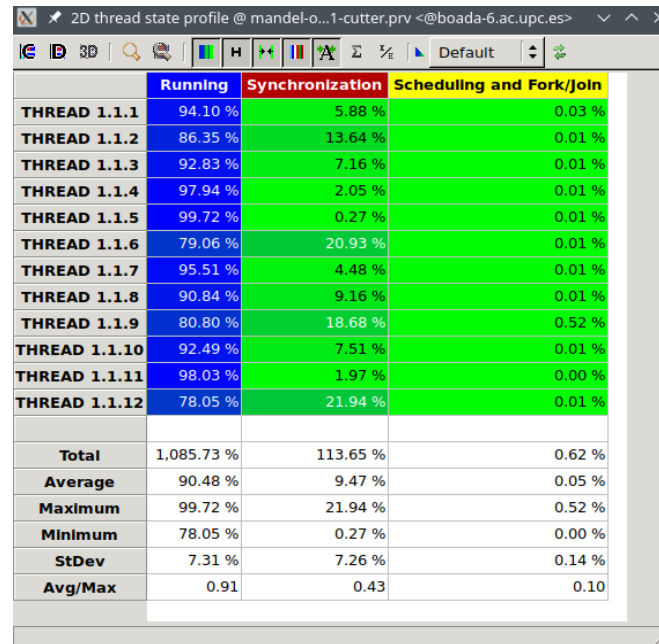


Fig. 34 Thread state profile

Here we can observe a big improvement in the running percentage of the threads, where they are almost at 100% running tasks.

Finally, we take a look at the histogram:

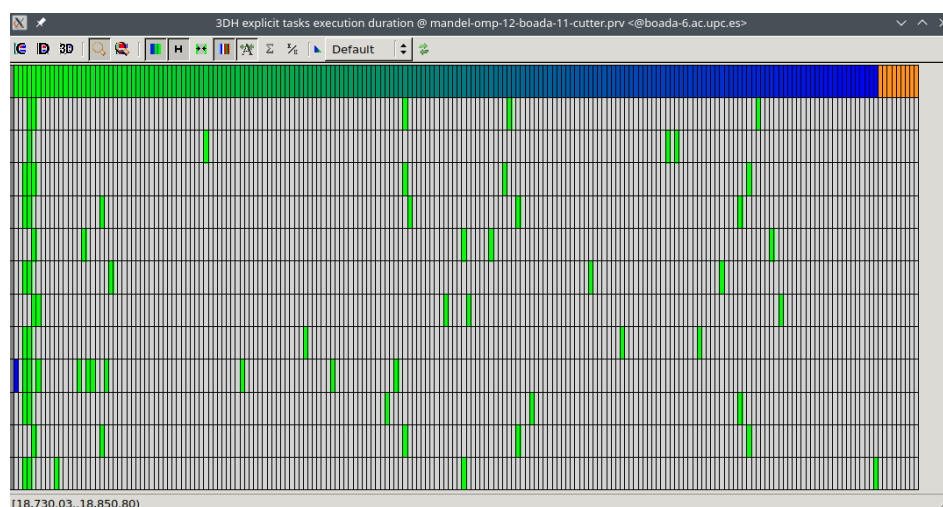


Fig. 35 Histogram

We can observe that the execution of the tasks are not organized in loops like the previous cases, only at the start. That's because the taskloop is in the outer loop of the program.

5. CONCLUSIONS

Through these two weeks doing this laboratory part, we have explored different ways to achieve parallelism for the Mandelbrot set computation, from a point strategy to a row strategy.

First of all, we used the parador tool to generate graphs that show dependencies between tasks. This helped us to study those dependencies to see the potential of both strategies, point and row strategy.

In the next section, we explored the different options that OpenOMP gives us to control the execution of the script.

Once we modified the script with the OpenOMP options, we looked for the scalability.

Later, we learned about the taskloop directive, which generates tasks out of the loop iterations. For the taskloop, we detected that a lot of overhead was produced due to synchronization and scheduling of tasks. Then, using taskloop nogroup, we manage to reduce that overhead.

Finally, we used the row strategy decomposition and we could see that it is a lot more efficient than the point strategy, it has a better performance and its scalability is near to the ideal scalability.

In this assignment we have seen the importance of knowing different ways to parallelize the tasks and compare them to see the best option to choose. We also have learned a lot of tools and options that help us to see how the parallelization works, how to improve it and compare different options.