PAR Laboratory Assignment

Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

PAR2303
Joan Manuel Ramos Refusta
Àlex Aguilera Martínez

1. Task decomposition and granularity analysis

In this part, we are going to explain the different task decomposition strategies and granularities explored with Tareador using the small test case -w 8. We are asked to change the sequential mandel-tar.c code for two possible task granularities in order to see the potential parallelism.

We have four versions to see here. Two versions row and point with display and two versions more without the display.

In the first case (row), a task corresponds with the computation of a whole row. Therefore, we have to modify the code introducing the tareador_start_task() between the loops and the tareador_end_task() at the end of the first for.

```
{
    /* Calculate points and save/display */
    for (row = 0; row < height; ++row) {
        tareador_start_task("Mandelbrot Row");
        for (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);

#if _DISPLAY_
            /* 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);
            }
#else
        output[row][col]=k;
#endif
        }
        tareador_end_task("Mandelbrot Row");
    }
}
```

Figure 1, code of mandel-tar-**row** tasks creation

We obtain the following dependence graph for the version without the display:

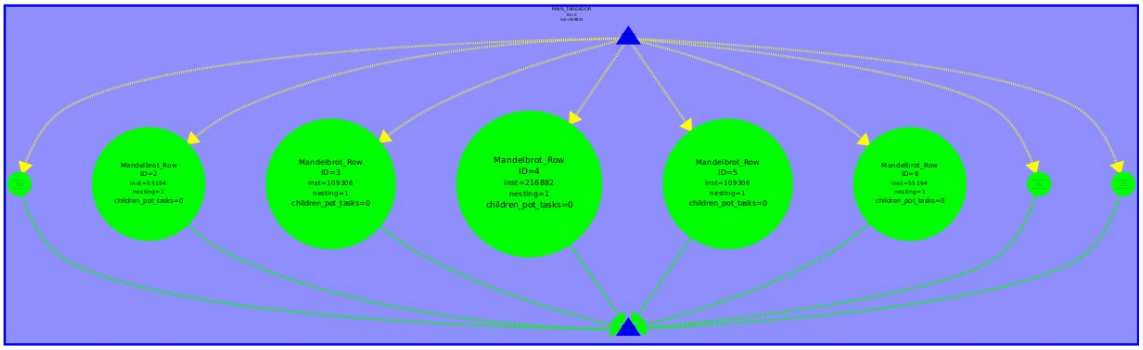

Figure 2, mandel-tar (row) dependence graph without display

However, we obtain a very different dependence graph with the display. You can see it in figure 3.

In the second case (point), a task corresponds with the computation of a single point. Therefore, we have to modify the code introducing the tareador_start_task() after the loops and the tareador_end_task() at the end of the second for.

```
{
    /* Calculate points and save/display */
    for (row = 0; row < height; ++row) {
        for (col = 0; col < width; ++col) {
            tareador_start_task("Mandelbrot 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);

#if _DISPLAY_
            /* 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);
            }
#else
        output[row][col]=k;
#endif
        tareador_end_task("Mandelbrot Point");
        }
    }
}
```

Figure 4, code of mandel-tar-**point** tasks creation

Figure 3

We obtain the following dependence graph without the display:

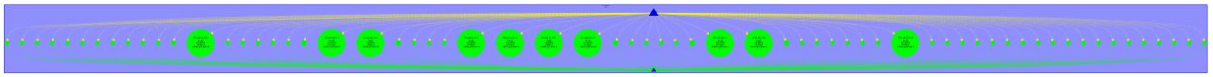

Figure 5, mandel-tar (point) dependence graph without display

Nevertheless, for the version with display, we get the dependence graph of figure 6.

As important common characteristics after analysing all the dependence graphs, firstly, we can see that the amount of work isn't distributed equally between all the tasks. (green nodes) The nodes which are in the middle usually execute more instructions, which means more amount of work.

Secondly, without display, we can see in figure 2 and 5 that there isn't any dependence between the created tasks (green nodes). Moreover, all the tasks have a dependence of the main task (the blue one on the top) and the last task has dependences of all the tasks. However, with display, the graph changes a lot. We can see in figure 3 and 6 that each task depends on the last one, it is like a sequential program. This result is due to the extra library to display the Mandelbrot set which needs more dependencies between the tasks than the version without display.

We found the serialization in the following part of the code of mandel-tar.c:

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

Figure 7, serialization part of the code mandel-tar.c

If we want to protect this section in parallel with OpenMP, we should use #pragma omp critical, in order to define a region where only one thread is working at the same time.

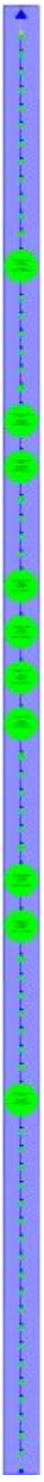

Figure 6

2. Point decomposition in OpenMP

In this section of the deliverable, we will explore the different options in the OpenMP tasking model to express the Point decomposition for the Mandelbrot computation program. We have to see 5 versions of point decomposition with different code. Let's start with version 1.

Version 1:

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

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

Figure 7, code of the mandel-omp.c (original version)

The generated image for the execution with 1 thread is correct, the sequential version is a little bit faster than the execution with 1 thread. With 8 threads, the execution time decreases.

Execution time for 1 thread: 3.32 seconds.
Execution time for 8 thread: 1.23 seconds.
Execution time for serial version: 3.05 seconds.

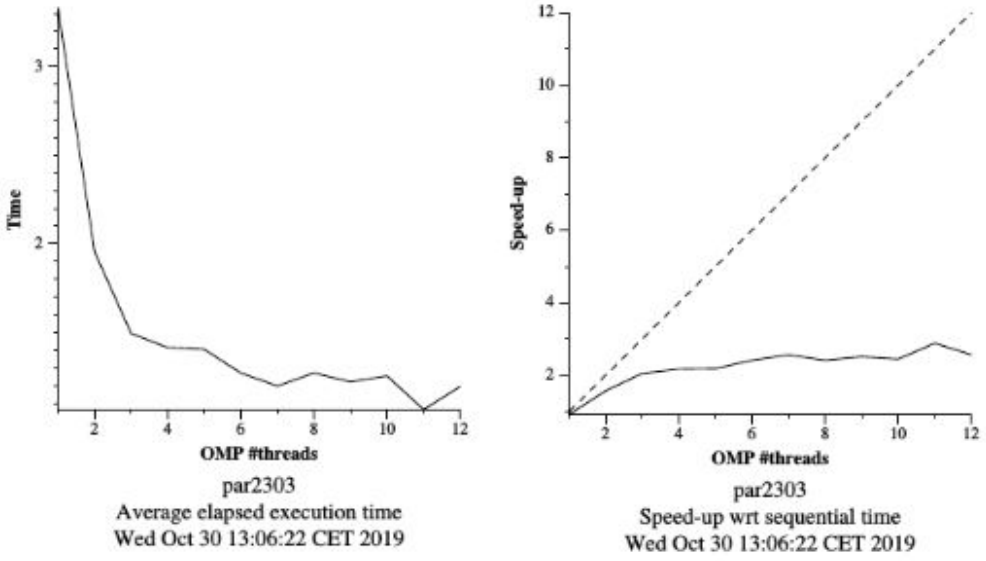

Figure 7, scalability graphs of mandel-omp-point (Version 1)

Observing figure 7, we can see that the speedup increases slowly at the time that the time that the number of threads increases and reaches the maximum speed-up at 11 threads, where the the execution time is the shortest.

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	78,006	190,400
THREAD 1.1.2	77,250	4,800
THREAD 1.1.3	80,604	16,800
THREAD 1.1.4	84,206	307,200
THREAD 1.1.5	74,187	40,800
THREAD 1.1.6	82,995	-
THREAD 1.1.7	81,639	4,000
THREAD 1.1.8	81,113	76,000
Total	640,000	640,000
Average	80,000	91,428.57
Maximum	84,206	307,200
Minimum	74,187	4,000
StDev	3,087.88	106,818.69
Avg/Max	0.95	0.30

Figure 8, task profile generated for mandel-omp with 8 threads (Version 1)

By looking at table 8, we can appreciate that tasks are not always created by the same thread. This job is performed by the first thread available that reaches the beginning of the parallel.

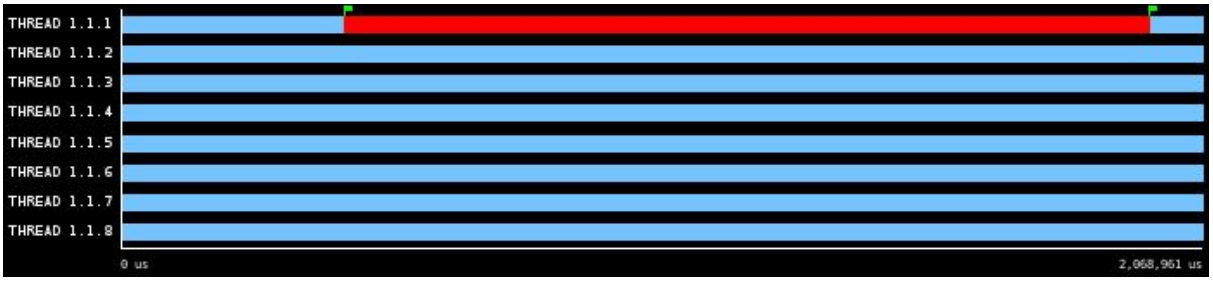

Figure 9, Parallel construct of mandel-omp.c with 8 threads.

As we can see at figure 9, the only thread that executes parallel construct is the first thread of all.

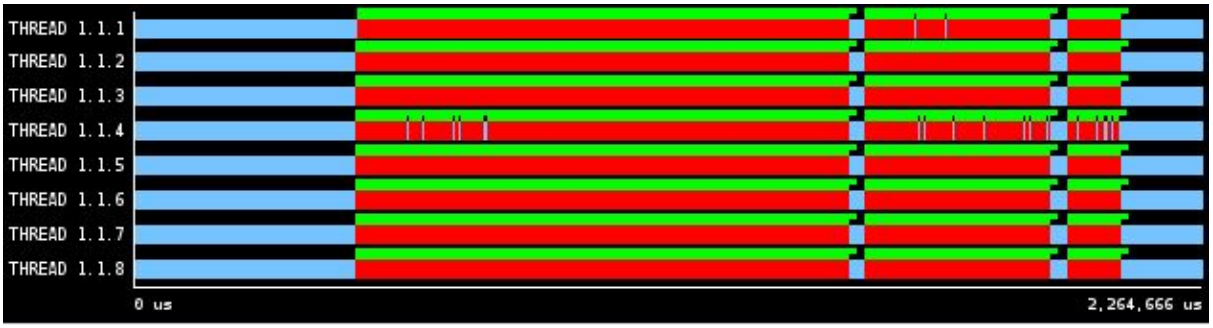

Figure 10, task execution of mandel-omp.c with 8 threads.

Version 2:

Now, let's try to make use of taskwait clause. We are going to use the following code:

```
#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)
        {
            (...)
        }
    }
    #pragma omp taskwait // waiting point for all child tasks
}
```

Figure 11, code of the mandel-omp-pointv2.c

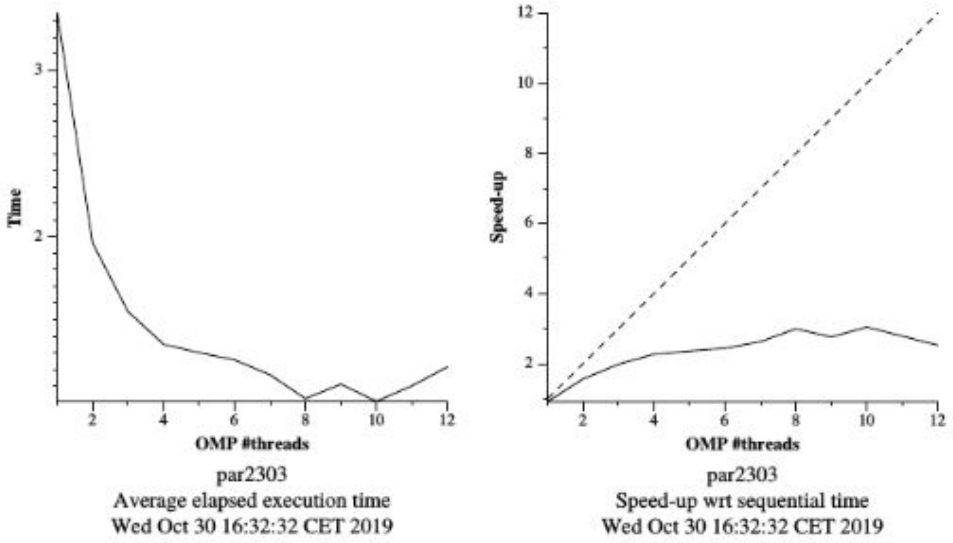

Figure 12, scalability graphs of mandel-omp-point with taskwait (Version 2)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	88,702	640,000
THREAD 1.1.2	80,648	-
THREAD 1.1.3	76,031	-
THREAD 1.1.4	77,654	-
THREAD 1.1.5	81,586	-
THREAD 1.1.6	79,503	-
THREAD 1.1.7	78,379	-
THREAD 1.1.8	77,497	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	88,702	640,000
Minimum	76,031	640,000
StDev	3,692.16	0
Avg/Max	0.90	1

Figure 13, task profile for mandel-omp-pointv2.c

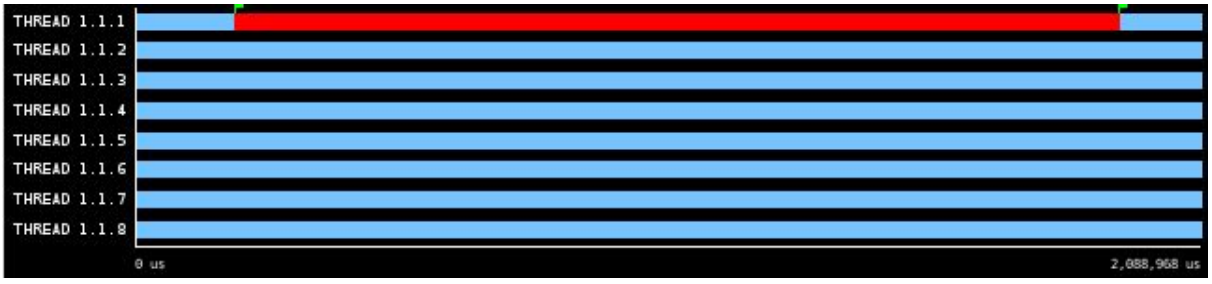

Figure 14, Parallel constructs of mandel-omp-pointv2.c

Again, the parallel construct happens at the first thread. There are 640.000 tasks and 800 taskwaits executions (one for every row) as we can see at figure 15.

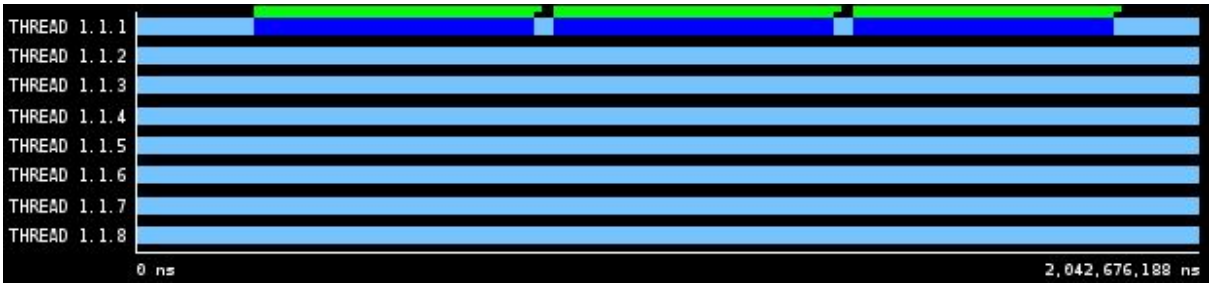

Figure 15, Taskwait constructs of mandel-omp-pointv2.c

Version 3:

Now, we can try to use taskgroup in order to see if there is any difference between the last versions. We are going to use the code which has been provided to us in this laboratory assignment.

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskgroup
    {
        for (int col = 0; col < width; ++col) {
                #pragma omp task firstprivate(row, col)
                {
                    (...)
                }
        }
    } //waiting point for all descendant tasks in taskgroup region
}
```

Figure 16, code of mandel-omp-pointv3.c

In this version there is no difference in the amount of tasks created, the distribution of executed tasks among threads or execution time compared to the previous version. That's because in version 2 we were creating a taskgroup manually with taskwaits, and for version 3 we are using the taskgroup.

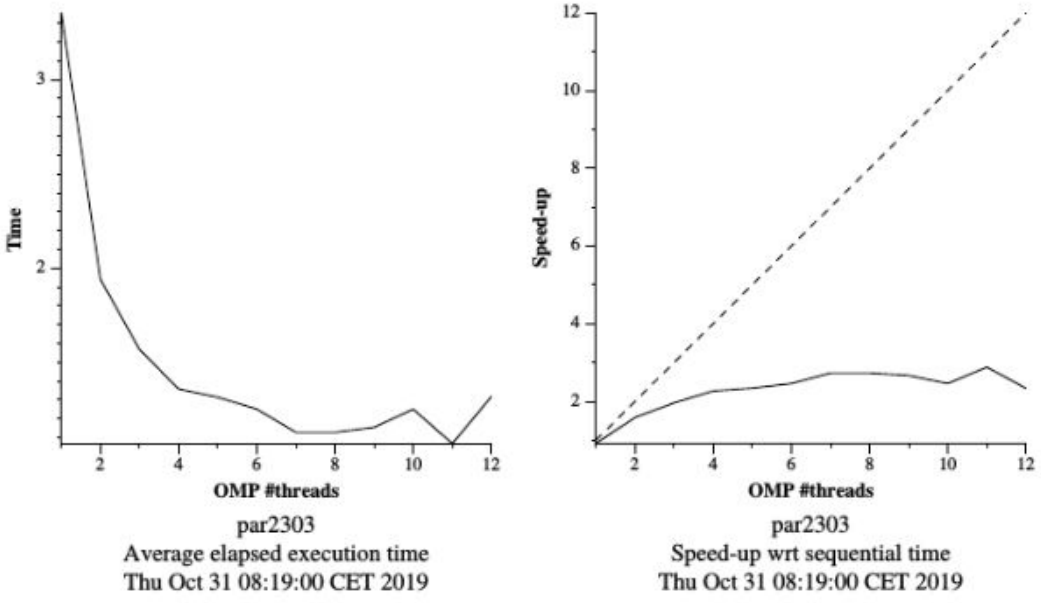

Figure 17, scalability graphs of mandel-omp with taskgroup (version 3)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	89,597	640,000
THREAD 1.1.2	81,752	-
THREAD 1.1.3	74,189	-
THREAD 1.1.4	76,781	-
THREAD 1.1.5	83,387	-
THREAD 1.1.6	75,302	-
THREAD 1.1.7	82,941	-
THREAD 1.1.8	76,051	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	89,597	640,000
Minimum	74,189	640,000
StDev	4,963.47	0
Avg/Max	0.89	1

Figure 18, task profile for mandel-omp-pointv3.c

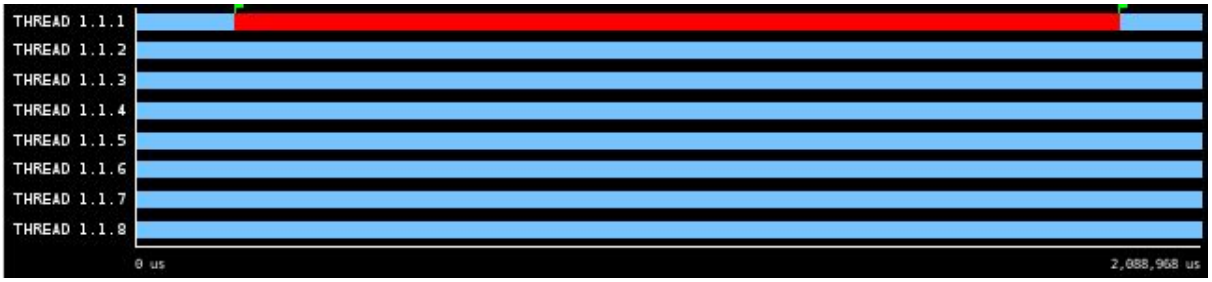

Figure 19, Parallel constructs of mandel-omp-pointv3.c

Figure 20, Taskgroup constructs of mandel-omp-pointv3.c

We can appreciate that the version 3 and 2 are practically the same, we can see that there is no dependencies except the one inside the critical region. The program runs slowlier with the barriers, we don't need to use taskgroup or taskwait.
As we can see in the figure 12 and 17, the strong scalability doesn't show any differences, both implementations have a bad strong scalability.

Version 4:

Finally, let's try to use the taskloop clause. Again, we are going to use the code which has been provided to us in this laboratory assignment.

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) num_tasks(800)
    for (int col = 0; col < width; ++col) {
            (...)
    }
}
```

Figure 21, code of mandel-omp-pointv4.c

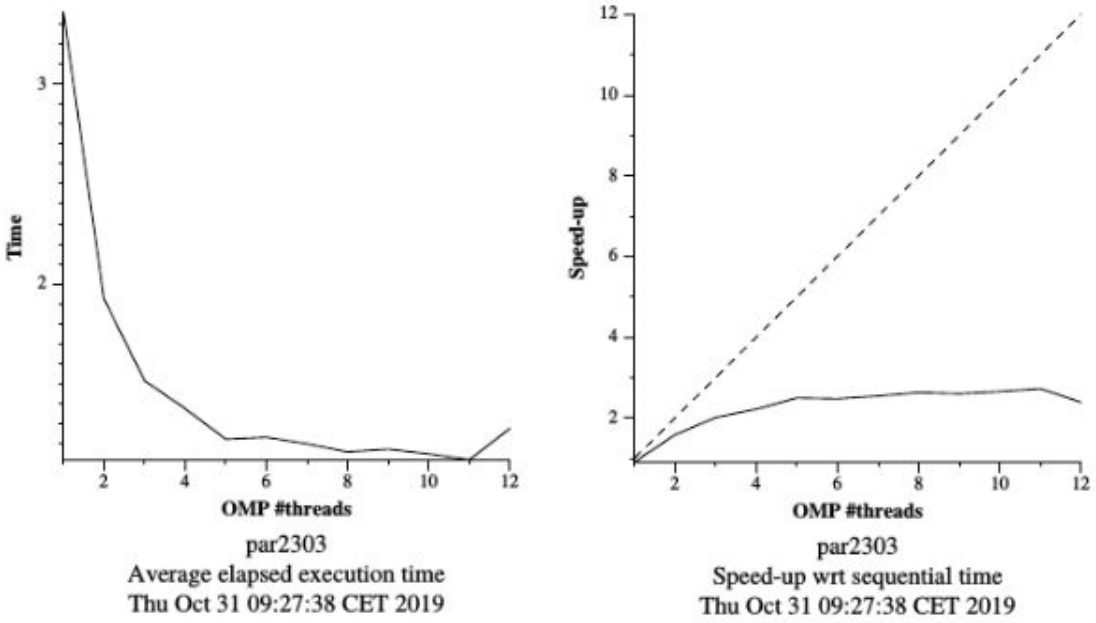

Figure 22, scalability graphs of mandel-omp-point with taskloop(version 4)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	76,262	4,360
THREAD 1.1.2	84,617	1,266
THREAD 1.1.3	82,949	1,150
THREAD 1.1.4	82,303	1,168
THREAD 1.1.5	84,523	1,216
THREAD 1.1.6	77,678	1,196
THREAD 1.1.7	79,135	1,269
THREAD 1.1.8	84,533	1,175
Total	652,000	12,800
Average	81,500	1,600
Maximum	84,617	4,360
Minimum	76,262	1,150
StDev	3,131.20	1,043.98
Avg/Max	0.96	0.37

Figure 23, task profile for mandel-omp-pointv4.c

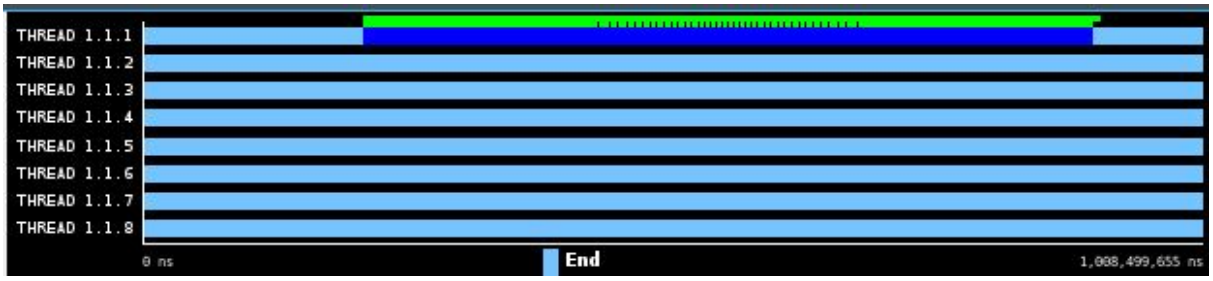

Figure 24, taskloop constructs of mandel-omp-pointv4.c

Version 5:

We will see the behaviour of a last point version using taskloop and adding the nogroup clause, including the following:

#pragma omp taskloop firstprivate(row) num_tasks(800) **nogroup**

We obtain the following results:

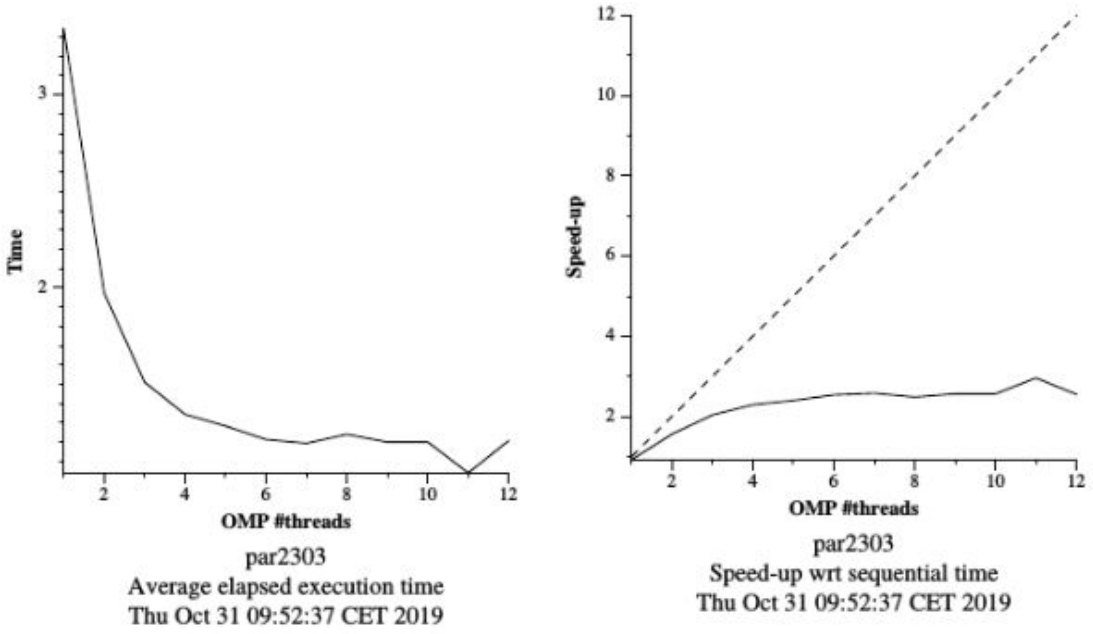

Figure 25, scalability graphs of mandel-omp-point with taskloop+nogroup (version 5)

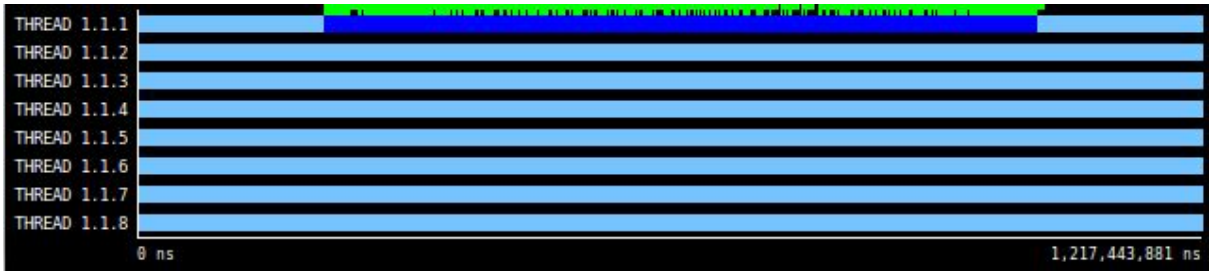

Figure 26, taskloop constructs of mandel-omp-pointv5.c

We can't see a much better performance in comparison with version 5 (only taskloop, without nogroup clause)

Now, we have to explore how this nogroup version behaves in terms of performance using 8 threads and different task granularities. We obtain the following table and plot:

Num_tasks/iteration	Time (s)
1	0.029794
2	0.031730
5	0.035960
10	0.056303
25	0.061605
50	0.140567
100	0.254491
200	0.275317
400	0.242464
800	0.172779

Figure 27, table of the execution time in seconds per num_tasks/iteration

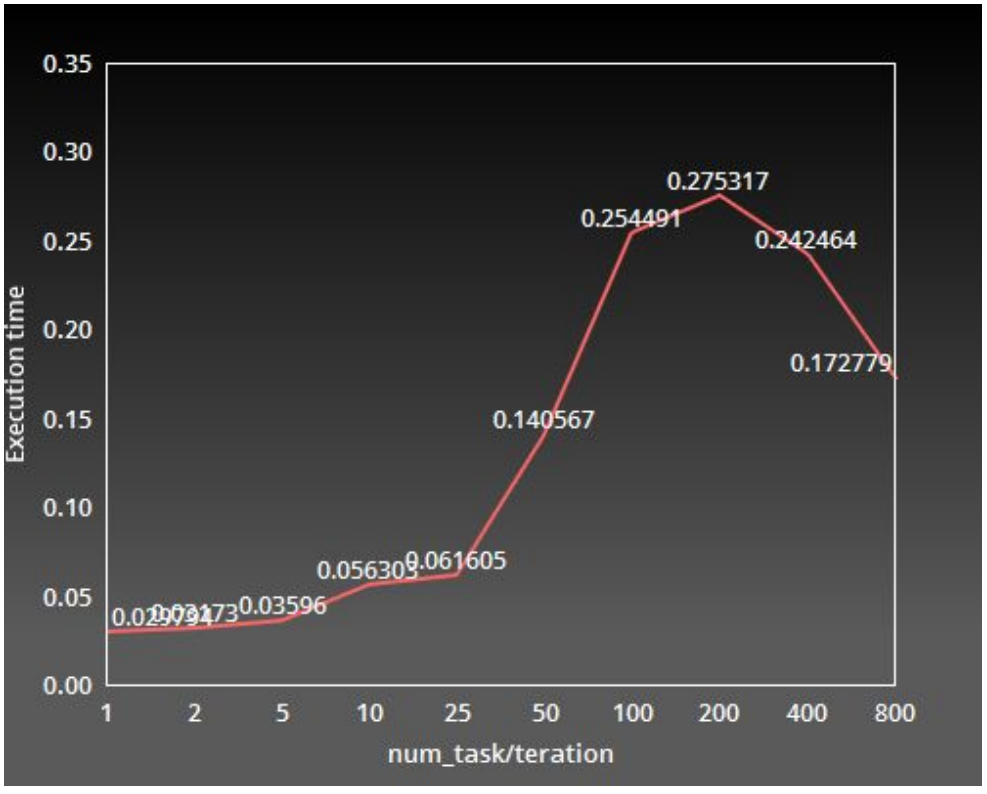

Figure 28, plot of the execution time per num_tasks/iteration

We can observe by the table and the respective plot that when the execution time reaches the worst value, and when the num_tasks/iteration decreases, the execution time decreases too.

3. Row decomposition in OpenMP

In this section of the deliverable, we will explore the row decomposition. We are going to use the following row decomposition code:

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

Figure 29, code of mandel-omp-row.c

We obtain the task profile of figure 30 which we can see how the total number of tasks decrease a lot in comparison with the last versions of the deliverable. (point versions)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	312	800
THREAD 1.1.2	70	-
THREAD 1.1.3	78	-
THREAD 1.1.4	62	-
THREAD 1.1.5	79	-
THREAD 1.1.6	63	-
THREAD 1.1.7	67	-
THREAD 1.1.8	69	-
Total	800	800
Average	100	800
Maximum	312	800
Minimum	62	800
StDev	80.34	0
Avg/Max	0.32	1

Figure 30, task profile of mendel-omp-row

However, it's quite obvious because we are only creating tasks with the first loop. The granularity is less finer than the point version.

Moreover, in figure 31, we can see how a little less than the half of the time, the program is executing sequentially (only thread 1 is doing something). This is quite different than the point versions.

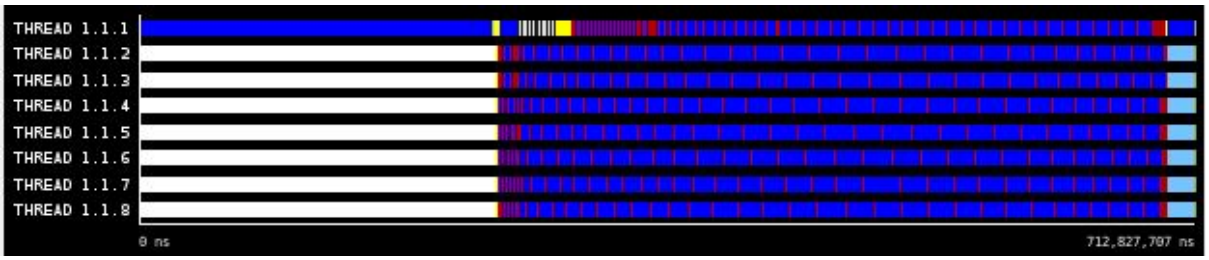

Figure 31, parallel execution timeline of row version

Finally, we have the scalability graphs in figure 32. We can't see any important difference between row and point versions, it's quite the same. When the number of threads increases, the execution time tends to decrease and the speedup tends to increase. Nothing out of the ordinary.

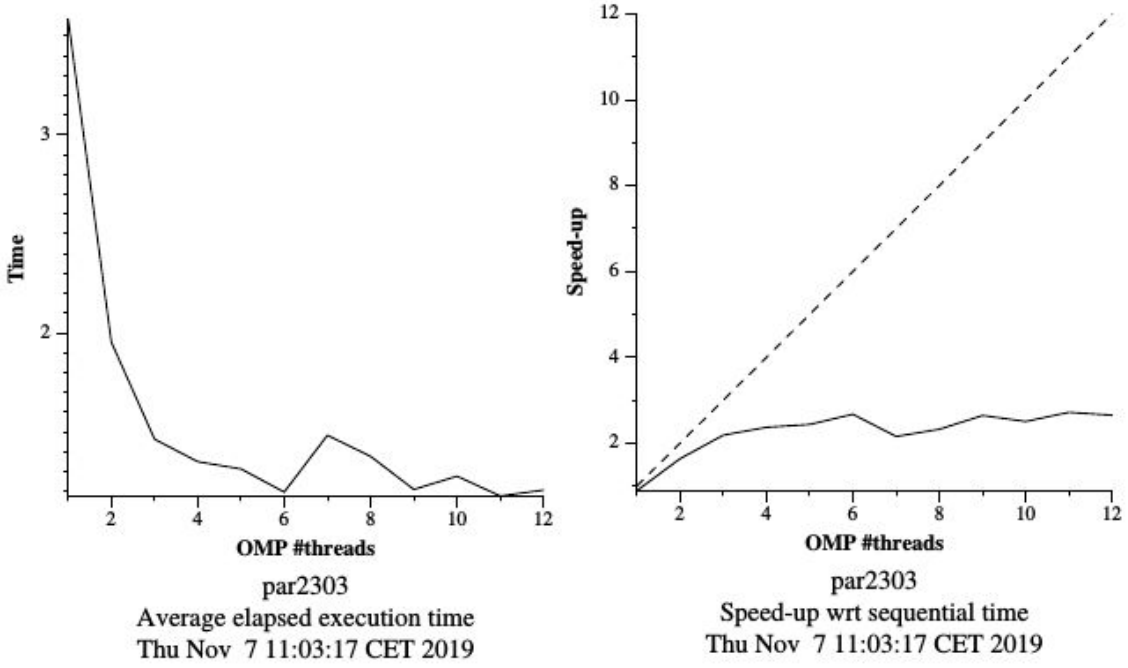

Figure 32, scalability graphs of mandel-omp-row

4. Optional: for–based parallelization

In this final section of the deliverable, we are going to make use of the for work–sharing construct in order to distribute the work among the threads in the team created in the parallel clause.

Therefore, we are going to start from the code of figure 33 and we are going to use different schedule kinds of OpenMP (static, dynamic and guided) in order to observe how they treat the load balancing problem in the Mandelbrot program.

```
#pragma omp parallel for schedule(static)
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        (...)
    }
}
```

Figure 33, code of mandel-omp-optionalpart.c (using static)

We obtain the following table specifying the time execution of each mode with static, guided or dynamic:

Schedule mode	Chunk	Time (seconds)
Static	No specified, so N/num_threads	0.118488s
Static	10	0.049700s
Dynamic	No specified, so 1	0.031871s
Dynamic	10	0.044703s
Guided	No specified, so 1	0.031873s
Guided	10	0.036634s

Figure 34, table of the execution time in seconds with each schedule option

In the document Short tutorial on OpenMP 4.5, explains that if the N is not specified happens what is in the table.

As we can see in the table of figure 34, the worst mode is the static with default chunk. However, if we specify the chunk (in this example, 10), we obtain a much better time execution of the code.

And, concerning dynamic and guided scheduling, we can't see any important difference between them. It is true that when the chunk is not specify, the time executing seems to be better, but we are talking about a very little difference.

Furthermore, in order to see the behaviour of the different modes when the number of threads increases (how scales), we thought that we can maybe generate the scalability graphs of each mode and see something else. However, all the graphs are very similar and we couldn't see any important difference to comment. You can see it in figure 35 and 36 with guided(10) and the worst mode that is the static (default), few differences.

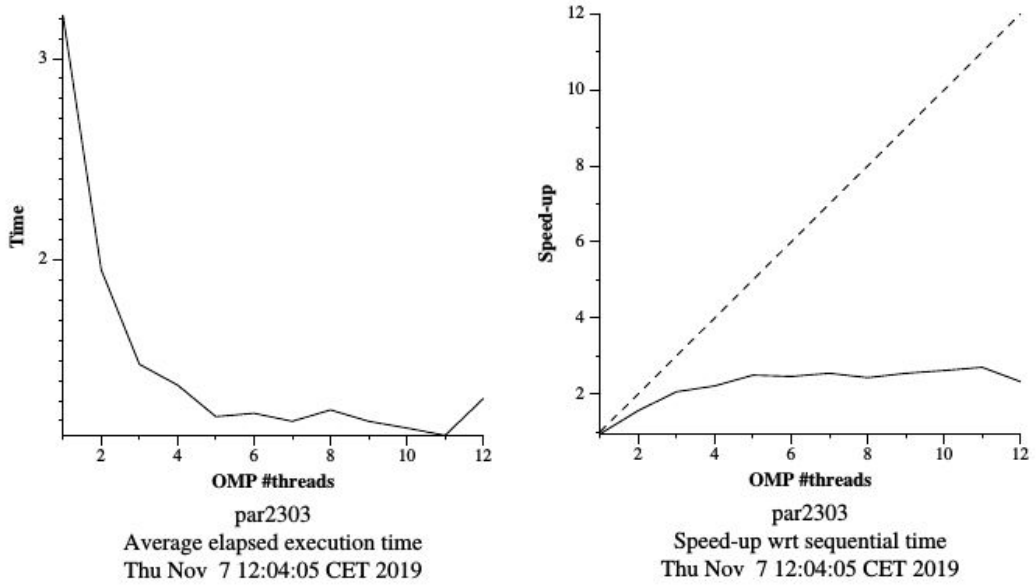

Figure 35, scalability graphs of mandel-omp-optionalpart (guided,10)

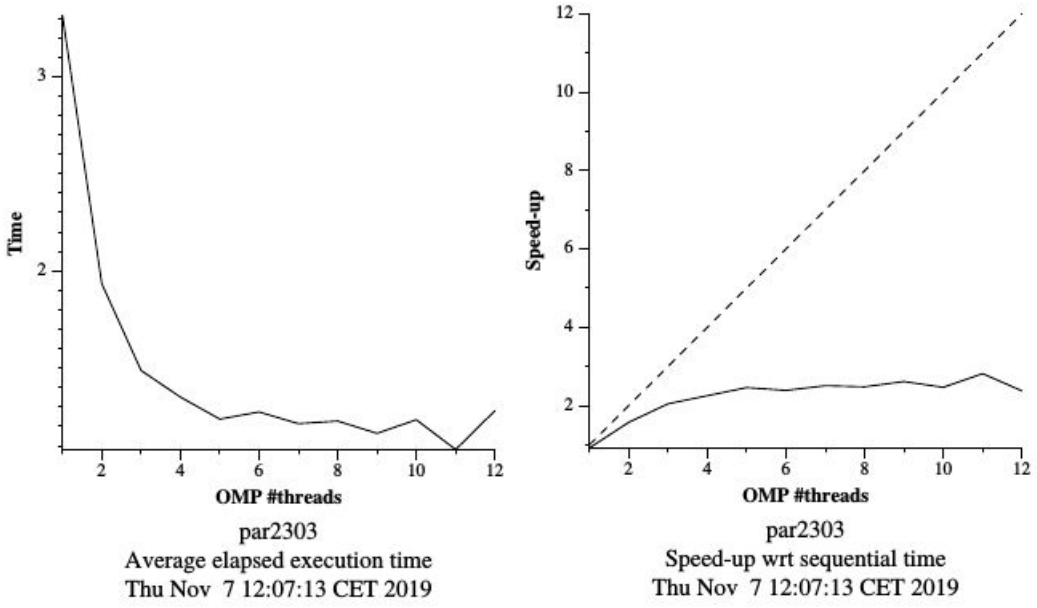

Figure 36, scalability graphs of mandel-omp-optionalpart (static,default)

5. Conclusions

In this laboratory assignment, we have learned how the different OpenMP clauses, which we learned in the last laboratory assignment, but we hadn't used them yet in a big program, can improve or worsen our parallel program.

Moreover, we have learned all of this using two possible task granularities in order to see the potential parallelism, although we have focused more in the point version. The first part of the assignment was very useful to see the dependencies between the tasks using tareador and the different granularities that we have implemented later with OpenMP.

We have to say that we haven't seen a much better performance using the different OpenMP clauses with point granularity, all the 5 versions were quite similar. However, in comparison with the row version, we could see some difference.

Finally, the optional part, we think that it has been a good practice to see how different schedulings (static, dynamic and guided) treat the load balancing problem in our program.

To sum up, we think that we have learned and have used new interesting things about parallelism which we didn't know and we will probably need to use in the following laboratory assignments.